Introduction to VMS System Services

Order Number: AA-LA68B-TE

November 1991

This manual describes how to use the VMS system services.

Revision/Update Information: This manual supersedes the *Introduction to VMS System Services*, Version 5.4.

Software Version: VMS Version 5.5

Digital Equipment Corporation
Maynard, Massachusetts

November 1991

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK4589

This document was prepared using VAX DOCUMENT, Version 1.2

Contents

3 Security Services

4 Event Flag Services

5 AST (Asynchronous System Trap) Services

6 Name Services

7 Input/Output Services

8 Process Control Services

9 Process Information Services

10 Timer and Time Conversion Services

11 Condition-Handling Services

12 Memory Management Services

13 Lock Management Services

14 DECdtm Services

15 Programming Examples

A User-Written System Services

Figures

Tables

Preface

This manual provides guidelines for how to use the system services on a VMS operating system.

You can use VMS system services only in programs written in languages that produce native code for the VAX hardware. At present these languages include VAX MACRO and the following high-level languages:

VAX Ada
VAX BASIC
VAX BLISS–32
VAX C
VAX COBOL
VAX COBOL–74
VAX CORAL
VAX DIBOL
VAX FORTRAN
VAX Pascal
VAX PL/1

Intended Audience

This manual is intended for system and application programmers who want to call system services.

Document Structure

This manual is organized as follows:

- Chapter 1 introduces the system services. It presents overviews of the categories of system services and explains the documentation format of the service descriptions used in the *VMS System Services Reference Manual.*
- Chapter 2 describes how to call system services. It contains detailed information for the VAX MACRO programmer and general information for the high-level language programmer. For additional information about a specific high-level language and programming examples in that language, see the user's guide for that language.
- Chapters 3 through 14 guide new users in understanding how the system services work and how to use them. Each category of services has its own chapter. Examples are provided in VAX MACRO and VAX FORTRAN, although they are explained in a way meaningful to all high-level language programmers.
- Chapter 15 contains sample programs that use various system services.
- Appendix A contains information about how you can code your own system services.

- Appendix B provides a programmer's guide for using shared memory.
- Appendix C provides instructions for implementing site-specific security policies.

Associated Documents

For a detailed description of each system service routine, see the *VMS System Services Reference Manual.*

The VAX Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines*, contains useful information for anyone who wants to call system services.

VAX MACRO programmers can find additional information about calling system services in the *VAX MACRO and Instruction Set Reference Manual.*

High-level language programmers can find additional information about calling system services in the language reference manual and language user's guide provided with the VAX language.

The following documents may also be useful:

- *Guide to Using VMS Command Procedures*
- *Guide to VMS File Applications*
- *Guide to VMS System Security*
- *VMS Networking Manual*
- *VMS Record Management Services Manual*
- *VMS I/O User's Reference Manual: Part I*
- *VMS I/O User's Reference Manual: Part II*

For a complete list and description of the manuals in the VMS document set, see the *Overview of VMS Documentation.*

Conventions

The following conventions are 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	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1, then press and release another key or a pointing device button.
. . .	In examples, a horizontal ellipsis indicates 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.

. . .	A vertical ellipsis indicates 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 format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
red ink	Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on. For online versions of the book, user input is shown in **bold**.
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 online versions of the book.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1

Introduction to System Services

System services are procedures that the VMS operating system uses to control resources available to processes; to provide for communication among processes; and to perform basic operating system functions, such as the coordination of input/output operations.

Although most system services are used primarily by the operating system on behalf of logged-in users, they are also available for general use and provide mechanisms that you can use in application programs. For example, when you log in to the operating system, the Create Process ($CREPRC) system service is called to create a process on your behalf. You may, in turn, write a program that calls the $CREPRC system service to create a subprocess to perform certain functions for an application.

System services can be divided into functional groups. The following table lists each group of system services and its function.

Services Group	Function
Security	The security services provide various mechanisms that you can use to enhance the security of VMS operating systems.
Event Flag	A process can use event flags to synchronize sequences of operations in a program. Event flag services clear, set, and read event flags, and place a process in a wait state pending the setting of an event flag or flags.
AST	Process execution can be interrupted by events (such as I/O completion) for the execution of designated subroutines. These software interrupts are called asynchronous system traps (ASTs) because they occur asynchronously to process execution. System services are provided so that a process can control the handling of ASTs.
Logical Names	Logical name services provide a generalized technique for maintaining and accessing character string logical name and equivalence name pairs. Logical names can provide device independence for system and application program input and output operations.

Services Group	Function
Input/Output	I/O services perform input and output operations directly, rather than through the file handling services of the VMS Record Management Services (RMS). I/O services do the following: • Perform logical, physical, and virtual input and output operations • Format output lines converting binary numeric values to ASCII strings and substituting variable data in ASCII strings • Perform network operations • Send messages to system processes
Process Control	Process control services let you create, delete, and control the execution of processes.
Process Information	Process information services let you obtain information about processes.
Timer and Time Conversion	Timer services schedule program events for a particular time of the day or after a specified interval of time has elapsed. The time conversion services provide a way to obtain and format binary time values for use with the timer services.
Condition-Handling	Condition handlers are procedures that can be designated to receive control when a hardware or software exception condition occurs during image execution. Condition-handling services designate condition handlers for special purposes.
Memory Management	Memory management services provide ways to use the virtual address space available to a program. Included are services that do the following: • Allow an image to increase or decrease the amount of virtual memory data available • Control the paging and swapping of virtual memory • Create and access files in memory that contain shareable code or data
Change Mode	Change mode services alter the access mode of a process to a more privileged mode to execute particular routines, or change the stack pointer for a less privileged mode. These services are used primarily by the operating system.
Lock Management	Lock management services let cooperating processes synchronize their access to shared resources.

Services Group	Function
DECdtm	DECdtm services provide for complete and consistent executions of distributed transactions. DECdtm services coordinate distributed transactions by using the two-phase commit protocol, and by implementing special logging and communication techniques. DECdtm services do the following: • Start transactions • End transactions • Abort transactions

1.1 Documentation Format for System Service Routines

Each system service routine in the *VMS System Services Reference Manual* is documented using a structured format called the routine template. This section discusses the main headings in the routine template, the information that is presented under each heading, and the format used to present the information.

The purpose of this section is to explain where to find information and how to read it correctly, not how to use it. For a substantive discussion of the contents, meaning, and use of the information provided in the routine template, see the *Introduction to VMS System Routines*.

Some main headings in the routine template contain information that requires no further explanation beyond what is given in Table 1–1. However, the following main headings contain information that does require additional discussion, and this discussion takes place in the remaining subsections of this section:

Format heading
Returns heading
Arguments heading
Condition Values Returned heading

Table 1–1 Main Headings in the Routine Template

Main Heading	Description
Routine Name	The routine entry point name appears at the top of the first page. It is usually, though not always, followed by the English text name of the routine.
Routine Overview	The routine overview appears directly below the routine name; the overview explains, usually in one or two sentences, what the routine does.
Format	The format heading follows the routine overview. The format gives the routine entry point name and the routine argument list.
Returns	The returns heading follows the routine format. It explains what information is returned by the routine.

(continued on next page)

Table 1–1 (Cont.) Main Headings in the Routine Template

Main Heading	Description
Arguments	The arguments heading follows the returns heading. Detailed information about each argument is provided under the arguments heading. If a routine takes no arguments, it is indicated by the word "None."
Description	The description heading follows the arguments heading. The description section contains information about specific actions taken by the routine: interaction between routine arguments, if any; operation of the routine within the context of VMS; user privileges needed to call the routine, if any; system resources used by the routine; and user quotas that may affect the operation of the routine. For some simple routines, a description section is not necessary because the routine overview provides the needed information.
Condition Values Returned	Always present. The condition values returned section follows the description section. It lists the condition values (typically status or completion codes) returned by the routine.

1.1.1 Format Heading

The following two types of information may be present under the Format heading:

- Procedure call format
- Explanatory text

All system service routines have a procedure call format. Use of the procedure call format results in a routine call conforming to the procedure call mechanism described in the VAX Procedure Calling and Condition Handling Standard; for example, an entry mask is created, registers are saved, and so on.

Explanatory text may follow the procedure call format. This text is present only when needed to clarify the format(s). For example, the call format indicates that arguments are optional by enclosing them in brackets ([ ]). However, square brackets alone cannot convey all the important information that may apply to optional arguments. For example, in some routines that have many optional arguments, if you select one optional argument, you must select another optional argument. In such cases, text following the format clarifies this fact.

A procedure call format is shown under the Format heading. For example:

ENTRY-POINT-NAME arg1 ,arg2 ,[arg3] ,nullarg ,[arg5] ,[arg6]

The format given here, though intended to be generic, is in fact specific to some extent; it is chosen in order to bring to light some of the syntactical mechanisms used to handle the more complex routine calls.

The sample format exemplifies the use of the following syntax rules.

Element	Syntax Rule
Entry Point Names	Entry point names are always shown in uppercase letters.
Argument Names	Argument names are always shown in lowercase letters.
Spaces	You must leave at least one space between the entry point name and the first argument, and between arguments.
Brackets ([ ])	Brackets enclose optional arguments; **arg3**, **arg5**, and **arg6** are optional arguments because they are enclosed in brackets. See Table 1–2 for a summary of the way that some languages treat optional arguments.
Commas	Between arguments, the comma always follows the space.
Null Arguments	A null argument is a placeholding argument. It is used to hold a place in the argument list for an argument Digital has not yet implemented. A null argument is always given the name "null_arg." When calling a routine that has a null argument, you must either (1) supply the value 0 for the null argument or (2) supply no value but include the comma in the call format to mark its place.

Languages like VAX MACRO and BLISS allow you to omit trailing arguments and their placeholders. Optional arguments are handled differently by other languages. If you are coding in VAX BASIC, VAX C, VAX COBOL, VAX FORTRAN, or VAX Pascal, you cannot omit trailing arguments; the compilers for these languages expect all arguments to be present.

For example, if you code a system service call from a C program and replace the **arg5** and **arg6** arguments with commas, the program will not compile; if you omit the arguments and the commas, the program compiles but you get a run-time error. The C language requires a zero in place of omitted arguments.

Table 1–2 describes how different languages expect optional arguments to be treated.

Table 1–2 Language Conventions for Optional Arguments

Language	How to Omit Optional Arguments
BASIC	Replace the argument with a comma
BLISS–32	Omit the keyword or replace the argument with 0
C	Replace the argument with 0
COBOL	Replace the argument with the keyword OMITTED
FORTRAN	Replace the argument with a comma
MACRO	Omit the keyword or replace the argument with 0
Pascal	Replace the argument with a comma
PL/1	Replace the argument with a comma

See your language reference manual for a more detailed explanation of how to handle placeholders for optional arguments.

1.1.2 Returns Heading

The information under the Returns heading describes what information, if any, the routine returns to the caller. For system services, the information returned is always a longword condition value.

This condition value contains various kinds of information, but most importantly for the caller, it describes (in bits 0 through 3) the completion status of the operation. Programmers test the condition value to determine if the routine completed successfully.

Status information is returned by means of a condition value in a VAX register (R0). This is of little importance to high-level language programmers because the high-level language programmer receives this status information in the return (or status) variable he or she uses when making the call. The run-time environment established for the high-level language program allows the status information in R0 to be moved automatically to the user's return variable.

The Condition Values Returned heading in the routine template describes the possible condition values that the routine can return.

1.1.3 Arguments Heading

Detailed information about each argument listed in the call format is shown under the Arguments heading. Arguments are described in the order in which they appear in the call format. If the routine has no arguments, it is indicated by the word *None*.

The following format is used to describe each argument.

argument-name

```
VMS Usage:    argument-VMS-data-type
type:         argument-data-type
access:       argument-access
mechanism:    argument-passing-mechanism
```

This is followed by one or more paragraphs of structured text describing how to use the argument. The argument descriptions are followed by descriptions of function codes and item codes if the **func** argument or the **itmlst** argument is used.

1.1.3.1 VMS Usage Entry

The VMS usage entry indicates the VMS data type of the argument, not the VAX standard data type. Each VMS data type has only one storage representation; for example, the VMS data type **access_mode** is an unsigned byte. In addition, a VMS data type might or might not have a conceptual meaning.

Most VMS data types can be considered conceptual types; that is, they carry meaning unique to the context of the VMS operating system. For example, the storage representation of the VMS data type **access mode** is an unsigned byte, and the conceptual basis for this unsigned byte is that it designates a hardware access mode and therefore has only four valid values: 0, designating kernel mode; 1, executive mode; 2, supervisor mode; and 3, user mode. However, some VMS data types are not conceptual types; that is, they specify a storage representation but carry no other semantic content from the point of view of VMS. For example, the VMS data type **byte_signed** is not a conceptual type.

The *Introduction to VMS System Routines* describes the VMS data types in more detail. It also contains language implementation charts, which describe how to construct each of the VMS data types in a number of high-level languages.

1.1.3.2 Type Entry

When a calling program passes an argument to a system service, the service expects the argument to be of a particular data type. The service descriptions in the *VMS System Services Reference Manual* indicate the expected data types for each argument.

Properly speaking, an argument does not have a data type; rather, the data specified by an argument has a data type. The argument is merely the vehicle for the passing of data to the called routine.

As described in the VAX Procedure Calling Standard in the *Introduction to VMS System Routines*, procedure calls result in the construction of an **argument list**. This argument list is a vector of longwords. The first longword in the list contains a count of the number of remaining longwords, and each remaining longword is one argument. Thus, an **argument** is one longword in the argument list.

Nevertheless, the phrase "argument data type" is frequently used to describe the data type of the data specified by the argument. This terminology is used because it is simpler and more straightforward than the strictly accurate phrase "data type of the data specified by the argument."

The *Introduction to VMS System Routines* describes the data types allowed by the VAX Procedure Calling Standard.

1.1.3.3 Access Entry

The argument-access entry describes the way in which the called routine accesses the data specified by the argument. The following three methods of access are the most common:

- Read only. Data upon which a routine operates, or data the routine needs to perform its operation, must be **read** by the called routine. Such data is also called **input** data. When an argument specifies input data, the "access" entry shows "read only."

  The term "only" indicates that the called routine does not both read and write (that is, modify) the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.

- Write only. Data that the called routine returns to the calling routine must be **written** into a location where the calling routine can access it. Such data is also called **output** data. When an argument specifies output data, the "access" entry shows "write only."

  The term "only" is present to indicate that the called routine does not read the contents of the location either before or after it writes into the location.

- Modify. When an argument specifies data that is both read and written by the called routine, the "access" entry shows "modify." In this case, the called routine reads the input data, which it uses in its operation, and then overwrites the input data with the results (the output data) of the operation. Thus, when the called routine completes execution, the input data the argument specifies is lost.

Following is a complete list of the access types allowed by the VAX Procedure Calling Standard:

- Read only
- Write only
- Modify

- Function call (before return)
- JMP after unwind
- Call after stack unwind
- Call without stack unwind

1.1.3.4 Mechanism Entry

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the argument-passing mechanism. There are three types of argument-passing mechanisms:

- By value. When the longword argument in the argument list contains the actual data to be used by the routine, the actual data is said to be passed to the routine by value. In this case, the longword argument contains the actual data; in other words, the argument is the actual data. Note that because an argument is only one longword in length, only data that can be represented in one longword can be passed by value.
- By reference. When the longword argument in the argument list contains the address of the data to be used by the routine, the data is said to be passed by reference. In this case, the argument is a pointer to the data.
- By descriptor. When the longword argument in the argument list contains the address of a descriptor, the data is said to be passed by descriptor. A descriptor consists of two or more longwords (depending on the type of descriptor used), which describe the location, length, and data type of the data to be used by the called routine. In this case, the argument is a pointer to a descriptor that itself is a pointer to the actual data.

  The only descriptor class that system services accept is class S.

The following table contains the passing mechanisms allowed by the VAX Procedure Calling Standard and the system services.

Passing Mechanism	Descriptor Code
By value	
By reference	
By reference, array reference	
By descriptor, fixed-length	DSC$K_CLASS_S

1.1.3.5 Explanatory Text Entry

For each argument, one or more paragraphs of explanatory text follow the type, access, and mechanism entries. The first paragraph is highly structured and always contains the following items of information:

- An initial sentence fragment that describes (1) the nature of the data specified by the argument and (2) the way in which the routine uses this data. For example, if an argument were supplying a number, which the routine was to convert to another data type, the initial sentence fragment would be something like the following: "number that is to be converted to the such-and-such data type."

- A sentence expressing the relationship between the argument and the data it specifies. This relationship is the passing mechanism used to pass the data.

  If the passing mechanism is by value, this sentence says something like the following: "the xxx argument contains (or is) the such-and-such data."

  If the passing mechanism is by reference, this sentence says something like the following: "the xxx argument is the address of the such-and-such data."

  If the passing mechanism is by descriptor, this sentence says something like the following: "the xxx argument is the address of a descriptor pointing to the such-and-such data."

Additional explanatory paragraphs follow each argument, as needed. For example, some arguments specify complex data consisting of many discrete fields, each of which has a particular purpose and use. In such cases, additional paragraphs provide detailed descriptions of each such field, symbolic names for the fields, if any, and guidance relating to their use.

1.1.4 Condition Values Returned Heading

A condition value is an unsigned longword that has the following uses in the VAX architecture:

- It indicates the success or failure of a called procedure.
- It describes an exception condition when an exception is signaled.
- It identifies system messages.
- It reports program success or failure to the command language level.

See the illustration in the *Introduction to VMS System Routines* that depicts the format and contents of the longword condition value. The *Introduction to VMS System Routines* also describes these contents and explains in detail the uses of the condition value.

Under the Condition Values Returned heading, a two-column list gives the symbolic code for each condition value that the routine can return and its accompanying description. This description explains whether the condition value indicates success or failure and, if failure, what user action may have caused the failure and what can be done to correct it.

Note that the list of condition values is as complete as possible. However, the complexity of some internal routines occasionally causes certain rare condition codes to be returned. If a condition value is not listed, see the *VMS System Messages and Recovery Procedures Reference Manual*.

Symbolic codes for condition values are system defined. The symbolic code defined for each condition value equates to a number that is identical to the longword condition value when interpreted as a number. In other words, though the condition value consists of several fields, each of which can be interpreted individually for specific information, the entire longword condition value itself can be interpreted as an unsigned longword integer, and this integer has an equivalent symbolic code.

Note that if a called routine generates an exception condition during execution, the exception condition is **signaled**; the exception condition is then **handled** by a condition handler (either user-supplied or system-supplied). Depending on the nature of the exception condition and on the condition handler that handles the exception condition, the called routine either continues normal execution or terminates abnormally.

The Condition Values Returned section describes the condition values returned by the routine when it completes execution without generating an exception condition.

1.1.5 Condition Values Returned in the I/O Status Block Heading

When the called routine returns a condition value in an I/O status block, the possible condition values that the routine can return are listed under the Condition Values Returned in the I/O Status Block heading.

Some system services complete asynchronously; that is, they return to the caller immediately after the call to the service is successfully queued but before the operation to be performed by the service has completed. This lets the calling program continue execution while the system service itself is executing. System services that complete asynchronously all have arguments that specify an I/O status block. When the system service operation has completed, a condition value specifying the completion status of the operation is written to the I/O status block.

The first word in the I/O status block receives the condition value for the final completion status of an asynchronous system service. Representing a longword condition value in a word-length field is possible for system services because the high-order word in system service condition values is 0.

One field in the condition value specifies which facility generated the condition value; this field is in the high-order word of the longword condition value. For the system facility, the value of this field is 0. This fact allows condition values generated by the system facility (which includes all system services) to be represented in a word, rather than a longword, because bits in the high-order word are all zeros.

For an explanation of the contents of the fields in the longword condition value, see the *Introduction to VMS System Routines*.

2

Calling System Services

System service procedures are called using the standard VAX procedure calling conventions. The programming languages that generate VAX native mode instructions provide mechanisms for specifying the procedure calls. These languages and supporting documentation are listed in the Preface.

When you code a system service call, you must supply whatever arguments the service requires.

When the service completes execution, it returns control to the calling program with a return condition value. The caller should analyze the condition value to determine the success or failure of the service call so that the program can alter the flow of execution, if necessary.

If you are a VAX MACRO programmer, you should read Section 2.4 for details on how to write the instructions that generate system service calls.

If you program in either VAX MACRO or a high-level language, you should read Section 2.2, Section 2.7, and Section 2.9. Section 2.2 provides information about specifying arguments to system services. Section 2.7 discusses methods for checking return status from system services. Section 2.9 provides programming examples in a number of VAX native languages to aid high-level language programmers in interpreting the programming examples that appear throughout Chapters 3 through 15.

If you program in a high-level language, you should read Section 2.8 for information about how to call system services from high-level languages. For detailed information and examples, see the user's guide for your programming language.

System service macros generate argument lists and CALL instructions to call system services. These macros are located in the system library SYS$LIBRARY:STARLET.MLB. When you assemble a source program, this library is searched automatically for unresolved references.

Knowledge of VAX MACRO rules for assembly language programming is required for understanding the material presented in this section. The *VAX MACRO and Instruction Set Reference Manual* contains the necessary prerequisite information.

2.1 System Services and System Integrity

Many system services are available and suitable for application programs, but the use of some services must be restricted to protect the performance of the system and the integrity of user processes.

For example, because the creation of permanent mailboxes uses system dynamic memory, the unrestricted use of permanent mailboxes could decrease the amount of memory available to other users. Therefore, the ability to create permanent mailboxes is controlled: a user must be specifically assigned the privilege to use the Create Mailbox ($CREMBX) system service to create a permanent mailbox.

The various controls and restrictions applied to system service usage are described in this chapter. The Description section of each system service in the *VMS System Services Reference Manual* lists any privileges and quotas necessary to use the service.

2.1.1 User Privileges

The system manager, who maintains the user authorization file for the system, grants privileges to use protected system services. The user authorization file contains, in addition to profile information about each user, a list of specific user privileges and resource quotas.

When you log in to the system, the privileges and quotas assigned to you are associated with the process created on your behalf. These privileges and quotas are applied to every image the process executes.

When an image issues a call to a system service that is protected by privilege, the privilege list is checked. If you have the specific privilege required, the image is allowed to execute the system service; otherwise, a condition value indicating an error is returned.

For a list of privileges, see the description of the Create Process ($CREPRC) system service in the *VMS System Services Reference Manual*.

2.1.2 Resource Quotas

Many system services require certain system resources for execution. These resources include system dynamic memory and process quotas for I/O operations. When a system service that uses a resource controlled by a quota is called, the process's quota for that resource is checked. If the process has exceeded its quota, or if it has no quota allotment, an error condition value may be returned.

2.1.3 Access Modes

A process can execute at any one of four access modes: user, supervisor, executive, or kernel. The access modes determine a process's ability to access pages of virtual memory. Each page has a protection code associated with it, specifying the type of access—read, write, or no access—allowed for each mode. The *VAX Architecture Handbook* provides additional information about access modes.

For the most part, user-written programs execute in user mode; system programs executing at the user's request (system services, for example) may execute at one of the other three, more privileged, access modes.

In some system service calls, the access mode of the caller is checked. For example, when a process tries to cancel timer requests, it can cancel only those requests that were issued from the same or less privileged access modes. For example, a process executing in user mode cannot cancel a timer request made from supervisor, executive, or kernel mode.

Note that many system services use access modes to protect system resources, and thus employ a special convention for interpreting access mode arguments. You can specify an access mode using a numeric value or a symbolic name. The following table shows the access modes and their numeric values, symbolic names, and privilege ranks.

Access Mode	Numeric Value	Symbolic Name	Privilege Rank
Kernel	0	PSL$C_KERNEL	High
Executive	1	PSL$C_EXEC	
Supervisor	2	PSL$C_SUPER	
User	3	PSL$C_USER	Low

The symbolic names are defined by the symbolic definition macro $PSLDEF.

System services that permit an access mode argument allow callers to specify only an access mode less privileged than, or equal in privilege to, the access mode from which the service was called. If the access mode specified is more privileged than the access mode from which the service was called, the less privileged access mode is always used.

To determine the mode to use, VMS compares the specified access mode with the access mode from which the service was called. Because this operation results in an access mode with a higher numeric value (when the access mode of the caller is different from the specified access mode), the access mode is said to be **maximized.**

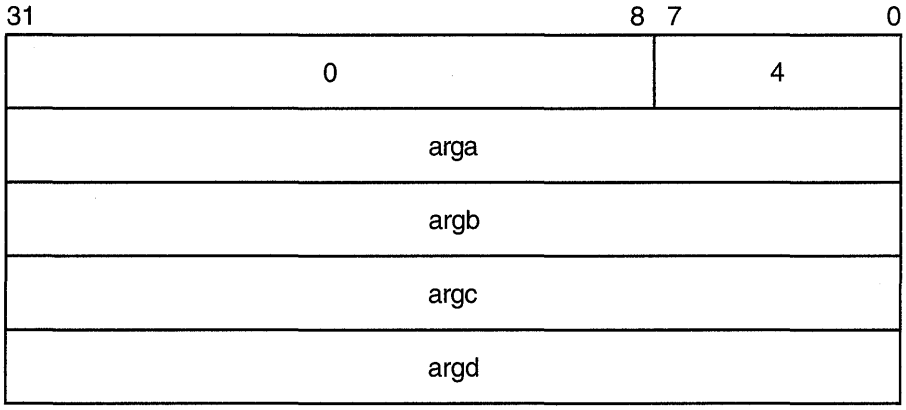
Because much of the code you write executes in user mode, you can omit the access mode argument. The argument value defaults to 0 (kernel mode), and when this value is compared with the value of the current execution mode (3, user mode), the higher value (3) is used.

2.2 Determining Arguments for System Services

You can determine the arguments required by a system service from each service's description in the *VMS System Services Reference Manual.* The Format section in each system service description indicates the positional dependencies and keyword names of each argument, as shown in the following sample:

```
$SERVICE arga ,argb ,argc ,argd
```

This format indicates that the macro name of the service is $SERVICE and that it requires four arguments, ordered as shown and with keyword names **arga**, **argb**, **argc**, and **argd**. You must use the following format for the argument list for this service.



ZK–0854–GE

All arguments are longwords. The first longword in the list must always contain, in its low-order byte, the number of arguments in the remainder of the list. The remaining three bytes must be zeros.

Many arguments to system services are optional; these are indicated by square brackets in the macro formats. For example, if the second and third arguments of $SERVICE are optional, the macro format looks like the following:

```
$SERVICE arga ,[argb] ,[argc] ,argd
```

If you omit an optional argument in a system service macro, the macro supplies a default value for the argument.

Arguments that are optional to system services always have default values, whether they are passed by value, by reference, or by descriptor. In almost every case, an optional argument defaults to 0. The macros used to call the system services allow some languages to set default values to values other than 0 (VAX MACRO and VAX BLISS–32 allow this). The descriptions of the optional arguments in the *VMS System Services Reference Manual* specify default values other than 0.

The description of an optional argument always specifies what action the service takes when the default value is used.

Arguments that specify a return address may be optional when the system service returns information; if the program does not require the information, you can omit the optional argument.

2.3 Obtaining Values for Symbolic Codes

Individual services have symbolic codes for special return conditions, argument list offsets, identifiers, and flags associated with these services. For example, the Create Process ($CREPRC) service (which is used to create a subprocess or a detached process) has symbolic codes associated with the various privileges and quotas you can grant to the created process.

The default system macro library, STARLET.MLB, contains the macro definitions for most system symbols. When you assemble a source program that calls any of these macros, the assembler automatically searches STARLET.MLB for the macro definitions. Each symbol name has a numeric value.

If your language has a method of obtaining values for these symbols, this method is explained in the user's guide.

If your language does not have such a method, you can do the following:

1. Write a short VAX MACRO program containing the desired macro(s).
2. Assemble the program and generate a listing. Using the listing, find the desired symbols and their hexadecimal values.
3. Define each symbol with its value within your source program.

For example, to use the Get Job/Process Information ($GETJPI) service to find out the accumulated CPU time (in 10-millisecond ticks) for a specified process, you must obtain the value associated with the item identifier JPI$_CPUTIM. You can do this in the following way:

1. Create the following three-line VAX MACRO program (named JPIDEF.MAR here; you may choose any name you want).

```
.TITLE  JPIDEF  Obtain values for $JPIDEF
$JPIDEF GLOBAL          ; These MUST be UPPERCASE
.END
```

2. Assemble and link the program to create the file JPIDEF.MAP.

```
$ MACRO JPIDEF
$ LINK/NOEXE/MAP/FULL  JPIDEF
%LINK-W-USRTFR, image NL:[].EXE; has no user transfer address
```

   The file JPIDEF.MAP contains the symbols defined by $JPIDEF listed both alphabetically and numerically.

3. Find the value of JPI$_CPUTIM and define the symbol in your program.

2.4 Calling System Services from VAX MACRO

System service macros generate argument lists and CALL instructions to call system services. These macros are located in the system library SYS$LIBRARY:STARLET.MLB. When you assemble a source program, this library is searched automatically for unresolved references.

Knowledge of VAX MACRO rules for assembly language programming is required for understanding the material presented in this section. The *VAX MACRO and Instruction Set Reference Manual* contains the necessary prerequisite information.

Each system service has four macros associated with it. These macros allow you to define symbolic names for argument offsets, construct argument lists for system services, and call system services. The following table lists the generic macros and the functions they serve.

Macro	Function
$nameDEF	Defines symbolic names for the argument list offsets
$name	Defines symbolic names for the argument list offsets and constructs the argument list
$name_S	Calls the system service and constructs the argument list
$name_G	Calls the system service and uses the argument list constructed by $name macro

2.4.1 Using Macros to Construct Argument Lists

There are two generic macros for constructing argument lists for system services:

$name
$name_S

The macro you use depends on which macro you are going to use to call the system service. If you use the $name_G macro to call a system service, you should use the $name macro to construct the argument list. If you use the $name_S macro to call a system service, you can also use it to construct the argument list.

2.4.1.1 Specifying Arguments with the $name_S Macro and the $name Macro

When you use the $name_S or the $name macro to construct an argument list for a system service, you can specify arguments in any of three ways:

- By using keywords to describe the arguments. All keywords must be followed by an equal sign (=) and then by the value of the argument.
- By using positional order, with omitted arguments indicated by commas in the argument positions. You can omit commas for optional trailing arguments.
- By using both positional dependence and keyword names (positional arguments must be listed first).

For example, $SERVICE may have the following format:

$SERVICE arga ,[argb] ,[argc] ,argd

Assume, for the purposes of this example, that **arga** and **argb** are arguments that require you to specify numeric values and that **argc** and **argd** require you to specify addresses.

The following two examples show valid ways of writing the $name_S macro to call $SERVICE.

$name_S Example 1: Using Keywords

```
MYARGD: .LONG   100
          .
          .
          .
        $SERVICE_S ARGB=#0,ARGC=0,ARGA=#1,ARGD=MYARGD
```

$name_S Example 2: Specifying Arguments in Positional Order

```
MYARGD: .LONG   100
          .
          .
          .
        $SERVICE_S #1,,,MYARGD
```

The argument list is pushed on the stack, as follows.

```
PUSHAL      MYARGD
PUSHL       #0
PUSHL       #0
PUSHL       #1
```

Note that all arguments, whether specified positionally or with keywords, must be valid assembler expressions because they are used as source operands in instructions.

The following two examples show valid ways of writing a $name macro to construct an argument list for a later call to $SERVICE.

$name Example 1: Using Keywords

```
LIST:   $SERVICE -
                ARGB=0, -
                ARGC=0, -
                ARGA=1, -
                ARGD=MYARGD
```

$name Example 2: Specifying Arguments in Positional Order

```
LIST:   $SERVICE -
                1,,,MYARGD
```

The argument list generated in both cases is as follows.

```
LIST:   .LONG   4
        .LONG   1
        .LONG   0
        .LONG   0
        .ADDRESS -
                MYARGD
```

Note that all arguments, whether specified in positional order or by keyword, must be expressions that the assembler can evaluate to generate .LONG or .ADDRESS data directives. Contrast this with the arguments for the $name_S macro, which must be valid assembler expressions because they are used as source operands in instructions.

2.4.1.2 Conventions for Specifying Arguments to System Services

You must specify the arguments according to the VAX MACRO assembler rules for specifying and addressing operands.

The way to specify a particular argument depends on the following factors:

- Whether the system service requires an address or a value as the argument. In the *VMS System Services Reference Manual*, the descriptions of the arguments following a system service macro format always indicate if the argument is an address. A Boolean value, number, or mask takes a value as the argument.
- The system service macro being used. The expansions of the $name and $name_S macros in the examples in the preceding section showed the code generated by each macro.

If you are unsure whether you specified a value or an address argument correctly, you can assemble the program with the .LIST MEB directive to check the macro expansion. See the *VAX MACRO and Instruction Set Reference Manual* for details.

2.4.1.3 Defining Symbolic Names for Argument List Offsets: $name and $nameDEF

You can refer symbolically to arguments in the argument list. Each argument in an argument list has an offset from the beginning of the list; a symbolic name is defined for the numeric offset of each argument. If you use the symbolic names to refer to the arguments in a list, you do not have to remember the numeric offset (which is based on the position of the argument shown in the macro format).

There are two additional advantages to referring to arguments by their symbolic names:

- Your program is easier to read.
- If an argument list for a system service changes with a later release of a system, the symbols remain the same.

You form the offset names for all system service argument lists by concatenating the service macro name with $_ and the keyword name of the argument. In the following example, *name* is the name for the system service macro and *keyword* is the keyword argument:

```
name$_keyword
```

Similarly, you can define a symbolic name for the number of arguments a particular macro requires, as follows:

name$_NARGS

You can define symbolic names for argument list offsets automatically whenever you use the $name macro for a particular system service. You can also define symbolic names for system service argument lists using the $nameDEF macro. This macro does not generate any executable code; it merely defines the symbolic names so they can be used later in the program. For example:

```
$QIODEF
```

This macro defines the symbol QIO$_NARGS and the symbolic names for the $QIO argument list offsets.

You may need to use the $nameDEF macro if you specify an argument list to a system service without using the $name macro, or if a program refers to an argument list in a separately assembled module.

For example, the $READEF and $READEFDEF macros define the values listed in the following table.

Symbolic Name	Meaning
READEF$_NARGS	Number of arguments in the list (2)
READEF$_EFN	Offset of EFN argument (4)
READEF$_STATE	Offset of STATE argument (8)

Thus, you can specify the $READEF macro to build an argument list for a $READEF system service call, as follows:

```
READLST:   $READEF  EFN=1,STATE=TEST1
```

Later, the program may want to use a different value for the **state** argument to call the service. The following lines show how you can do this with a call to the $name_G macro.

```
MOVAL   TEST2,READLST+READEF$_STATE
$READEF_G READLST
```

The MOVAL instruction replaces the address TEST1 in the $READEF argument list with the address TEST2; the $READEF_G macro calls the system service with the modified list.

2.4.2 Using Macros to Call System Services

There are two generic macros for writing calls to system services:

$name_S
$name_G

Which macro you use depends on how the argument list for the system service is constructed.

- The $name_S macro requires you to supply the arguments to the system service in the system service macro. The macro generates code to push the argument list onto the call stack during program execution. With this macro, you can use registers to contain or point to arguments so that you can write reentrant programs.

- The $name_G macro requires you to construct an argument list elsewhere in the program and specify the address of this list as an argument to the system service. (A macro is provided to create an argument list for each system service.) With this macro, you can use the same argument list, with modifications if necessary, for more than one invocation of the macro.

The $name_S macro generates a CALLS instruction; the $name_G macro generates a CALLG instruction. The services are called according to the standard procedure calling conventions. System services save all registers except R0 and R1, and restore the saved registers before returning control to the caller.

The following sections describe how to code system service calls using each of these macros.

2.4.2.1 The $name_S Macro

The $name_S macro call has the following format:

$name_S arg1, ..., argn

The macro generates code to push the arguments on the stack in reverse order. The actual instructions used to place the arguments on the stack are determined as follows:

- If the system service requires a value for an argument, either a PUSHL instruction or a MOVZWL to –(SP) instruction is generated.
- If the system service requires an address for an argument, a PUSHAB, PUSHAW, PUSHAL, or PUSHAQ instruction is generated, depending on the context.

The macro then generates a call to the system service in the following format:

CALLS #n,@#SYS$name

In this format, *n* is the number of arguments on the stack.

2.4.2.2 Example of $name_S Macro Call

Because a $name_S macro constructs the argument list at execution time, you can supply addresses and values using register addressing modes. The following line can be used to execute the $READEF_S macro:

```
$READEF_S EFN=#1,STATE=(R10)
```

R10 contains the address of the longword to receive the status of the flags.

This macro instruction is expanded as follows.

```
PUSHAL  (R10)
PUSHL   #1
CALLS   #2,@#SYS$READEF
```

2.4.2.3 The $name_G Macro

The $name_G macro requires a single operand:

$name_G label

label
Address of the argument list.

The $name Macro

Macros are provided to create argument lists for the $name_G macro. The format of the macros is as follows:

label: $name arg1,...,argn

label
Symbolic address of the generated argument list. This is the label given as an argument in the $name_G macro.

$name
The service macro name.

arg1,...,argn
Arguments to be placed in successive longwords in the argument list.

The $name_G macro (used with the $name macro) is especially useful for doing the following:

- Making calls to system services that have long argument lists
- Calling services repeatedly during the execution of a single program with the same, or essentially the same, argument list

2.4.2.4 Example of $NAME and $name_G Macro Calls

The example that follows shows how you can write a call to the Read Event Flags ($READEF) system service using an argument list created by $name.

The $READEF system service has the following macro format:

$READEF efn ,state

The **efn** argument must specify the number of an event flag cluster, and the **state** argument must supply the address of a longword to receive the contents of the cluster.

You may specify these arguments using the $name macro, as follows.

```
READLST:
        $READEF EFN=1, -          ; Argument list for $READEF
                STATE=TESTFLAG
```

This $READEF macro generates the following code.

```
READLST:
        .LONG   2                 ; Argument list for $READEF
        .ADDRESS 1
        .ADDRESS -
                TESTFLAG
```

Executing the $READEF macro requires only the following line:

```
$READEF_G READLST
```

The macro generates the following code to call the Read Event Flags system service:

```
CALLG  READLST,@#SYS$READEF
```

SYS$READEF is the name of a vector to the entry point of the Read Event Flags system service. The linker automatically resolves the entry point addresses for all system services.

2.5 System Service Completion

When a system service completes, control is returned to your program. You can specify how and when control is returned to your program by choosing synchronous or asynchronous forms of system services and by enabling process execution modes.

The following sections describe:

- When synchronous system services return control to your program
- When asynchronous system services return control to your program
- How you can synchronize the completion of asynchronous system services
- How control is returned to your program when special process execution modes are enabled

2.5.1 Synchronous and Asynchronous System Services

You can execute a number of system services either synchronously or asynchronously (for example, SYS$GETJPI and SYS$GETJPIW). The *W* at the end of the system service name indicates the synchronous version of the system service.

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

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

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

The following is an example of using the $SYNCH system service to check the completion status of the asynchronous service $GETJPI.

Example of $SYNCH System Service in VAX FORTRAN

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

STATUS = SYS$GETJPI (%VAL(FLAG),
2                    PID_VALUE,
2                    ,
2                    NAME_BUF_LEN,
2                    JPISTATUS,
2                    ,)
IF (.NOT. STATUS) CALL LIB$SIGNAL (%VAL(STATUS))
.
.
.
STATUS = SYS$SYNCH (%VAL(FLAG),
2                   JPISTATUS)
IF (.NOT. JPISTATUS) THEN
  CALL LIB$SIGNAL (%VAL(JPISTATUS))
END IF

END
```

2.5.2 Process Execution Modes

When an error occurs during the execution of a system service, two process execution modes affect how control is returned to the calling program:

- Resource wait mode
- System service failure exception mode

If you change the default setting in a program for either of these modes, the program must handle the special return conditions that result. The next two sections discuss considerations for using these modes.

2.5.2.1 Resource Wait Mode

Normally, when a system service is called and a required resource is not available, the process is placed in a wait state until the resource becomes available. Then, the service completes execution. This mode is called **resource wait mode.**

In a real-time environment, however, it may not be practical or desirable for a program to wait. In these cases, you can choose to disable resource wait mode so that, when a required resource is unavailable, control returns immediately to the calling program with an error condition value. You can disable (and reenable) resource wait mode with the Set Resource Wait Mode ($SETRWM) system service.

How a program responds to the unavailability of a resource depends very much on the application and the particular service being called. In some instances, the program may be able to continue execution and retry the service call later. In other instances, it may be necessary only to note that the program is being required to wait.

2.6 Condition Values Returned from System Services

When a system service finishes execution, a numeric status value is always returned. For VAX MACRO calls, the status value is returned in general register R0; however, the mechanisms used in high-level languages vary. See the appropriate user's guide.

Depending on your specific needs, you can test just the low-order bit, the low-order three bits, or the entire value, as follows:

- The low-order bit indicates successful (1) or unsuccessful (0) completion of the service.
- The low-order three bits, taken together, represent the severity of the error. The severity code values are as follows.

Value	Meaning	Symbolic Name
0	Warning	STS$K_WARNING
1	Success	STS$K_SUCCESS
2	Error	STS$K_ERROR
3	Informational	STS$K_INFO
4	Severe or fatal error	STS$K_SEVERR
5–7	Reserved	

  The symbolic definition macro $STSDEF defines the symbolic names.
- The remaining bits (bits 3 through 31) classify the particular return condition and the operating system component that issued the condition value. For system service return status values, the high-order word (bits 16 through 31) contains zeros.

Each numeric condition value has a unique symbolic name in the following format:

SS$_code

where *code* is a mnemonic describing the return condition.

For example, the following usually indicates a successful return:

```
SS$_NORMAL
```

An example of an error return condition value is as follows:

```
SS$_ACCVIO
```

This condition value indicates that an access violation occurred because a service could not read an input field or write an output field.

The symbolic definitions for condition values are included in the default system library SYS$LIBRARY:STARLET.OLB. You can obtain a listing of these symbolic codes at assembly time by invoking the system macro $SSDEF. To check return conditions, use the symbolic names for system condition values.

VMS does not automatically handle system service failure or warning conditions; you must test for them and handle them yourself. This contrasts with the operating system's handling of exception conditions detected by the hardware or software; the system handles these exceptions by default, although you can intervene in or override the default handling by declaring a condition handler (see Chapter 11).

2.6.1 Information Provided by Condition Values

Condition values returned by system services may provide information; that is, they do not indicate only whether the service completed successfully. The usual condition value indicating success is SS$_NORMAL, but others are defined. For example, the condition value SS$_BUFFEROVF, which is returned when a character string returned by a service is longer than the buffer provided to receive it, is a success code. This condition value, however, gives the program additional information.

Warning returns and some error returns indicate that the service may have performed some, but not all, of the requested function.

The possible condition values that each service can return are described with the individual service descriptions in the *VMS System Services Reference Manual*. When you write calls to system services, read the descriptions of the return condition values to determine whether you want the program to check for particular return conditions.

2.7 Testing Return Condition Values

To check for successful completion after a system service call, the program can test the low-order bit of R0 and branch to an error checking routine if this bit is not set, as follows:

```
BLBC    R0,errlabel             ; Error if low bit clear
```

Programs should not test for success by comparing the return status to SS$_NORMAL. A future release of VMS may add new alternate success codes to an existing service, causing programs that test for SS$_NORMAL to fail.

The error checking routine may check for specific values or for specific severity levels. For example, the following instruction checks for an illegal event flag number error condition:

```
CMPL    #SS$_ILLEFC,R0          ; Is event flag number illegal?
```

Note that return condition values are always longword values; however, all system services always return the same value in the high-order word of all condition values returned in R0.

2.7.1 System Messages Generated by Condition Values

When you execute a program with the DCL command RUN, the command interpreter uses the contents of R0 to issue a descriptive message if the program completes with a nonsuccessful status.

The following code fragment shows a simple error-checking procedure in a main program.

```
        $READEF_S -
                EFN=#64, -
                STATE=TEST
        BSBW    ERROR
          .
          .
          .
ERROR:  BLBC    R0,10$          ; Check register 0
        RSB                     ; Success, return
10$:    RET                     ; Exit with R0 status
```

After a system service call, the BSBW instruction branches to the subroutine ERROR. The subroutine checks the low-order bit in register 0 and, if the bit is clear, branches to a RET instruction that causes the program to exit with the status of R0 preserved. Otherwise, the subroutine issues an RSB instruction to return to the main program.

If the event flag cluster requested in this call to $READEF is not currently available to the process, the program exits and the command interpreter displays the following message:

```
%SYSTEM-F-UNASEFC, unassociated event flag cluster
```

The keyword UNASEFC in the message corresponds to the condition value SS$_UNASEFC.

The following three severe errors generated by the calls, not the services, can be returned from calls to system services.

Error	Meaning
SS$_ACCVIO	The argument list cannot be read by the caller (using the $name_G macro), and the service is not called. This meaning of SS$_ACCVIO is different from its meaning for individual services. When SS$_ACCVIO is returned from individual services, the service is called, but one or more arguments to the service cannot be read or written by the caller.
SS$_INSFARG	Not enough arguments were supplied to the service.
SS$_ILLSER	An illegal system service was called.

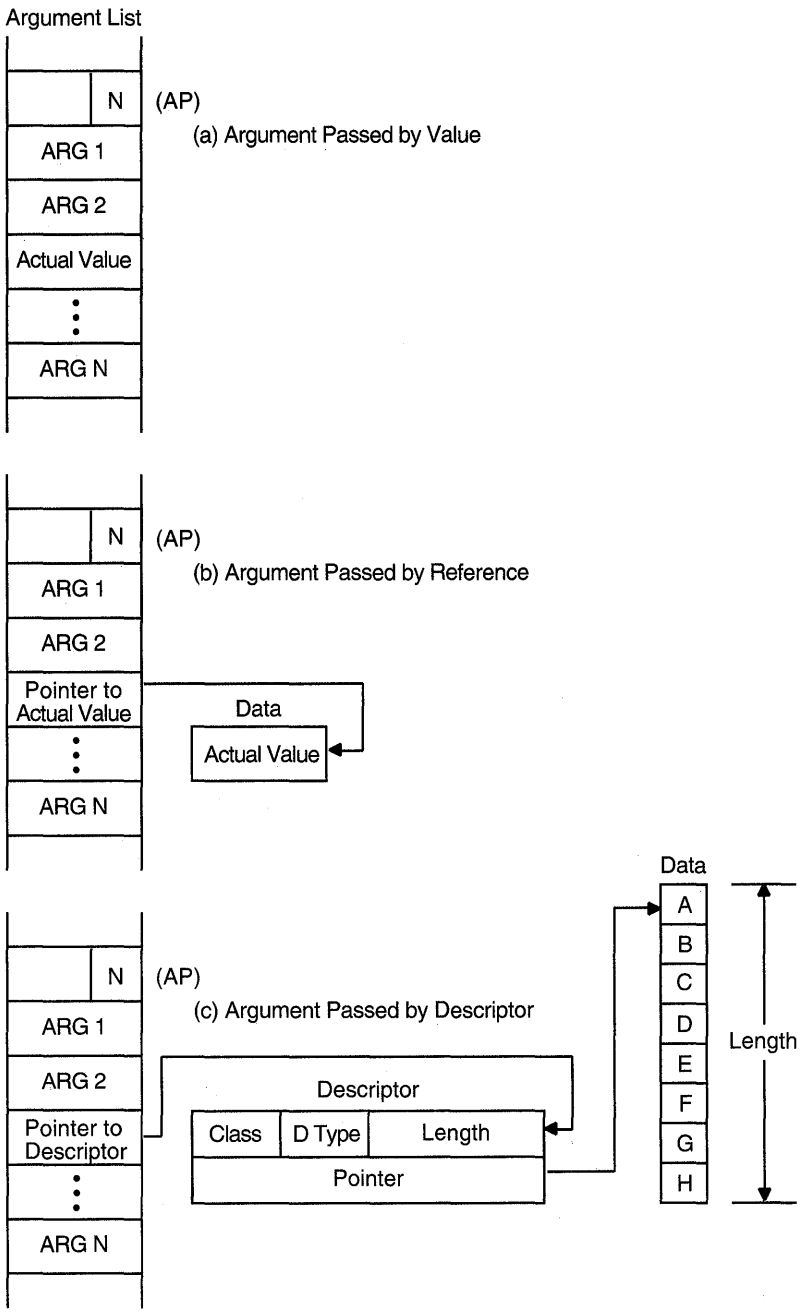
2.8 High-Level Language Calls

Each high-level language supported by VMS provides some mechanism for calling an external procedure and for passing arguments to that procedure. The specifics of the mechanism and the terminology used, however, vary from one language to another. This manual does not describe the ways in which each high-level language calls system services. For specific information, Digital recommends that you refer to the appropriate high-level language user's guide.

VMS system services are external procedures that accept arguments. There are three ways to pass arguments to system services: by value, by reference, and by descriptor. For more information, see Section 1.1.3.4.

The *VMS System Services Reference Manual* provides a description of each service that indicates how each argument is to be passed. Phrases such as "an address" and "address of a character string descriptor" identify reference and descriptor arguments, respectively. Words like "Boolean value," "number," "value," or "mask" indicate an argument passed by value. Figure 2–1 shows how arguments are passed to the system services.

Figure 2–1 Procedure Argument Passing Mechanisms



Note: ARG 1, ARG 2, and ARG N can be passed by value, by reference, or by descriptor in any of these examples.

:(AP) = Argument Pointer

N = Number of Arguments

ZK–1962–GE

Some services also require service-specific data structures that indicate functions to be performed or hold information to be returned. The *VMS System Services Reference Manual* includes descriptions of these service-specific data structures. You can use this information and information from your programming language manuals to define such service-specific item lists.

2.8.1 Testing Return Condition Values in High-Level Languages

When a service returns control to your program, it places a return status value in the general register R0. The value in the low-order word indicates either that the service completed successfully or some specific error prevented the service from performing some or all of its functions. After each call to a system service, you must check whether it completed successfully. You can also test for specific error conditions. (See Section 2.6 for more information about return status values.)

Each language provides some mechanism for testing the return status. Often you need only check the low-order bit, such as by a test for TRUE (success or informational return) or FALSE (error or warning return).

To check the entire value for a specific return condition, each language provides a way for your program to determine the values associated with specific symbolically defined codes. You should always use these symbolic names when you write tests for specific conditions.

For information about how to test for these codes, see the user's guide for your programming language.

2.9 Interpreting the Programming Examples

Chapters 3 through 15 contain programming examples (using VAX MACRO and VAX FORTRAN) designed to familiarize you with the system services and their arguments. The examples do not show complete programming sequences; rather, they show the code or arguments, or both, pertinent to a particular discussion.

Some of the more complex examples contain numeric symbols that correspond to a list of explanatory text.

Although the examples are written using VAX MACRO and VAX FORTRAN, they are designed to be as meaningful as possible to programmers using other high-level languages. Example 2–1 shows a portion of a VAX MACRO program and the equivalent code in the following languages:

VAX Ada
VAX BASIC
VAX BLISS–32
VAX COBOL
VAX FORTRAN
VAX Pascal

Example 2–1 Interpreting MACRO Examples

MACRO Example

```
CYGDES: .ASCID  /CYGNUS/ ❶; Descriptor for CYGNUS string
TBLDES: .ASCID  /LNM$FILE_DEV/❷ ; Logical name table
NAMBUF: .BLKB   255 ❸       ; Output buffer
NAMLEN: .BLKW   1   ❹     ; Word to receive length
ITEMS:  .WORD   255             ; Output buffer length
        .WORD   LNM$STRING      ; Item code
        .ADDRESS -              ; Output buffer
                NAMBUF
        .ADDRESS -              ; Return length
                NAMLEN
        .LONG   0               ; List terminator
          .
          .
          .
        .ENTRY  ORION,0 ❺         ; Routine entry point & mask
❻ $TRNLNM_S -
                TABNAM=TBLDES, -
                LOGNAM=CYGDES, -
                ITMLST=ITEMS
❼ BLBC    R0,ERROR        ; Check for error
          .
          .
          .
       .END
```

MACRO Notes

❶ The input character string descriptor argument is defined using the .ASCID directive.

❷ The name of the table to search is defined using the .ASCID directive.

❸ Enough bytes to hold the output data are allocated for an output character string argument.

❹ The MACRO directive .BLKW reserves a word to hold the output length.

❺ A routine name and entry mask show the beginning of executable code in a routine or subroutine.

❻ A macro name that has the suffix _S or _G calls the service.

You can specify arguments by keyword (as in this example) or by positional order. (Keyword names correspond to the names of the arguments shown in lowercase in the system service format descriptions in the *VMS System Services Reference Manual*.) If you omit any optional arguments (that is, accept the defaults), you can omit them completely if you specify arguments by keyword. If you specify arguments by positional order, however, you must specify the comma for each missing argument.

Use the number sign (#) to indicate a literal value for an argument.

❼ The BLBC instruction causes a branch to a subroutine named ERROR (not shown) if the low bit of the condition value returned from the service is clear (low bit clear = failure or warning). You can use a BSBW instruction to branch unconditionally to a routine that checks the return status.

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

Ada Equivalent

```
with SYSTEM, TEXT_IO, STARLET, CONDITION_HANDLING;  ❶
procedure ORION is
    -- Declare variables to hold equivalence name and length
    --
    EQUIV_NAME: STRING (1..255);  ❷
    pragma VOLATILE (EQUIV_NAME);
    NAME_LENGTH: SYSTEM.UNSIGNED_WORD;
    pragma VOLATILE (NAME_LENGTH);

    -- Declare itemlist and fill in entries.
    --
    ITEM_LIST: STARLET.ITEM_LIST_3_TYPE (1..2) :=  ❸
        (1 =>
            (ITEM_CODE   => STARLET.LNM_STRING,  ❹
             BUF_LEN     => EQUIV_NAME'LENGTH,
             BUF_ADDRESS => EQUIV_NAME'ADDRESS,
             RET_ADDRESS => NAME_LENGTH'ADDRESS),
         2 =>
            (ITEM_CODE   => 0,
             BUF_LEN     => 0,
             BUF_ADDRESS => SYSTEM.ADDRESS_ZERO,
             RET_ADDRESS => SYSTEM.ADDRESS_ZERO));

    STATUS: CONDITION_HANDLING.COND_VALUE_TYPE;  ❺
begin
    -- Translate the logical name
    --
    STARLET.TRNLNM (  ❻
     STATUS => STATUS,
     TABNAM => "LNM$FILE_DEV",
     LOGNAM => "CYGNUS",
     ITMLST => ITEM_LIST);

    -- Display name if success, else signal error
    --
    if not CONDITION_HANDLING.SUCCESS (STATUS) then  ❼
     CONDITION_HANDLING.SIGNAL (STATUS);
    else
        TEXT_IO.PUT ("CYGNUS translates to """);
        TEXT_IO.PUT (EQUIV_NAME (1..INTEGER(NAME_LENGTH)));
        TEXT_IO.PUT_LINE ("""");
    end if;
end ORION;
```

Ada Notes

❶ The **with** clause names the predefined packages of declarations used in this program. SYSTEM and TEXT_IO are standard Ada packages; STARLET defines the VMS system service routines, data types, and constants; and CONDITION_HANDLING defines error handling facilities.

❷ Enough space is allocated to EQUIV_NAME to hold the longest possible logical name. NAME_LENGTH will receive the actual length of the translated logical name. The VOLATILE pragma is required for variables which will be modified by means other than an assignment statement or being an output parameter to a routine call.

❸ ITEM_LIST_3_TYPE is a predeclared type in package STARLET which defines the VMS 3-longword item list structure.

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

❹ The dollar sign character is not valid in Ada identifiers; package STARLET defines the fac$ names by removing the dollar sign.

❺ COND_VALUE_TYPE is a predeclared type in package CONDITION_HANDLING which is used for return status values.

❻ System services are defined in package STARLET using names that omit the prefix SYS$. The passing mechanisms are specified in the routine declaration in STARLET so they need not be specified here.

❼ In this example, any failure status from the $TRNLNM service is signaled as an error. Other means of error recovery are possible; see the VAX Ada documentation for more details.

BASIC Equivalent

```
10  SUB ORION ❶                  ! Subprogram ORION
    OPTION TYPE=EXPLICIT              ! Require declaration of all
                                      ! symbols

    EXTERNAL LONG FUNCTION SYS$TRNLNM  ! Declare the system service
    EXTERNAL WORD CONSTANT LNM$_STRING ! The request code that
                                      ! we will use
    DECLARE WORD NAMLEN, ❷        ! Word to receive length
            LONG SYS_STATUS           ! Longword to receive status
    COMMON (BUF) STRING NAME_STRING = 255 ❸

    RECORD ITEM_LIST                  ! Define item
                                      ! descriptor structure
        WORD BUFFER_LENGTH            ! The buffer length
        WORD ITEM                     ! The request code
        LONG BUFFER_ADDRESS           ! The buffer address
        LONG RETURN_LENGTH_ADDRESS    ! The address of the return len
                                      ! word
        LONG TERMINATOR               ! The terminator
    END RECORD ITEM_LIST              ! End of structure definition

    DECLARE ITEM_LIST ITEMS           ! Declare an item list
    ITEMS::BUFFER_LENGTH = 255%       ! Initialize the item list
    ITEMS::ITEM = LNM$_STRING
    ITEMS::BUFFER_ADDRESS = LOC( NAME_STRING )
    ITEMS::RETURN_LENGTH_ADDRESS = LOC( NAMLEN )
    ITEMS::TERMINATOR = 0
                                  ❹
    SYS_STATUS = SYS$TRNLNM( , 'LNM$FILE_DEV', 'CYGNUS',, ITEMS) ❺

    IF (SYS_STATUS AND 1%) = 0% ❻
    THEN
        ! Error path
    ELSE
        ! Success path
    END IF
    END SUB
```

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

BASIC Notes

❶ The SUB statement defines the routine and its entry mask.

❷ The DECLARE WORD NAMLEN declaration reserves a 16-bit word for the output value.

❸ The COMMON (BUF) STRING NAME_STRING = 255 declaration allocates 255 bytes for the output data in a static area. The compiler builds the descriptor.

❹ The SYS$ form invokes the system service as a function.

Enclose the arguments in parentheses, and specify them in positional order only. Specify a comma for each optional argument that you omit (including trailing arguments).

❺ The input character string is specified directly in the system service call; the compiler builds the descriptor.

❻ The IF statement performs a test on the low-order bit of the return status. This form is recommended for all status returns.

BLISS Equivalent

```
MODULE ORION=

BEGIN
EXTERNAL ROUTINE
    ERROR_PROC: NOVALUE;                ! Error processing routine

LIBRARY 'SYS$LIBRARY:STARLET.L32';      ! Library containing VMS
                                        ! macros (including $TRNLNM).
                                        ! This declaration
                                        ! is required.

GLOBAL ROUTINE ORION: NOVALUE=

    BEGIN
    OWN
        NAMBUF : VECTOR[255, BYTE],      ! Output buffer
        NAMLEN : WORD,                   ! Translated string length
        ITEMS : BLOCK[16,BYTE]
                INITIAL(WORD(255,        ! Output buffer length
                        LNM$_STRING),    ! Item code
                        NAMBUF,          ! Output buffer
                        NAMLEN,          ! Address of word for
                                         ! translated
                                         ! string length
                        0);              ! List terminator

    LOCAL                                ! Return status from
        STATUS;                          ! system service

    STATUS = $TRNLNM(TABNAM = %ASCID'LNM$FILE_DEV',
                     LOGNAME = %ASCID'CYGNUS',
                     ITMLST = ITEMS); ❶

    IF NOT .STATUS THEN ERROR_PROC(.STATUS);  ❷

    END;
```

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

BLISS Notes

❶ The macro is invoked by its service name, without a suffix.

Enclose the arguments in parentheses, and specify them by keyword. (Keyword names correspond to the names of the arguments shown in lowercase in the system service format descriptions in the *VMS System Services Reference Manual*.)

❷ The return status, which is assigned to the variable STATUS, is tested for TRUE or FALSE. FALSE (low bit = 0) indicates failure or warning.

COBOL Equivalent

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ORION.     ❶
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TABNAM PIC X(11) VALUE "LNM$FILE_DEV".
01 CYGDES PIC X(6) VALUE "CYGNUS".
01 NAMDES PIC X(255) VALUE SPACES.  ❷
01 NAMLEN PIC S9(4) COMP.
01 ITMLIS.
   02 BUFLEN PIC S9(4) COMP VALUE 225.
   02 ITMCOD PIC S9(4) COMP VALUE 2.  ❸
   02 BUFADR POINTER VALUE REFERENCE NAMDES.
   02 RETLEN POINTER VALUE REFERENCE NAMLEN.
   02 FILLER PIC S9(5) COMP VALUE 0.
01 RESULT PIC S9(9) COMP.    ❹

PROCEDURE DIVISION.
START-ORION.
    CALL "SYS$TRNLNM"    ❺
      USING OMITTED
            BY DESCRIPTOR TABNAM
            BY DESCRIPTOR CYGDES  ❻
            OMITTED
            BY REFERENCE ITMLIS
      GIVING RESULT.
    IF RESULT IS FAILURE     ❼
       GO TO ERROR-CHECK.
    DISPLAY "NAMDES:  ", NAMDES(1:NAMLEN).
    GO TO THE-END.
ERROR-CHECK.
    DISPLAY "Returned Error: ", RESULT CONVERSION.
THE-END.
    STOP RUN.
```

COBOL Notes

❶ The PROGRAM-ID paragraph identifies the program by specifying the program name, which is the global symbol associated with the entry point. The compiler builds the entry mask.

❷ Enough bytes are allocated for the alphanumeric output data. The compiler generates a descriptor when you specify "USING BY DESCRIPTOR" in the CALL statement.

❸ The value of the symbolic code LNM$STRING is 2. Section 2.3 explains how to obtain values for symbolic codes.

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

❹ This definition reserves a signed longword with COMP (binary) usage to receive the output value.

❺ The service is called by the SYS$ form of the service name, and the name is enclosed in quotation marks.

Specify arguments in positional order only, with USING.... You cannot omit arguments; if you are accepting the default for an argument, you must pass the default value explicitly (OMITTED in this example).

You can specify explicitly how each argument is being passed: by descriptor, by reference (that is, by address), or by value. You can also implicitly specify how an argument is being passed: through the default mechanism (by reference), or through association with the last specified mechanism (thus, the last two arguments in the example are implicitly passed by value).

❻ The input string is defined as alphanumeric (ASCII) data. The compiler generates a descriptor when you specify USING BY DESCRIPTOR in the CALL statement.

❼ The IF statement tests RESULT for a failure status. In this case, control is passed to the routine ERROR-CHECK.

FORTRAN Equivalent

```
SUBROUTINE ORION
IMPLICIT NONE  ! Require declaration of
         ! all symbols
INCLUDE '($SYSSRVNAM)' ! Declare system service names  ❶
INCLUDE '($LNMDEF)' ! Declare $TRNLNM item codes
INCLUDE '(LIB$ROUTINES)'! Declare LIB$ routines

STRUCTURE /ITEM_LIST_3_TYPE/  ! Structure of item list  ❷
  INTEGER*2 BUFLEN  ! Item buffer length
  INTEGER*2 ITMCOD  ! Item code
  INTEGER*4 BUFADR  ! Item buffer address
  INTEGER*4 RETADR  ! Item return length address
END STRUCTURE
RECORD /ITEM_LIST_3_TYPE/ ITEMLIST(2)  ! Declare itemlist

CHARACTER*255 EQUIV_NAME    ! For returned equivalence name
INTEGER*2 NAMLEN     ! For returned name length
VOLATILE EQUIV_NAME,NAMLEN  ❸

INTEGER*4 STATUS     ! For returned service status  ❹

! Fill in itemlist
!
ITEMLIST(1).ITMCOD = LNM$_STRING
ITEMLIST(1).BUFLEN = LEN(EQUIV_NAME)  ❺
ITEMLIST(1).BUFADR = %LOC(EQUIV_NAME)
ITEMLIST(1).RETADR = %LOC(NAMLEN)
ITEMLIST(2).ITMCOD = 0 ! For terminator
ITEMLIST(2).BUFLEN = 0

! Call SYS$TRNLM
!
STATUS = SYS$TRNLNM (,      ! ATTR omitted  ❻
1       'LNM$FILE_DEV',! TABNAM
2       'CYGNUS',     ! LOGNAM
3       ,      ! ACMODE omitted
4       ITEMLIST)     ! ITMLST
```

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

```
! Check return status, display translation if successful
!
IF (.NOT. STATUS) THEN    ❼
    CALL LIB$SIGNAL(%VAL(STATUS))
ELSE
    WRITE (*,*) 'CYGNUS translates to: "',
1       EQUIV_NAME(1:NAMLEN), '"'
END IF
END
```

FORTRAN Notes

❶ The module $SYSSRVNAM in the FORTRAN system default library FORSYSDEF.TLB contains INTEGER and EXTERNAL declarations for each of the system services, so you need not explicitly provide any in your program. Module $LNMDEF defines constants and data structures used when calling the logical name services, and module LIB$ROUTINES contains declarations for the LIB$ Run-Time Library routines.

❷ The structure of a VMS 3-longword item list is declared and then used to define the record variable ITEM_LIST. The second element will be used for the terminator.

❸ The VOLATILE declaration is required for variables which are modified by means other than a direct assignment or as an argument in a routine call.

❹ Return status variables should always be declared as longword integers.

❺ The LEN intrinsic function returns the allocated length of EQUIV_NAME. The %LOC built-in function returns the address of its argument.

❻ By default, FORTRAN passes arguments by reference, except for strings which are passed by CLASS_S descriptor. Arguments are omitted in FORTRAN by leaving the comma as a placeholder. All arguments must be specified or explicitly omitted.

❼ A condition value can be tested for success or failure by a true/false test. For more information on testing return statuses, see the VAX FORTRAN documentation.

Pascal Equivalent

```
[INHERIT('SYS$LIBRARY:STARLET',    ❶
         'SYS$LIBRARY:PASCAL$LIB_ROUTINES')]
PROGRAM ORION (OUTPUT);

TYPE
    Item_List_Cell = RECORD CASE INTEGER OF    ❷
        1:( { Normal Cell }
            Buffer_Length : [WORD] 0..65535;
            Item_Code     : [WORD] 0..65535;
            Buffer_Addr   : UNSIGNED;
            Return_Addr   : UNSIGNED
            );
        2:( { Terminator }
            Terminator    : UNSIGNED
            );
        END;

    Item_List_Template(Count:INTEGER) = ARRAY [1..Count] OF Item_List_Cell;
```

(continued on next page)

Example 2–1 (Cont.) Interpreting MACRO Examples

```
VAR
    Item_List       : Item_List_Template(2);
    Translated_Name : [VOLATILE] VARYING [255] OF CHAR;  ❸
    Status          : INTEGER;

    BEGIN

    { Specify the buffer to return the translation }  ❹
    Item_List[1].Buffer_Length  := SIZE(Translated_Name.Body);
    Item_List[1].Item_Code  := LNM$_String;
    Item_List[1].Buffer_Addr  := IADDRESS(Translated_Name.Body);
    Item_List[1].Return_Addr  := IADDRESS(Translated_Name.Length);

    { Terminate the item list }
    Item_List[2].Terminator  := 0;

    { Translate the CYGNUS logical name }
    Status := $trnlnm(Tabnam := 'LNM$FILE_DEV', Lognam := 'CYGNUS',    ❺
        Itmlst := Item_List);
    IF NOT ODD(Status)  ❻
    THEN
        LIB$SIGNAL(Status)
    ELSE
        WRITELN('CYGNUS is equivalent to ',Translated_Name);

    END.
```

Pascal Notes

❶ The Pascal environment file STARLET.PEN defines VMS system services, data structures and constants. PASCAL$LIB_ROUTINES declares the LIB$ Run-Time Library routines.

❷ The structure of an item list entry is defined using a variant record type.

❸ The VARYING OF CHAR type is a varying-length character string with two components; a word-integer length and a character string body, which in this example is 255 bytes long. The VOLATILE attribute is required for variables which are modified by means other than a direct assignment or as an argument in a routine call.

❹ The functions SIZE and IADDRESS are used to obtain the allocated size of the string body and the address of the string body and length. The returned length will be stored into the length field of the varying string Translated_Name, so that it will appear to be the correct size.

❺ The definition of the $TRNLNM routine in STARLET.PEN contains specifications of the passing mechanism to be used for each argument, thus it is not necessary to specify the mechanism here.

❻ The IF statement performs a logical test following the function reference to see if the service completed successfully. If an error or warning occurs during the service call, the error is signaled.

3
Security Services

The VMS security system services provide various mechanisms that you can use to enhance the security of VMS operating systems. These services include facilities to do the following:

- Create and maintain a rights database.
- Create and translate access control list (ACL) entries.
- Modify a process rights list.
- Check access protection.
- Provide a security erase pattern for disks.
- Control magnetic tape access.

The following table lists the system services related to system security.

Service	Function
$ADD_HOLDER	Adds a holder record to the rights database
$ADD_IDENT	Adds an identifier to the rights database
$ASCTOID	Translates an identifier name to a binary value
$CHANGE_ACL	Creates or modifies an ACL
$CHECK_ACCESS	Invokes a system access protection check on behalf of another user
$CHKPRO	Invokes a system access protection check
$CREATE_RDB	Initializes the rights database
$ERAPAT	Generates a security erase pattern
$FIND_HELD	Returns identifier(s) held by a holder in rights database
$FIND_HOLDER	Returns holder(s) of an identifier in rights database
$FINISH_RDB	Deallocates record stream and clears context value when searching the rights database
$FORMAT_ACL	Formats an ACE into a text string
$FORMAT_AUDIT	Converts a security auditing event message from binary format to ASCII text
$GRANTID	Adds an identifier to the process or the system rights list
$HASH_PASSWORD	Applies the hash algorithm you select to an ASCII password string and returns a quadword hash value that represents the encrypted password
$IDTOASC	Translates an identifier value to its identifier name
$MOD_HOLDER	Modifies a holder record in rights database
$MOD_IDENT	Modifies an identifier record in rights database

Service	Function
$MTACCESS	Controls magnetic tape access
$PARSE_ACL	Converts a text ACE into binary format
$REM_HOLDER	Deletes a holder record from identifier's list of holders in the rights database
$REM_IDENT	Deletes an identifier and all holders of that identifier from rights database
$REVOKID	Removes an identifier from process or system rights list

3.1 Overview of the VMS Protection Scheme

The basis of the VMS security scheme is an **identifier** which is a 32-bit binary value that represents a process to the system. An identifier can represent an individual user, a group of users, or some aspect of the environment in which a user is operating. A process is a **holder** of an identifier when that identifier can represent that process to the system.

The system **rights database** is an indexed file consisting of identifier and holder records. Those records define the identifiers and the holders of those identifiers on a system. When a process logs in to the system, LOGINOUT creates a rights list for the process from the applicable entries in the rights database. Thus, a **process rights list** contains all the identifiers that the process holds. A process can be the holder of a number of identifiers. Each of those identifiers determines the identity and the access rights of the list holder. The process rights list becomes part of the process and is propagated to any created processes.

When a process attempts to access an object in the system, VMS 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.

The VMS protection scheme provides security with the mechanism of the access control list (ACL). An ACL consists of access control list entries (ACEs) that specify the type of access an identifier has to an object like a file, device, or mailbox. 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 rights database.

The following sections describe each of the components of the security scheme—identifiers, rights database, process rights list, and ACLs—and the system services affecting those components.

3.2 Identifiers

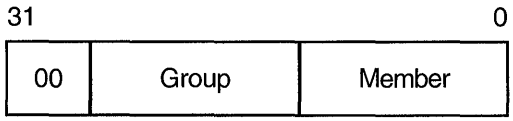
The basic component of the VMS protection scheme is an identifier. This 32-bit binary value represents various types of agents using the system. The types of agents represented include individual users, groups of users, and environments in which a process is operating.

3.2.1 Identifier Format

Identifiers have two formats in the rights database: UIC format and ID format. The high-order bits of the identifier value specify the format of the identifier. Two high-order zero bits identify a UIC format identifier; bit 31, set to 1, identifies an ID format identifier.

Each UIC identifier is unique and represents a system user. The UIC identifier contains the two high-order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382.

The following is a diagram of the UIC format.

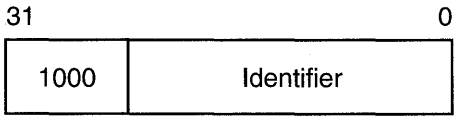


UIC Format

ZK–1905–GE

Bit 31, set to 1, specifies ID format. Bits 30 through 28 are reserved by Digital. The remaining bits specify the identifier value.

The following is a diagram of the ID format.



ID Format

ZK–1906–GE

3.2.2 Identifier Names

To the system an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights database.

An identifier name consists of 1 to 31 alphanumeric characters 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.

3.2.3 System-Defined Identifiers

System-defined identifiers, or environmental identifiers, 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 DECnet–VAX 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, LOGINOUT includes one or more of these identifiers when creating the process rights list.

3.2.4 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.

You define identifiers and their holders in the rights database with the Authorize Utility or with the appropriate system services. You can define an identifier in the rights database to allow users from different UIC groups to hold an identifier. 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 system objects based on the identifiers the user holds, rather than on the user's UIC. See Section 3.4 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 3.3.2.4. Also, for more information, see the *Guide to VMS System Security*.

3.2.5 Identifier Attributes

An identifier has attributes associated with it in the rights database. Part of the process rights list includes the attributes of any identifiers that the process holds. A holder of an identifier can hold an attribute only if the identifier holds the attribute.

Attributes that may be added to identifiers include the DYNAMIC and RESOURCE attributes. The DYNAMIC attribute allows unprivileged holders of an identifier to add or remove the identifier from the process rights list. The RESOURCE attribute allows the holder of an identifier to charge resources, like disk blocks, to an identifier. Conversely, a holder who does not have the RESOURCE attribute cannot charge resources to the identifier, and an unprivileged holder who does not have the DYNAMIC attribute cannot modify the identifier.

The following example demonstrates the advantages of defining an identifier and holder(s) for a project.

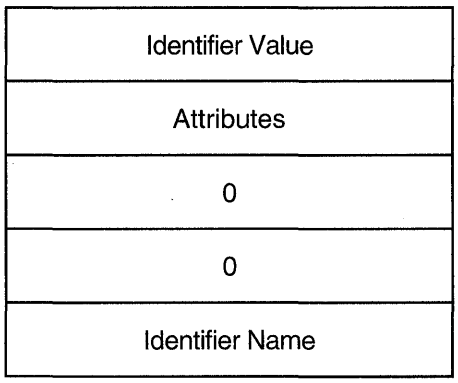
The physics department of a school may have a common library with an associated disk quota on the system. (If disk quotas are in use, you must establish a quota file entry for the identifier to allow anyone to charge space to it.) You want to allow the faculty members to charge any disk quota that they use in conjunction with the library to an identifier, and to prevent the students from charging resources to that identifier. You could define an identifier PHYSICS in

the rights database with the holders FRED, a faculty member, and GEORGE, a student. If you can specify the RESOURCE attribute for FRED, that holder can charge resources to the PHYSICS identifier; if you do not specify the RESOURCE attribute for GEORGE, that holder cannot charge resources to the PHYSICS identifier.

3.3 Rights Database

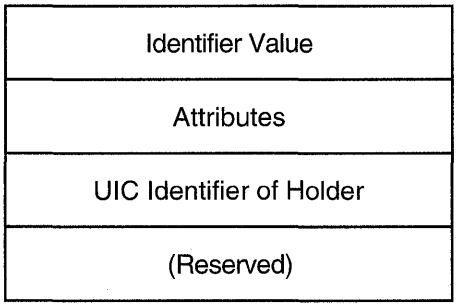
The rights database is an indexed file containing two types of records that define all identifiers: identifier records and holder records.

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. The following figure depicts the 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. A holder record has the following format.



ZK–1907–GE

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 third key is the identifier name. Through the use of the secondary key of the holder ID, all the rights held by a process can be retrieved quickly when LOGINOUT creates the process rights list.

3.3.1 Initializing a Rights Database

The rights database is initialized in one of the following ways:

- When a system is installed or upgraded
- With the Authorize Utility
- With the $CREATE_RDB system service

When you call $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, $CREATE_RDB fails with the error code RMS$_FEX. To create a new rights database when one already exists, you must explicitly delete or rename the old one.

When a rights database is initialized, it is equated to the logical name RIGHTSLIST, which you must define as a system logical name at executive mode. If the logical name does not exist, the rights database is given the default file specification SYS$SYSTEM:RIGHTSLIST.DAT.

When created, RIGHTSLIST.DAT has the default protection (S:RWED,O:RWED,G:RWE,W:R). World read access to the directory in which the database is to be located is required so that all users can read the records in the database. In order to use $CREATE_RBD, write access to the database is necessary. If the database is in SYS$SYSTEM, which is the default, you need SYSPRV privilege to grant write access to the database.

When $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.

3.3.2 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
- Holder(s) of each identifier
- Attribute of each identifier and each holder of each identifier

You can use the Authorize Utility or one of the following system services to add, delete, display, modify, or translate the various elements of the rights database.

Action	Element	Service Used
Translate	Identifier name to identifier binary value	$ASCTOID
	Identifier binary value to identifier name	$IDTOASC
Add	New identifier record	$ADD_HOLDER
	Identifier to holder record	$ADD_IDENT
Find	Identifier value held by holder	$FIND_HELD

Action	Element	Service Used
	Holder(s) of an identifier	$FIND_HOLDER
	All identifiers	$IDTOASC
Modify	Attribute in holder record	$MOD_HOLDER
	Attribute in identifier record	$MOD_IDENT
Delete	Holder from identifier record	$REM_HOLDER
	Identifier and all its holders	$REM_IDENT

The following table shows what access you need with which services.

Service	Required Access
$ADD_HOLDER	Write
$ADD_IDENT	Write
$ASCTOID	Read
$CREATE_RDB	Write
$FIND_HELD	Read
$FIND_HOLDER	Read
$FINISH_RDB	Read
$IDTOASC	Read
$MOD_HOLDER	Write
$MOD_IDENT	Write
$REM_HOLDER	Write
$REM_IDENT	Write

3.3.2.1 Translating Identifier Names and Binary Values

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 $ASCTOID and $IDTOASC system services to translate from one format to another. When you pass to $ASCTOID the address of a string descriptor pointing to an identifier name, the corresponding identifier binary value is returned. Conversely, you use the $IDTOASC service to translate a binary identifier value to an ASCII identifier name string.

You can also use the $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 $IDTOASC until the status code SS$_NOSUCHID is returned. The $IDTOASC service returns the identifier names in alphabetical order. When SS$_NOSUCHID is returned, $IDTOASC clears the context longword and deallocates the record stream. If you complete your calls to $IDTOASC before SS$_NOSUCHID is returned, use $FINISH_RDB to clear the context longword and to deallocate the record stream.

The following programming example uses $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
```

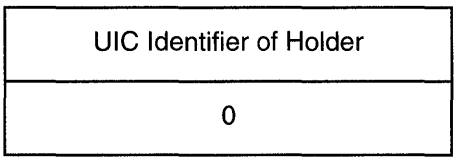
3.3.2.2 Adding Identifiers and Holders to Rights Database

To add identifiers to the rights database, use the $ADD_IDENT service in a program. When you call $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 $ADD_IDENT to specify attributes in the identifier record. Attributes are valid for a holder of an identifier *only* when they are set in both the identifier record and the holder record. The **attrib** argument is a longword containing a bit mask specifying the attributes. The symbol KGB$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.)

When $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 holder(s) of that identifier with the $ADD_HOLDER system service. You pass the binary identifier value with the **id** argument; you specify the holder with the **holder** argument, which is the address of a quadword data structure in the following format.



ZK-1903-GE

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 $ADD_IDENT. Attributes are valid for a holder of an identifier only when they are set in both the identifier record and the holder record.

After $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.

3.3.2.3 Determining Holders of Identifiers

To determine the holders of a particular identifier, use the $FIND_HOLDER service in a program. When you call $FIND_HOLDER, use the **id** argument to pass the binary value of the identifier whose holder you want to determine. On successfully completing execution, $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 $FIND_HOLDER as detailed in Section 3.3.3. Because $FIND_HOLDER identifies the records by the same key (holder ID), it returns the records in the order in which they were written.

3.3.2.4 Determining Identifiers Held

To determine the identifier(s) held by a holder, use the $FIND_HELD service in a program. When you call $FIND_HELD, use the **holder** argument to specify the holder whose identifier is to be found.

On completing execution, $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 $FIND_HELD as detailed in Section 3.3.3. Because $FIND_HELD identifies the records by the same key (identifier), it returns the records in the order in which they were written.

The following programming example uses $FIND_HELD to determine if a user is the holder of a particular identifier. This example also demonstrates how to define an identifier to represent particular terminals.

```
        .title  SECURE_TERMINAL

;
; This module verifies that the user is executing this program
; from one of a set of "secure terminals" as outlined in file:
; SECURITY$:SECURE_TERMINAL.DATA;1.
;
        .psect  SECURE_TERMINAL,LONG

;
; The full names of all "secure terminals" are stored in
; file SECURITY$:SECURE_TERMINAL.DATA;1, which is an indexed file
; containing only the names of the secure terminals.  If a name
; is found in that file, it is considered "secure."
;
        .align  LONG

XAB:    $XABKEY pos = 0, -
                siz = 64

        .align  LONG

FAB:    $FAB    fnm = <SECURITY$:SECURE_TERMINAL.DATA;1>, -
                fac = <GET>, -
                shr = <GET, PUT, UPD, DEL>, -
                org = IDX, -
                rfm = FIX, -
                mrs = 64, -
                xab = XAB

        .align  LONG

RAB:    $RAB    fab = FAB, -
                kbf = BUFFER, -
                ksz = 64, -
                ubf = BUFFER, -
                usz = 64, -
                rac = KEY

;
; Declare the identifier name.
;
NAME:
LENGTH: .blkl   1
        .address        BUFFER
BUFFER: .blkb   64
HOLDER: .blkl   1
        .long   0

ID_NAME:        .ascid  /SECURE_TERMINAL/
ID:     .blkl   1

CONTXT: .long   0
HELD:   .blkl   1

;
; In order to get the name of the particular terminal, we need this item list.
;
        .align  LONG

ITMLST: .word   64
        .word   JPI$_TERMINAL
        .address        BUFFER
        .address        LENGTH
        .long   0

ABORT_: jmp     ABORT
```

```
;
; Here we go ...
;
        .entry  SECURE_TERMINAL,^m<>

;
; First, get the full device name
;
        $GETJPIW_S      itmlst = ITMLST
        blbc    r0, ABORT_

        $OPEN   fab = FAB
        blbc    r0, ABORT_

        $CONNECT        rab = RAB
        blbc    r0, ABORT_

        $GET    rab = RAB
        blbc    r0, ABORT_

;
; If we have gotten here, then our terminal is secure.
;
        $DISCONNECT     rab = RAB
        $CLOSE  fab = FAB

;
; Is this user allowed to use the secure terminals
; (a holder of the SECURE_TERMINAL identifier)?
;
        MOVW    #JPI$_USERNAME, ITMLST+2
        $GETJPIW_S      itmlst = ITMLST

        pushal  LENGTH
        pushal  NAME
        pushal  NAME
        calls   #3, STR$TRIM

        $ASCTOID_S      name = NAME, id = HOLDER
        $ASCTOID_S      name = ID_NAME, id = ID

$1:     $FIND_HELD_S    holder = HOLDER, id = HELD, contxt = CONTXT
        blbc    r0, ABORT

        cmpl    ID, HELD
        bneq    $1

;
; Now pass control on to the program.
;
        calls   #0, @#MAIN_PROGRAM_PROPER
        $EXIT_S R0

        .weak   MAIN_PROGRAM_PROPER

;
; Else kick the user out.
;
        .external       LIB$SIGNAL
LIB$SIGNAL_:
        .address        LIB$SIGNAL
```

```
ABORT:  pushl   #SS$_NOPRIV
        pushl   #0
        pushl   r0
        calls   #3, @LIB$SIGNAL_

        $EXIT_S #SS$_NORMAL

        .end    SECURE_TERMINAL
```

3.3.2.5 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 $MOD_IDENT service in a program. Use the **id** argument to pass the binary value of the identifier whose record you want to modify. To enable attributes, use the **set_attrib** argument, which is a longword containing a bit mask specifying the attributes. The symbol KGB$V_RESOURCE, defined in the system macro library $KGBDEF, sets the RESOURCE bit in the attribute longword, and the symbol KGB$V_DYNAMIC sets the DYNAMIC bit. (You can use the prefix KGB$M rather than KGB$V.)

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 or value, or both, with the **new_name** and **new_value** arguments. **New_name** 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), $MOD_IDENT updates all the corresponding holder records with the new holder identifier value.

When $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.

3.3.2.6 Modifying a Holder Record

To modify a holder record, use the $MOD_HOLDER service in a program. When you call $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 $MOD_HOLDER service to enable or disable the attributes of an identifier in the same way as with $MOD_HOLDER.

When $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 $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$M_DYNAMIC)
            end if
*
* Were we successful?
*
      if (.not. STATUS) then
         NEW_ATTR = OLD_ATTR
         call LIB$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
```

3.3.2.7 Removing Identifiers and Holders from the Rights Database

To remove an identifier and *all of its holders*, use the $REM_IDENT service in a program. When you call $REM_IDENT, use the **id** argument to pass the binary value of the identifier you want to remove. When $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 $REM_HOLDER service in a program. When you call $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, $REM_HOLDER removes the holder from the list of the identifier's holders.

3.3.3 Search Operations

You can search the entire rights database when using the $IDTOASC, $FIND_HELD, and $FIND_HOLDER services. You initialize the context longword to 0, and repeatedly call one of the three services until the status code SS$_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 $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 third key is the identifier name.

During a searching operation, the service obtains the first record with an indexed RMS GET operation. The key used for the GET operation depends on the service. The $FIND_HOLDER service uses the identifier binary value; $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 how the service returns records in a searching operation.

The following table summarizes how records are returned by the $IDTOASC, $FIND_HELD, and $FIND_HOLDER services when used in a searching operation.

Service	Record Order
$IDTOASC	Identifier name order.
$FIND_HELD	First GET operation—holder key. Subsequent records are returned in the order in which they were written.
$FIND_HOLDER	First GET operation—identifier key. Subsequent records are returned in the order in which they were written.

The following programming example uses $IDTOASC, $FINISH_RDB, and $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, which 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
```

3.4 Creating, Translating, and Maintaining Access Control List Entries

An access control list (ACL) is a list of entries defining the type of access allowed to an object in the system like a file, device, or mailbox. When a process attempts to access an object with an associated ACL, the system allows access based on the type of access specified by the entries in the ACL.

To the system, access control list entries (ACEs) are in binary form; however, ACEs are easy to use because they have text string format. You use $FORMAT_ACL and $PARSE_ACL to translate ACEs from one format to another in the same way that $IDTOASC and $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 $CHANGE_ACL system service in a program.

3.4.1 Format of ACE Types

There are four types of ACEs:

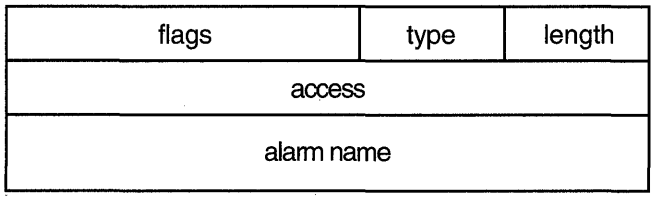
- Alarm
- Application dependent
- Default protection
- Identifier

The alarm ACE defines the types of access to an object that cause a security alarm to be generated. The application ACE contains application-dependent or user-defined information. The default protection ACE defines the default protection for a directory; that protection can be propagated to the files and subdirectories created in that directory. The identifier ACE controls the type of access allowed to a particular user or group of users as specified by an identifier.

An ACE's type determines its format. The following sections describe the format of each of the four types of ACE. Symbols specifying byte offsets and type values are defined in the system macro library ($ACEDEF).

3.4.1.1 Alarm ACE

The access alarm ACE sets a security alarm on an object in the system. The following figure illustrates its format.



ZK–1710–GE

Field	Symbol Name	Description
Length	ACE$B_SIZE	Byte containing the length in bytes of the ACE buffer
Type	ACE$B_TYPE	Byte containing the type value ACE$C_ALARM
Flags	ACE$W_FLAGS	Word containing alarm ACE information and ACE type-independent information
Access	ACE$L_ACCESS	Longword containing a mask indicating the access modes to be watched
Alarm Name	ACE$T_AUDITNAME	Character string containing the alarm name

The flags word contains information specific to alarm ACEs and information applicable to all types of ACEs. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. The following symbols are bit offsets to the alarm ACE information.

Bit	Meaning When Set
ACE$V_SUCCESS	Indicates that the alarm is raised when access is successful
ACE$V_FAILURE	Indicates that the alarm is raised when access fails

The symbols shown in Table 3–1 are bit offsets to ACE information that is applicable to all types of ACE.

Table 3–1 ACE Type-Independent Information

Bit	Meaning When Set
ACE$V_DEFAULT	This ACE is added to the ACL of any file created in the directory whose ACL contains this ACE. This option is applicable only for an ACE in a directory file's ACL.
ACE$V_HIDDEN	This ACE is application dependent. The DCL ACL commands and the ACL editor cannot be used to change the setting; the DCL command DIRECTORY/ACL does not display it.

(continued on next page)

Table 3–1 (Cont.) ACE Type-Independent Information

Bit	Meaning When Set
ACE$V_NOPROPAGATE	This ACE is not propagated among versions of the same file.
ACE$V_PROTECTED	This ACE is not deleted if the entire ACL is deleted; instead this ACE must be explicitly deleted.

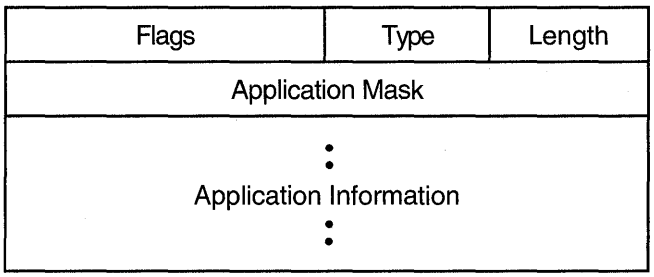
The following symbol values are offsets to bits within the access mask.

Bit	Meaning When Set
ACE$V_READ	Read access is monitored.
ACE$V_WRITE	Write access is monitored.
ACE$V_EXECUTE	Execute access is monitored.
ACE$V_DELETE	Delete access is monitored.
ACE$V_CONTROL	Modification of the access field is monitored.

You can also obtain the symbol values as masks with the appropriate bit set using the prefix ACE$M rather than ACE$V.

3.4.1.2 Application-Dependent ACE

The application ACE contains application-dependent information. The following figure illustrates its format.



ZK–1711–GE

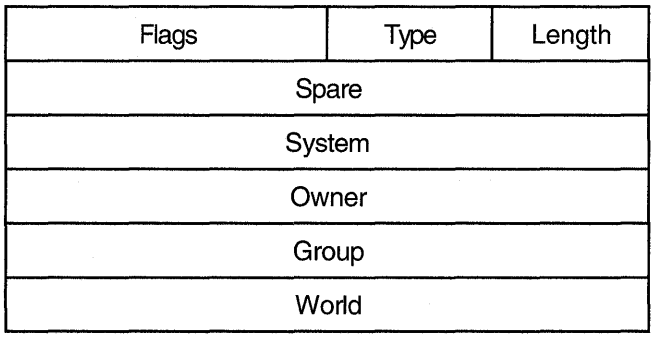
Field	Symbol Name	Description
Length	ACE$B_SIZE	Byte containing the length in bytes of the ACE buffer.
Type	ACE$B_TYPE	Byte containing the type value ACE$C_INFO.
Flags	ACE$W_FLAGS	Word containing application ACE information and ACE type-independent information.
Application Mask	ACE$L_INFO_FLAGS	Longword containing a mask defined and used by the application.
Application Information	ACE$T_INFO_START	Variable-length data structure defined and used by the application. The length of this data is implied by the length field.

The flags word contains information specific to application ACEs and information applicable to all types of ACE. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. For details on the ACE type-independent information, see Table 3–1. The following symbol is a bit offset to the application ACE information.

Bit	Meaning When Set
ACE$V_INFO_TYPE	Four-bit field containing a value indicating whether the application is a CSS application (ACE$C_CSS), a customer application (ACE$C_CUST), or a VMS application (ACE$C_VMS).

3.4.1.3 Default Protection ACE

The default protection ACE specifies the default protection for all files and subdirectories created in the directory. This type of ACE can be used only in the ACL of a directory file. The following figure illustrates its format.



ZK–1712–GE

Field	Symbol Name	Description
Length	ACE$B_SIZE	Byte containing the length in bytes of the ACE buffer.
Type	ACE$B_TYPE	Byte containing the type value ACE$C_DIRDEF.
Flags	ACE$W_FLAGS	Word containing ACE type-independent information.
Spare	ACE$L_SPARE1	Longword reserved for future use and so must be 0.
System	ACE$L_SYS_PROT	Longword containing a mask indicating the access mode granted to system users. Each bit represents one type of access.
Owner	ACE$L_OWN_PROT	Longword containing a mask indicating the access mode granted to the owner. Each bit represents one type of access.
Group	ACE$L_GRP_PROT	Longword containing a mask indicating the access mode granted to group users. Each bit represents one type of access.
World	ACE$L_WOR_PROT	Longword containing a mask indicating the access mode granted to the world. Each bit represents one type of access.

The flags word contains the ACE type-independent information. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. For details, see Table 3–1.

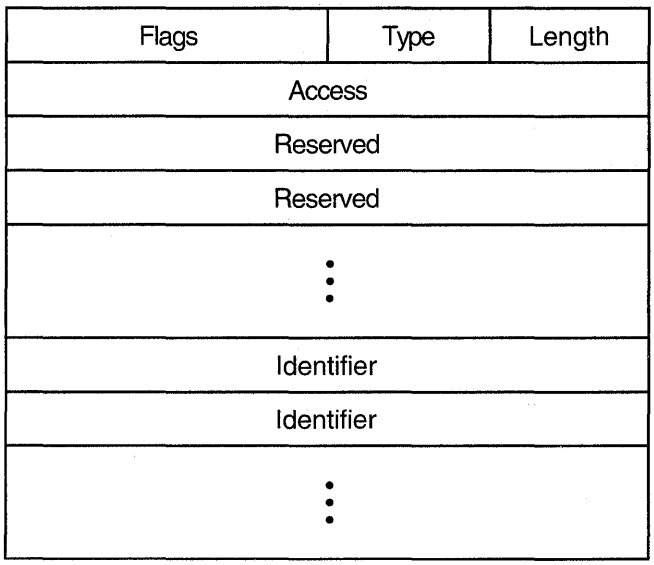
The system interprets the bits within the access mask as shown in the following table. The symbol values are offsets to bits within the mask indicating the access mode granted in the system, owner, group, and world fields.

Bit	Meaning When Set
ACE$V_READ	Read access is granted.
ACE$V_WRITE	Write access is granted.
ACE$V_EXECUTE	Execute access is granted.
ACE$V_DELETE	Delete access is granted.

You can also obtain the symbol values as masks with the appropriate bit set by using the prefix ACE$M rather than ACE$V.

3.4.1.4 Identifier ACE

The identifier ACE controls the type of access allowed based on identifiers. Access is controlled by whether an exact match exists in the process rights list for the identifier(s) in the ACE. The following figure illustrates its format.



ZK–1713–GE

Field	Symbol Name	Description
Length	ACE$B_SIZE	Byte containing the length in bytes of the ACE buffer.
Type	ACE$B_TYPE	Byte containing the type value ACE$C_KEYID.

Field	Symbol Name	Description
Flags	ACE$W_FLAGS	Word containing identifier ACE information and ACE type-independent information.
Access	ACE$L_ACCESS	Longword containing a mask indicating the access mode granted to the specified identifiers.
Reserved	ACE$V_RESERVED	Longwords containing application-specific information. The number of reserved longwords is specified in the flags field.
Identifier	ACE$L_KEY	Longwords containing identifiers. The number of longwords is implied by ACE$B_LENGTH. If an accessor holds all the listed identifiers, the ACE is said to match the accessor and the access specified in ACE$L_ACCESS is granted.

The flags word contains information specific to identifier ACEs and information applicable to all types of ACE. In the flags word, the first byte contains flags specific to each ACE type; the second byte contains flags common to all ACE types. For details on the ACE type-independent information, see Table 3–1. The following symbol is a bit offset to the identifier ACE information.

Bit	Meaning When Set
ACE$V_RESERVED	Four-bit field containing the number of longwords to reserve for application-dependent data. The number must be between 0 and 15. The reserved longwords, if any, immediately precede the identifiers.

The following symbol values are offsets to bits within the mask indicating the access mode granted in the system, owner, group, and world fields.

Bit	Meaning When Set
ACE$V_READ	Read access is granted.
ACE$V_WRITE	Write access is granted.
ACE$V_EXECUTE	Execute access is granted.
ACE$V_DELETE	Delete access is granted.
ACE$V_CONTROL	Modification of the access field is granted.

You can also obtain the symbol values as masks with the appropriate bit set by using the prefix ACE$M rather than ACE$V.

3.4.2 Translating ACEs

To translate ACEs from binary format into a text string, use the $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; the second byte contains the type, which in turn defines the format of the ACE. The following four values specify ACE type.

Value	ACE Type
ACE$C_ALARM	Alarm ACE
ACE$C_INFO	Application-dependent ACE
ACE$C_DIRDEF	Default protection ACE
ACE$C_KEYID	Identifier 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 that define the names of the bits in the access mask of the ACE. 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 $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 $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; the second byte contains the type, which in turn defines the format of the ACE. If $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 $FORMAT_ACL are used.

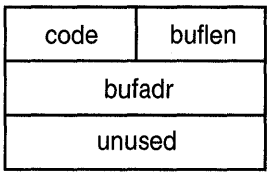
3.4.3 Creating and Maintaining ACEs

To create or modify an ACL associated with a system object, you use the $CHANGE_ACL service. You specify the object whose ACL is to be modified with either the **chan** argument, which specifies the I/O channel associated with the object, or the **objnam** argument, which specifies the object name. If you specify **objnam**, **chan** must be omitted or specified as 0. The **objtyp** argument specifies the type of object.

The values specifying object type are as follows.

ACL$C_DEVICE	Object is a device
ACL$C_FILE	Object is a Files–11 On-Disk Structure Level 2 file
ACL$C_GROUP_GLOBAL_SECTION	Object is a group global section
ACL$C_JOBCTL_QUEUE	Object is a batch or print queue
ACL$C_LOGICAL_NAME_TABLE	Object is a logical name table
ACL$C_SYSTEM_GLOBAL_SECTION	Object is a system global section

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. The **itmlst** argument is an item list specifying the changes to be made to the ACL. Each item code consists of three elements. The following figure illustrates the format of the item code.



ZK-1701-GE

The item list ends with a longword containing the value 0. The **buflen** argument contains the number of bytes in the buffer containing information passed to or from $CHANGE_ACL pointed to by **bufadr**. The third longword of the standard item descriptor is not used by $CHANGE_ACL and should be 0.

The item code specifies the change to be made to the ACL. The following symbols for the item codes are defined in the system macro library ($ACLDEF).

Bit	Meaning When Set
ACL$C_ACLLENGTH	Returns the size, in bytes, of the object's ACL. The **bufadr** argument points to a longword that contains the size.
ACL$C_ADDACLENT	Adds an ACE to the beginning of the ACL when **contxt** is 0, to the end of the ACL when **contxt** is –1, or at a location pointed to by a prior ACL$C_FNDACETYP or ACL$C_FNDACLENT. The **bufadr** argument points to a variable-length data structure containing the ACE to be added. You can add more than one ACE with ACL$C_ADDACLENT; however, **buflen** must contain the total size of all ACEs contained in the buffer.
ACL$C_DELACLENT	Deletes the ACE pointed to by **bufadr** or, if **bufadr** is specified as 0, the ACE pointed to by a prior ACL$C_FNDACETYP or ACL$C_FNDACLENT.
ACL$C_DELETEACL	Deletes the entire ACL with the exception of protected ACEs.
ACL$C_FNDACETYP	Locates an ACE of the type pointed to by **bufadr**.
ACL$C_FNDACLENT	Locates the ACE pointed to by **bufadr**.
ACL$C_RLOCK_ACL	Obtains a read lock on an object in order to lock out all writers from the object's ACL. Regardless of the caller's mode, ACL locks are user-mode locks so that all access modes interlock ACLs correctly.
ACL$C_WLOCK_ACL	Obtains an exclusive lock on an object in order to lock out all other readers and writers from the object's ACL. Regardless of the caller's mode, ACL locks are user mode locks so that all access modes interlock ACLs correctly.
ACL$C_MODACLENT	Replaces the ACE pointed to by a prior ACL$C_FNDACETYP or ACL$C_FNDACLENT with the ACE pointed to by **bufadr**.

Bit	Meaning When Set
ACL$C_READACE	Reads the ACE pointed to by ACL$C_FNDACETYP or ACL$C_FNDACLENT into the buffer pointed to by **bufadr**.
ACL$C_READACL	Reads the object's ACL. The **contxt** argument should be initially set to 0. Complete ACEs are read into the buffer pointed to by **bufadr**.
ACL$C_UNLOCK_ACL	Releases the lock obtained with ACL$C_RLOCK_ACL or ACL$C_WLOCK_ACL.

When you add an ACE with ACL$C_ADDACLENT or locate an ACE with ACL$C_FNDACETYP or ACL$C_FNDACLENT, $CHANGE_ACL searches the ACL for a match for the ACE in the ACE buffer. The $CHANGE_ACL service does not always make a match based on the entire ACE buffer; instead, the ACE type determines how $CHANGE_ACL makes a match. For example:

- A default protection ACE (ACE$C_DIRDEF) matches only on the type field (ACE$B_TYPE). An ACL can have only one default protection ACE because $CHANGE_ACL stops searching when it locates a match.
- An identifier ACE (ACE$C_KEYID) matches on the flags (ACE$W_FLAGS) and identifier (ACE$L_KEY) fields.
- An alarm ACE (ACE$C_ALARM) matches on the flags (ACE$W_FLAGS) and access mask (ACE$L_ACCESS) fields.
- All other ACE types match on the entire ACE buffer.

Because $CHANGE_ACL uses these matching rules, adding an ACE sometimes results in the replacement of another ACE. For example, if you add an identifier ACE with the same flags and identifier fields but with a different access mask, the new ACE replaces the old ACE. When you add an ACE on the top of an ACL, $CHANGE_ACL deletes any matching ACE because it is not seen. If you add an ACE below a matching ACE in an ACL, the added ACE has no effect because it is not seen.

The following programming example uses $CHANGE_ACL to add an ACE to the ACL of a terminal. (See Section 3.6 for a related example.)

```
Module SECURE (main = MAIN, addressing_mode(external=general)) =
begin

!
!       Insert a record into the specified terminal's ACL so that
!       holders of the SECURE_TERMINAL identifier may do confidential
!       work with that terminal.
!
!       To use: $ SECURE tt20:
!
!       Confidential applications will, of course, need to use
!       SYS$CHKPRO to verify that users are authorized to use them.
!

   library
      'SYS$LIBRARY:LIB';
   forward routine
      MAIN;
   external routine
```

```
    LIB$GET_FOREIGN,          ! To get the name of the terminal
    SYS$CHANGE_ACL,           ! To make the actual changes to the ACL
    SYS$PARSE_ACL;            ! To translate the ACE from ASCII

compiletime

    POSITION = 0;

macro

!
!       Some of the routines require dynamic string descriptors.
!

    DYNAMIC_DESCRIPTOR =
        block[DSC$K_D_BLN, byte]
        preset( [DSC$B_CLASS] = DSC$K_CLASS_D, [DSC$B_DTYPE] = 0,
                [DSC$W_LENGTH] = 0, [DSC$A_POINTER] = 0 ) %,

!
!       These two macros are used solely for initializing the access name table.
!

    INITIALIZE[BIT_NUMBER, BIT_NAME] =
        [BIT_NUMBER, DSC$W_LENGTH] = %charcount(BIT_NAME),
        [BIT_NUMBER, DSC$A_POINTER] = uplit byte(BIT_NAME) %,

    IGNORE(START, FINISH)[] =
        %if START leq FINISH %then
            [START, DSC$W_LENGTH] =
                %charcount(%string('BIT_', START)),
            [START, DSC$A_POINTER] =
                uplit byte(%string('BIT_', START))
            %if START lss FINISH %then , %fi
            %assign(POSITION, START+1)
            IGNORE(%number(POSITION), FINISH)
        %fi %;

own

    STATUS,

    OBJNAM: DYNAMIC_DESCRIPTOR,               ! The name of this terminal
    BUFADR: block[ACL$S_ADDACLENT, byte],     ! The new ACE

    ACLENT: block[DSC$K_D_BLN, byte]
            preset( [DSC$W_LENGTH] = ACL$S_ADDACLENT,
                    [DSC$A_POINTER] = BUFADR ),

    ITMLST: $ITMLST_DECL(),

!
!       The Access Name Table:
!
!               Here we specify the ASCII names of all the access types.
!

    ACCNAM: blockvector[32, DSC$K_S_BLN, byte]
            preset( INITIALIZE( 0, 'READ',
                                1, 'WRITE',
                                2, 'LOGICAL',
                                3, 'PHYSICAL',
                                4, 'CONTROL',

                                5, 'CONFIDENTIAL' ),  ! Our hero !

                    IGNORE(6, 31) );
```

```
!
!       Prompt the user for the terminal's name.
!       Create a new ACE.
!       Add the ACE to the ACL of the terminal.
!

   routine MAIN =
   begin

      LIB$GET_FOREIGN(OBJNAM, %ascid'Device: ');
      SYS$PARSE_ACL(%ascid'(IDENTIFIER=SECURE_TERMINAL,ACCESS=CONFIDENTIAL)',
                    ACLENT, 0, ACCNAM);

      $ITMLST_INIT( itmlst = ITMLST,
                    ( itmcod = ACL$C_ADDACLENT,
                      bufsiz = .BUFADR[ACE$B_SIZE],
                      bufadr = BUFADR ) );

      if not
         (STATUS = SYS$CHANGE_ACL(0, %ref(ACL$C_DEVICE), OBJNAM, ITMLST, 0,0,0))
      then
         signal_stop(.STATUS);

      return SS$_NORMAL;

   end;

end
eludom
```

3.5 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 a 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 subsystem can alter the process or system rights list with the $GRANTID or $REVOKID services. These services are not intended for the general system user. The $GRANTID service adds an identifier to a rights list or, if the identifier is already part of the rights list, the $GRANTID service modifies the attributes of the identifier. The $REVOKID service removes an identifier from a rights list. If the identifier, specified by either **id** or **name**, is the holder of any other identifiers, the identifier is removed from those holder records.

The $GRANTID and $REVOKID services treat the **pidadr** and **prcnam** arguments the same way all other process control services treat these arguments. For more details, see the *Guide to VMS System Security*.

You may 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 may not 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 *VMS DCL Dictionary*.

3.6 Checking Access Protection

VMS provides two system services that 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 but on behalf of a third party attempting access to an object. These services are described in the following subsections.

3.6.1 SYS$CHKPRO

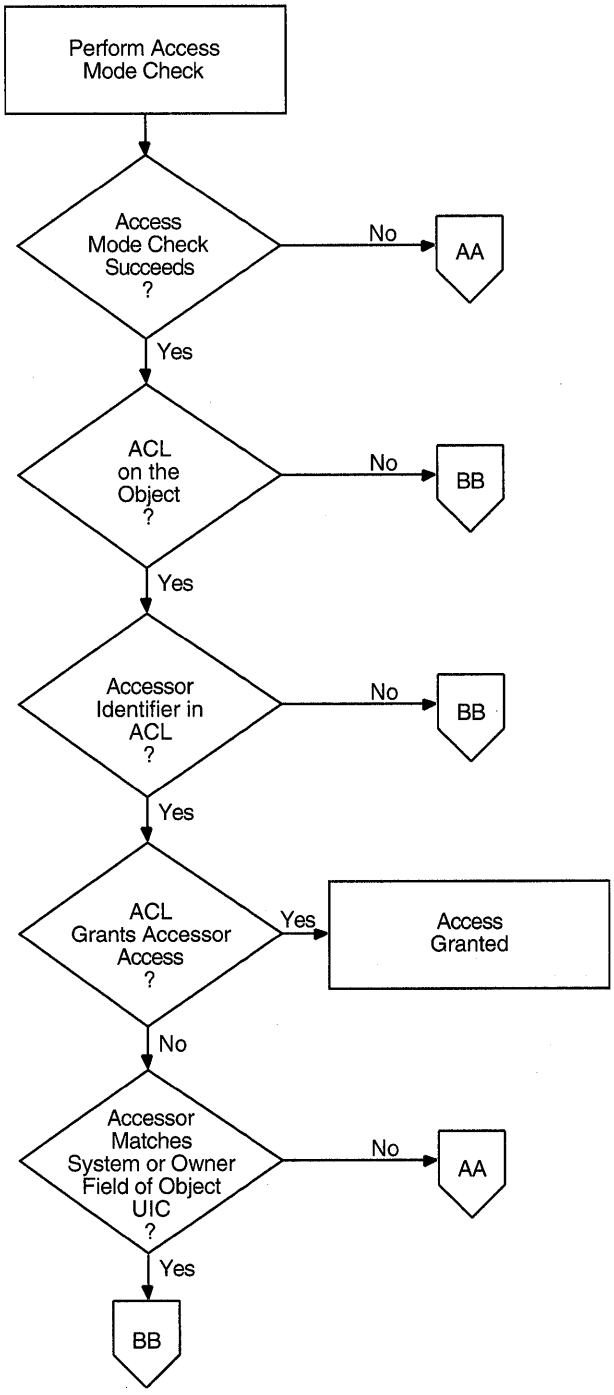
The $CHKPRO service invokes the access protection check used by the system. The service does not grant or deny access; rather, it performs the protection check on behalf of a layered product, application program, or other similar subsystem that in turn must specifically grant or deny access.

To pass the input and output information to $CHKPRO, use the **itmlst** argument, which is the address of an item list of descriptors. The $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, $CHKPRO returns the status SS$_NORMAL; if the accessor cannot access the object, $CHKPRO returns the status SS$_NOPRIV. The $CHKPRO service does not grant or deny access. The subsystem itself must grant or deny access based on the output (SS$_NORMAL or SS$_NOPRIV) from $CHKPRO.

The $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 $CHKPRO, see the description of $CHKPRO in the *VMS System Services Reference Manual.*

Figure 3–1 provides a flowchart of the steps that $CHKPRO follows when performing a protection check.

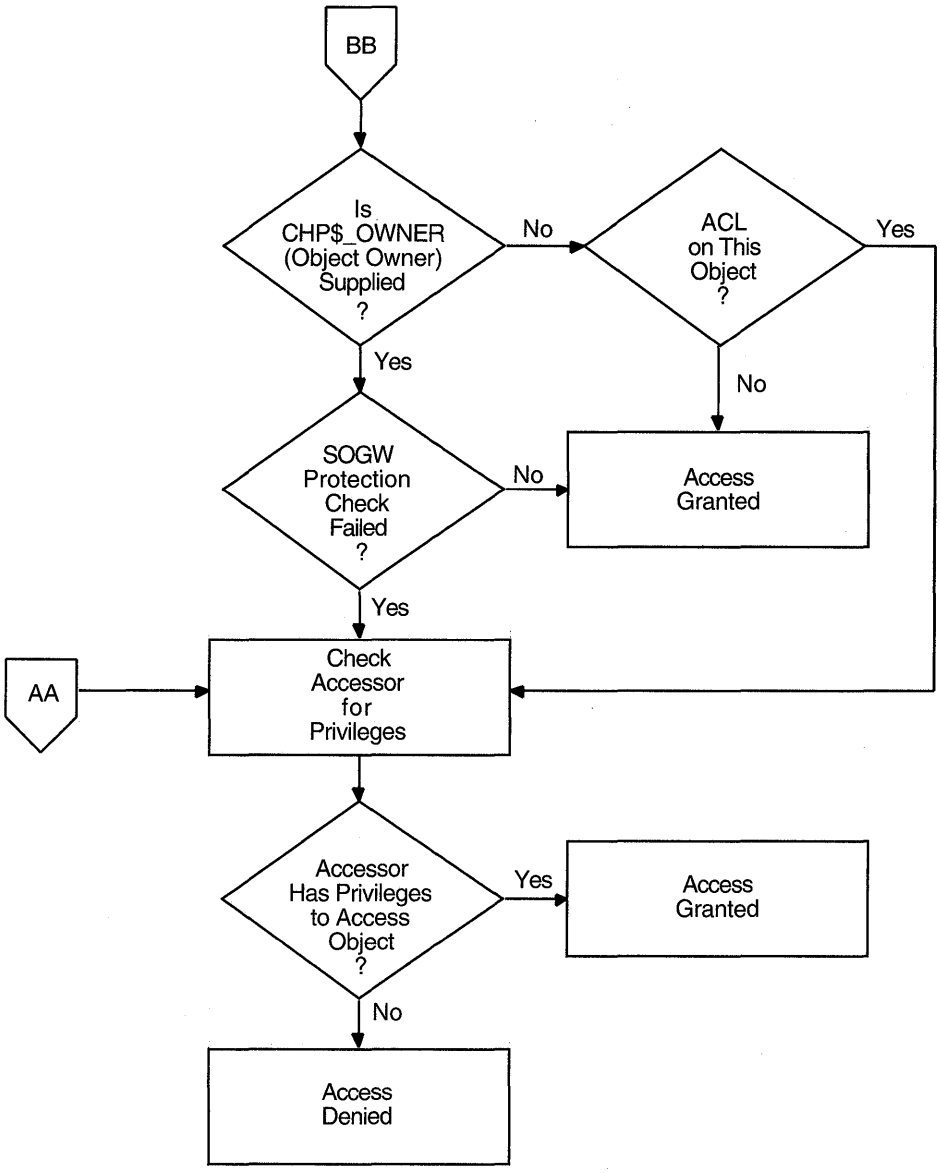
Figure 3–1 Flowchart of $CHKPRO Operation



ZK–6375.1–GE

(continued on next page)

Figure 3–1 (Cont.) Flowchart of $CHKPRO Operation



ZK–6375.2–GE

3.6.2 SYS$CHECK_ACCESS

Whereas SYS$CHKPRO performs the system access protection check on a user attempting access to an object, SYS$CHECK_ACCESS executes the protection check on behalf of a third-party accessor. An example of this would be 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 $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 arguments **usrnam**, **objnam**, and **objtyp**, respectively. The $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, $CHECK_ACCESS returns the status SS$_NORMAL; if the accessor cannot access the object, $CHECK_ACCESS returns the status SS$_NOPRIV.

The $CHECK_ACCESS service does not grant or deny access. The subsystem itself must grant or deny access based on the output (SS$_NORMAL or SS$_ NOPRIV) from $CHECK_ACCESS.

The $CHECK_ACCESS 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 $CHECK_ACCESS, see the description of $CHECK_ACCESS in the *VMS System Services Reference Manual*.

The following programming example uses $CHKPRO to verify that a user is authorized to use a terminal for confidential work. The $CHKPRO service does not explicitly grant access; it only performs the protection check. The application itself must grant or deny access based on the output from $CHKPRO. See Section 3.4.3.

```
Module CHECK (main = MAIN, addressing_mode(external=general)) =
begin

   library

      'SYS$LIBRARY:LIB';

   forward routine

      MAIN;

   external routine

      SYS$CHKPRO,
      SYS$CHANGE_ACL,
      LIB$GET_VM;

   own

      STATUS,

      ACLLENGTH,
      ACL: ref block[, byte],

      ITMLST1: $ITMLST_DECL(),
      ITMLST2: $ITMLST_DECL(items=2);

   routine MAIN =
   begin

!
!       Query for the size of the user terminal's ACL.
!

      $ITMLST_INIT( itmlst = ITMLST1,
         ( itmcod = ACL$C_ACLLENGTH, bufadr = ACLLENGTH ) );
      SYS$CHANGE_ACL(0, %ref(ACL$C_DEVICE), %ascid'TT:', ITMLST1, 0,0,0);

!
!       Allocate memory to store the ACL.
!

      LIB$GET_VM(%ref(.ACLLENGTH), ACL);

!
!       Read the entire ACL into the buffer.
!

      $ITMLST_INIT( itmlst = ITMLST1,
         ( itmcod = ACL$C_READACL, bufadr = .ACL, bufsiz = .ACLLENGTH ) );
      SYS$CHANGE_ACL(0, %ref(ACL$C_DEVICE), %ascid'TT:', ITMLST1, 0,0,0);
```

```
!       Check the object for CONFIDENTIAL (BIT_5) access.
!

      $ITMLST_INIT( itmlst = ITMLST2,
         ( itmcod = CHP$_ACL, bufadr = .ACL, bufsiz = .ACLLENGTH ),
         ( itmcod = CHP$_ACCESS, bufadr = uplit(%b'100000') ) );

      if not (STATUS = SYS$CHKPRO(ITMLST2)) then
         signal_stop(.STATUS);

      return SS$_NORMAL;

   end;

end
eludom
```

3.7 Additional Security Services

The VMS operating system provides four additional system services that affect system security:

- The $ERAPAT service provides a consistent mechanism by which users can write a security erase pattern for disks. The security erase patterns can be custom configured to fit the individual needs of a site.
- The $FORMAT_AUDIT service converts a security auditing event message from binary format to ASCII text. Event messages can come from either the audit server listener mailbox or the system security audit log file.
- The $HASH_PASSWORD service applies the hash algorithm you select to an ASCII password string and returns a quadword hash value that represents the encrypted password.
- The $MTACCESS service checks the accessibility field in a magnetic tape label to determine if a volume is protected by VMS.

For more information, see the descriptions of $ERAPAT, $FORMAT_AUDIT, $HASH_PASSWORD, and $MTACCESS in the *VMS System Services Reference Manual*.

4

Event Flag Services

Event flags are status posting bits maintained by VMS for general programming use. Programs can use event flags to perform a variety of signaling functions. Event flag services clear, set, and read event flags. They also can place a process in a wait state pending the setting of an event flag or flags. The following system services are event flag services:

- Associate Common Event Flag Cluster ($ASCEFC)
- Disassociate Common Event Flag Cluster ($DACEFC)
- Delete Common Event Flag Cluster ($DLCEFC)
- Set Event Flag ($SETEF)
- Clear Event Flag ($CLREF)
- Read Event Flags ($READEF)
- Wait for Single Event Flag ($WAITFR)
- Wait for Logical OR of Event Flags ($WFLOR)
- Wait for Logical AND of Event Flags ($WFLAND)

Some system services set an event flag to indicate the completion or the occurrence of an event; the calling program can test the flag. The following are some of the system services that use event flags to signal events to the calling process:

- Enqueue Lock Request ($ENQ and $ENQW)
- Get Device/Volume Information ($GETDVI and $GETDVIW)
- Get Job/Process Information ($GETJPI and $GETJPIW)
- Get Systemwide Information ($GETSYI and $GETSYIW)
- Queue I/O Request ($QIO and $QIOW)
- Set Timer ($SETIMR)
- Update Section File on Disk ($UPDSEC)
- Update Section File on Disk and Wait ($UPDSECW)

Event flags can be used by more than one process as long as the cooperating processes are in the same 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.

4.1 Event Flag Numbers and Event Flag Clusters

Each event flag has a unique decimal 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 $QIO system service, then event flag number 1 is set when the I/O operation completes.

To allow manipulation of groups of event flags, the flags are ordered in clusters, with 32 flags in each cluster, numbered from right to left, corresponding to bits 0 through 31 in a longword. The clusters are also numbered from 0 to 3. 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.

There are two types of clusters: local event flag clusters and common event flag clusters.

- A local event flag cluster can only be used internally by a single process. Local clusters are automatically available to each process.
- A common event flag cluster can be shared by cooperating processes in the same group. Before a process can refer to a common event flag cluster, it must explicitly "associate" with the cluster. Association is described in Section 4.5.

The ranges of event flag numbers and the clusters to which they belong are summarized in Table 4–1.

Table 4–1 Summary of Event Flag and Cluster Numbers

Cluster Number	Event Flag Numbers	Description	Restriction
0	0–31	Process-local event flag clusters for general use	Event flags 24 through 31 reserved for system use
1	32–63		
2	64–95	Assignable common event flag cluster	Must be associated before use
3	96–127		

Specifying Event Flag and Event Flag Cluster Numbers

The same system services manipulate flags in both local and common event flag clusters. Because the event flag number implies the cluster number, it is not necessary to 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.

To prevent accidental use of an event flag already in use elsewhere in your program, you should allocate and deallocate local event flags. The *VMS Run-Time Library Routines Volume* describes routines you can use to allocate an arbitrary event flag (LIB$GET_EF), to allocate a particular event flag (LIB$RESERVE_EF), or to deallocate an event flag (LIB$FREE_EF) from the process-wide pool of available local event flags. No similar routines exist for common event flags.

4.2 Examples of Event Flag Services

Local event flags are most commonly used in conjunction with other system services. For example, you can use the Set Timer ($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, you could specify an event flag number for the $SETIMR service and then use the Wait for Single Event Flag ($WAITFR) system service, as follows.

```
TIME:   .BLKQ   1               ; Will contain time interval to wait
          .
          .
          .
        $SETIMR_S -             ; Set the timer
                EFN=#3, -
                DAYTIM=TIME
        $WAITFR_S -
                EFN=#3          ; Wait until timer expires
```

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 10.

4.3 Event Flag Waits

The following three system services place the process in a wait state until an event flag, or group of event flags, is set:

- The Wait for Single Event Flag ($WAITFR) system service places the process in a wait state until a *single* flag has been set.
- The Wait for Logical OR of Event Flags ($WFLOR) system service places the process 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 ($WFLAND) system service places the process 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 ($QIO) system service. The following example shows a program segment that issues two $QIO system service calls, and uses the $WFLAND system service to wait until both I/O operations complete before it continues execution.

```
$QIO_S  EFN=#1,... ❶          ; Issue first I/O request
BSBW    ERROR                   ; Check for error
$QIO_S  EFN=#2,...              ; Issue second I/O request
BSBW    ERROR                   ; Check for error
$WFLAND_S -  ❷                ; Wait until both complete
        EFN=#1, - ❸
        MASK=#^B0110
BSBW    ERROR                   ; Check for error
  .
  .                             ; Continue execution
  .
```

❶ The event flag argument is specified in each $QIO request. Both of these event flags are in cluster 0.

❷ After both I/O requests are successfully queued, the program calls the Wait for Logical AND of Event Flags ($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 that flags 1 and 2 are to be waited for.

❸ Note that the $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 carefully coordinated. (See Section 7.3.1 for more information about the recommended technique for testing I/O completion.)

4.4 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 ($SETEF) and Clear Event Flag ($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 $SETEF and $CLREF services return successful status codes that indicate whether the specified flag was set or clear when the service was called. The caller can thus determine the previous state of the flag, if necessary. The codes returned are SS$_WASSET and SS$_WASCLR.

All event flags in a common event flag cluster are initially clear when the cluster is created. The next section describes the creation of common event flag clusters.

4.5 Creating Common Event Flag Clusters

Common event flags act as a communication link between images executing in different processes in the same group. 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 Section 2.5.1.

Before any processes can use event flags in a common event flag cluster, the cluster must be created. The Associate Common Event Flag Cluster ($ASCEFC) system service creates a common event flag cluster. After a cluster is created, other processes in the same group can call $ASCEFC to establish their association with the cluster, so they can access flags in it.

When a common event flag cluster is created, it must be identified by a name string. (Section 4.8 explains the format of this string.) Each process that associates with the cluster must use the same name to refer to it; the $ASCEFC system service establishes correspondence between the cluster name and the cluster number that a process assigns to 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.

```
CLUSTER:
        .ASCID  /COMMON_CLUSTER/        ; Cluster name
          .
          .
          .
        $ASCEFC_S -                     ; Create cluster 2
                EFN=#65, -
                NAME=CLUSTER
```

Subsequently, other processes in the same group may 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.

Common event flag clusters are either temporary or permanent. The **perm** argument to the $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. Disassociation can be performed explicitly, with the Disassociate Common Event Flag Cluster ($DACEFC) system service, or implicitly, when the image exits.

Permanent clusters require the creating process to have the PRMCEB user privilege. They continue to exist until they are explicitly marked for deletion with the Delete Common Event Flag Cluster ($DLCEFC) system service.

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 $ASCEFC system service creates the cluster.

4.6 Disassociating and Deleting Common Event Flag Clusters

When a process no longer needs access to a common event flag cluster, it issues the Disassociate Common Event Flag Cluster ($DACEFC) system service. When all processes associated with a temporary cluster have issued a $DACEFC system service, the system deletes the cluster. If a process does not explicitly disassociate itself from a cluster, the system performs an implicit disassociation when the image that called $ASCEFC exits.

Permanent clusters, however, must be explicitly marked for deletion with the Delete Common Event Flag Cluster ($DLCEFC) system service. After the cluster has been marked for deletion, it is not deleted until all processes associated with it have been disassociated.

4.7 Example of Using a Common Event Flag Cluster

The following is an example of 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

CNAME:  .ASCID /TITUS/  ; Descriptor for cluster name
           .
           .
❶ $ASCEFC_S -    ; Create common cluster
                 EFN=_#64, -
                 NAME=CNAME ❷
        BSBW     ERROR   ; Check for error
           .
           .
❸ $WFLAND_S -
                 EFN=_#64, -
                 MASK=_#_^B1110  ; Wait for flags 1,2,3
        BSBW     ERROR   ; Check for error
❹ $DACEFC_S -
                 EFN=_#64        ; Disassociate cluster

Process CYGNUS

ORION_FLAGS: .ASCID /TITUS/    ; Descriptor for
           .                   ; cluster name
           .
❺ $ASCEFC_S -
                 EFN=_#64, -
                 NAME=ORION_FLAGS
        BSBW     ERROR   ; Check for error
        $SETEF_S -      ; Set event flag 1
                 EFN=_#65
        BSBW     ERROR   ; Check for error
        $DACEFC_S -     ; Disassociate
                 EFN=_#64

Process LYRA

SHARE: .ASCID /TITUS/   ; Descriptor for cluster name
           .
           .
❻ $ASCEFC_S -    ; Associate with cluster 3
                 EFN=_#96, -
                 NAME=SHARE
        BSBW     ERROR   ; Check for error
        $SETEF_S -      ; Set flag 3
                 EFN=_#99
        BSBW     ERROR   ; Check for error
        $DACEFC_S -     ; Disassociate
                 EFN=_#96

Process PEGASUS

CLUSTER: .ASCID /TITUS/ ; Descriptor for cluster name
           .
           .
❼ $ASCEFC_S -    ; Associate with cluster
                 EFN=_#64, -
                 NAME=CLUSTER
        BSBW     ERROR   ; Check for error
        $WAITFR_S -     ; Wait for flag 1
                 EFN=_#65
        BSBW     ERROR   ; Check for error
           .             ; Continue
           .
        $SETEF_S -      ; Set flag 2
                 EFN=_#66
        BSBW     ERROR   ; Check for error
        $DACEFC_S -     ; Disassociate
                 EFN=_#64
```

❶ Assume for this example that ORION is the first process to issue the $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.

❷ The argument **name** in the $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, containing event flags 64 through 95. Cooperating processes CYGNUS, LYRA, and PEGASUS must use the same character string name to refer to this cluster.

❸ The continuation of process ORION depends on work done by processes CYGNUS, LYRA, and PEGASUS. The Wait for Logical AND of Event Flags ($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.

❹ When all three event flags are set, process ORION continues execution and calls the $DACEFC system service. Because ORION did not specify the **perm** argument when it created the cluster, TITUS is deleted.

❺ Process CYGNUS executes, associates with the cluster, sets event flag 65 (flag 1 in the cluster), and disassociates.

❻ 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 is setting flag number 3 in TITUS.

❼ Process PEGASUS associates with the cluster, waits for an event flag set by process CYGNUS, and sets an event flag itself.

4.8 Cluster Name

The **name** argument to the Associate Common Event Flag Cluster ($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.

```
NAMEDESC:
        .ASCID  /CLUS_RT/       ; Descriptor for logical name of cluster
        .
        .
        .
        $ASCEFC_S -
                ...,NAME=NAMEDESC,...
```

The following logical name translation takes place:

1. CEF$ is prefixed to CLUS_RT.
2. CEF$CLUS_RT is translated to CLUS_RT_001. (No further translation is successful. 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 VMS operating system strips the underscore and considers the resultant string to be the actual name (that is, no further translation is performed).
- If the name string is the result of a logical name translation, the name string is checked to see if it has the "terminal" attribute. If the name string is marked with the "terminal" attribute, VMS considers the resultant string to be the actual name (that is, no further translation is performed).

4.9 Example of Using Event Flag 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, 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))
.
.
.
```

5

AST (Asynchronous System Trap) Services

Some system services allow a process to request that it be interrupted when a particular event occurs. Because the interrupt occurs asynchronously (out of sequence) with respect to the process's execution, the interrupt mechanism is called an asynchronous system trap (AST). The trap provides a transfer of control to a user-specified procedure that handles the event.

The following system services are AST services:

- Set AST Enable ($SETAST)
- Declare AST ($DCLAST)
- Set Power Recovery AST ($SETPRA)

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.

The following are some of the services that use ASTs:

- Declare AST ($DCLAST)
- Enqueue Lock Request ($ENQ)
- Get Device/Volume Information ($GETDVI)
- Get Job/Process Information ($GETJPI)
- Get Systemwide Information ($GETSYI)
- Queue I/O Request ($QIO)
- Set Timer ($SETIMR)
- Set Power Recovery AST ($SETPRA)
- Update Section File on Disk ($UPDSEC)

For example, if you call the Set Timer ($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.

The following example shows a typical program that calls the $SETIMR system service with a request for an AST when a timer event occurs.

```
NOON:   .BLKQ   1               ; Will contain 12:00 system time
        .ENTRY  LIBRA,0         ; Entry mask for LIBRA
          .
          .
          .
❶     $SETIMR_S -               ; Set timer
                DAYTIM=NOON, -
                ASTADR=TIMEAST
        BSBW    ERROR           ; Check for error
          .
          .                              +---------+
          .  <---------------------------! Timer   !
          .                              !Interrupt! ❷
          .                              +---------+
        .ENTRY  TIMEAST,^M<>            ; Entry mask for AST routine
        .                              ; Handle timer request
❸       .
        RET                             ; Done
        .END    LIBRA
```

❶ The call to the $SETIMR system service requests an AST at 12:00 noon.

The DAYTIM argument refers to the quadword NOON, which must contain the time in system time (64-bit) format. For details on how this is done, see Chapter 10. The ASTADR argument refers to TIMEAST, the address of the AST service routine.

When the call to the system service completes, the process continues execution.

❷ The timer expires at 12:00 noon and notifies the system. The system interrupts execution of the process and gives control to the AST service routine.

❸ The user routine TIMEAST 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.

5.1 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 5.5.

5.2 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.

5.2.1 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 detail in Chapter 4.

5.2.2 Hibernation

A process can place itself in a wait state with the Hibernate ($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 8.

5.2.3 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.

5.3 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 $QIO or $SETIMR request) when the process requests notification by means of an AST.

The Declare AST ($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.

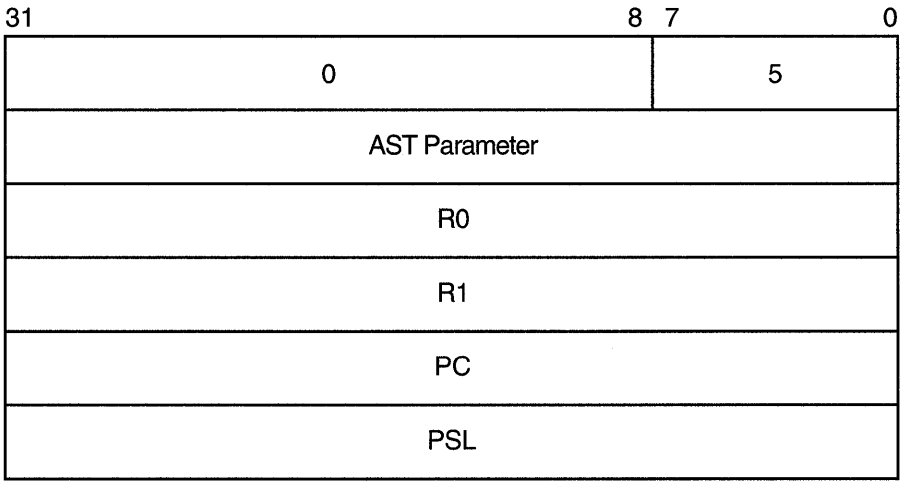
You may find occasional use for the $DCLAST system service in your programming applications; you may also find the $DCLAST service useful when you want to test an AST service routine.

5.4 The AST Service Routine

An AST service routine must be a separate procedure. The system calls the AST with a CALLG instruction; the routine must return using a RET instruction. If the service routine modifies any registers other than R0 or R1, it must set the appropriate bits in the entry mask so that the contents of those registers are saved.

Because knowing when the AST service routine will begin executing is impossible, 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 Argument Pointer register (AP) points to an argument list that has the following format.



ZK–0855–GE

The registers R0 and R1, the PC, and the PSL in this list are those that 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 $DCLAST system service, you can supply a value for the AST parameter. If you do not specify a value, it defaults to 0.

The following example illustrates an AST service routine. In this example, the ASTs are queued by the $DCLAST system service; the ASTs are delivered to the process immediately so that the service routine is called following each $DCLAST system service call.

```
        .ENTRY  CELESTEF,0          ; Entry mask
          .
❶       $DCLAST_S -                 ; AST with parameter=1
                ASTADR=ASTRTN, -
                ASTPRM=#1
            .
            .
            .
        $DCLAST_S -                     ; AST with parameter=2
                ASTADR=ASTRTN, -
                ASTPRM=#2
            .
            .
            .
        RET                             ; Return control
;
ASTRTN: .WORD   0                       ; Entry mask
  ❷     CMPL    #1,4(AP)            ; Check if AST parameter=1
        BEQL    10$                     ; If equal, goto 10$
        CMPL    #2,4(AP)                ; Check if AST parameter=2
        BEQL    20$                     ; If equal, goto 20$
          .
10$:      .                             ; Handle first AST
        RET                             ; Return
20$:      .                             ; Handle second AST
        RET                             ; Return
          .
        .END CELESTEF
```

❶ The program CELESTEF calls the $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.

❷ 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.

5.5 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 ($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.

- 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.

5.6 Example of Using AST Services

The following is an example of a VAX FORTRAN program that finds the 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
```

6
Name Services

The VMS name services include the logical name services and the distributed name services.

The VMS 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 VMS logical name services are as follows:

- Create Logical Name ($CRELNM)
- Create Logical Name Table ($CRELNT)
- Delete Logical Name ($DELLNM)
- Translate Logical Name ($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 DIGITAL Distributed Naming Service (DECdns) 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.

6.1 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 7
- Common event flag cluster names: Chapter 4
- Global section names: Chapter 12
- Shared memory: Appendix B

The following sections describe various concepts you should be aware of when using the logical name system services. For further discussion of logical names, see the *VMS DCL Dictionary*.

6.1.1 Logical Names and Equivalence Names

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 remember. System managers and privileged users choose mnemonics for files, system devices, and search lists that are frequently accessed by all users.

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 is assigned an index number starting at 0.

Logical names and their equivalence strings are contained 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 ($CRELNM), Create Mailbox and Assign Channel ($CREMBX), or Mount Volume ($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.

6.1.2 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:

- Logical name directory tables
- Default logical name tables
- User-defined logical name tables

When the process is created, the logical name directory tables and the default logical name tables are created for each new process.

6.1.2.1 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 $CRELNT system service. In addition, the process directory table contains system-assigned logical name table names, the name of the process logical name table LNM$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 on privileges, see Section 6.1.3.

Logical names other than logical name table names may exist within these tables. The maximum length of 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.

6.1.2.2 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 maximum length of 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

Most entries in the process logical name table are made at user and supervisor mode. The following example shows how process-private logical names can be created in user mode by an image.

```
LOGDESC:
        .ASCID                  /ABC/
EQVNAM1:
        .ASCII                  /XYZ/
EQVLEN1=
        .-EQVNAM1
EQVNAM2:
        .ASCII                  /DEF/
EQVLEN2=
        .-EQVNAM2
TABDESC:
        .ASCID                  /LNM$PROCESS/
CRELST:
        .WORD                   EQVLEN1         ;Length of first equivalence name
        .WORD                   LNM$_STRING     ;Logical name string
        .ADDRESS                EQVNAM1         ;First equivalence name
        .LONG                   0
        .WORD                   EQVLEN2         ;Length of second equivalence name
        .WORD                   LNM$_STRING     ;Logical name string
        .ADDRESS                EQVNAM2         ;Second equivalence name
        .LONG                   0
        .LONG                   0
$CRELNM_S -
        LOGNAM = LOGDESC,-              ;Logical name
        TABNAM = TABDESC,-              ;Table name
        ITMLST = CRELST                 ;Equivalence strings
```

In the preceding example, a logical name ABC was created and represents 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 logical name creation at supervisor mode through DCL:

```
$  DEFINE/SUPERVISOR_MODE/TABLE=LNM$PROCESS ABC XYZ,DEF
```

Process logical names created in user mode are deleted whenever the creating process runs an image down. This behavior is illustrated by the following DCL commands.

```
$  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$LOGIN	Original default device and directory
SYS$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 need no privileges to modify the job logical name table. For a discussion on privileges, see Section 6.1.3.

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 on privileges, see Section 6.1.3.

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 on privileges, see Section 6.1.3.

6.1.2.3 User-Defined Logical Name Tables

You can create process-private tables and shareable tables by calling the $CRELNT system service in a program. However, you must have SYSPRV privilege to create a shareable table. For a discussion on privileges, see Section 6.1.3.

Processes other than the creating process cannot use logical names contained in process-private tables.

Logical name tables are created through the $CRELNT system service either with the DCL command CREATE/NAME_TABLE or by calling $CRELNT in a program. If granted access, processes other than the creating process can use shareable tables.

The maximum length of logical names created in user-defined logical name tables must not exceed 255 characters. Equivalence strings must not exceed 255 characters.

6.1.3 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 6–1.

Table 6–1 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.

6.1.4 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 $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 on access modes, see Section 2.1.3.

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 $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 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.

6.1.5 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$M_CONCEALED—Specifies that the equivalence string for the logical name is an RMS concealed device name.
- LNM$M_CONFINE—Prevents process-private logical names from being copied to subprocesses. Subprocesses are created by the DCL command SPAWN or by the LIB$SPAWN Run-Time Library procedure. This attribute is specified only in a $CRELNM or $CRELNT system service call.
- LNM$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, it is deleted.

  If specified in a $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 $CRELNM or $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 $CRELNT system service call.
- LNM$M_CASE_BLIND—Governs the translation process and causes $TRNLNM to ignore uppercase and lowercase differences in letters when searching for logical names. This attribute is specified only in a $TRNLNM system service call.
- LNM$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 $CRELNM system service call. When the item code LNM$_ATTRIBUTES is specified through $TRNLNM, the system returns the current attributes associated with the logical name and equivalence string at the current index value.

The following attributes may be returned:

- LNM$M_CONCEALED—Indicates that the equivalence string at the current index value for the logical name is a VMS RMS concealed device name.
- LNM$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 $CRELOG system service.
- LNM$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 be the same. For more information about attributes, refer to the appropriate system service in the *VMS System Services Reference Manual.*

6.1.6 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) and 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 may 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.

6.1.6.1 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.

6.1.6.2 Default Logical Name Table Quotas

The process, group, and system logical name tables have unlimited quotas.

6.1.6.3 Job Logical Name Table Quotas

Because the job logical name table is a shareable table, and you need no 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 job logical name table quota at the time of its creation:

- For all processes that activate LOGINOUT, the quota for the job logical name 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 AUTHORIZE/JTQUOTA= command.
- For all processes that do not activate LOGINOUT, the quota for the job logical name table may be specified as a quota list item PQL$_JTQUOTA in the call to the Create Process ($CREPRC) system service. If a detached process is to be created by means of the DCL command RUN/DETACHED, then the /JOB_TABLE_QUOTA qualifier is used to specify the $CREPRC quota list item.
- For all processes that do not activate LOGINOUT and do not specify a PQL$_JTQUOTA quota list item in their call to $CREPRC, the quota for the job logical name table is taken from the dynamic System Generation Utility (SYSGEN) parameter PQL$_DJTQUOTA. You may use SYSGEN to display both PQL$_DJTQUOTA and PQL$_MJTQUOTA, the default and minimum job logical name table quotas.

6.1.6.4 User-Defined Logical Name Table Quotas

User-defined logical name tables may 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.

6.1.7 Logical Name and Equivalence Name Format Conventions

The operating system uses special conventions for assigning logical names to equivalence names and 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 four-byte header that contains the following information.

Byte(s)	Contents
0	^X1B (Escape character)
1	^X00
2-3	VMS 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 then to use only the desired part of the equivalence string. The following program segment demonstrates how to do this.

```
ILST:
        .WORD     LNM$C_NAMLENGTH
        .WORD     LNM$_STRING
        .LONG     RESSTRING
        .LONG     RESDESC
        .LONG     0
TABDESC:
        .ASCID    /LNM$FILE_DEV/        ; Device/file table name

LOGDESC:                                ; Logical name to be translated
        .ASCID    /INPUT_DEVICE/
RESDESC:                                ; Descriptor for result string
        .LONG     LNM$C_NAMLENGTH       ; Size of result string
        .ADDRESS -                      ; Address of result string
                  RESSTRING
RESSTRING:                              ; Result string destination
        .BLKB     LNM$C_NAMLENGTH
    .
    .
    .

        $TRNLNM_S -                     ; Translate logical name
                  LOGNAM=LOGDESC, -
                  TABNAM=TABDESC, -
                  ITMLST=ILST
        BLBC      R0,ERR                ; Branch if error
        CMPW      RESSTRING, ^X001B     ; Is first character an escape?
        BNEQ      1$                    ; No, continue at 1$
        SUBW      #4,RESDESC            ; Yes, subtract 4 from length...
        ADDL      #4,RESDESC+4          ; and add 4 to address of string
1$:
    .
    .
    .
```

6.1.8 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 present a name that designates the containing logical name table. 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 that iteratively translates to the names of several logical name tables. A multivalued logical name is also known as a search list. The tables are used in the order in which they appear.

As mentioned earlier, 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 name of the logical name table used whenever file specifications or device names are translated by VMS RMS or the I/O services. This name must translate to a list of one or more logical name table names specifying 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 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.

6.2 Creating a Logical Name—$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 $CRELNM system service as shown in the following example. In either case, the result is the same: the logical name DISK is equated to the physical device name DUA2 in table LNM$JOB.

```
LOGDESC:
        .ASCID                  /DISK/
TABDESC:
        .ASCID                  /LNM$JOB/
LNMATTR:
        .LONG                   LNM$M_TERMINAL
CRELST:
        .WORD                   4
        .WORD                   LNM$_ATTRIBUTES
        .ADDRESS                LNMATTR
        .LONG                   0
        .WORD                   EQVLEN
        .WORD                   LNM$_STRING
        .ADDRESS                EQVNAM
        .LONG                   0
        .LONG                   0

EQVNAM:
        .ASCII                  /DUA2:/
EQVLEN=
        .-EQVNAM
   .
   .
   .
$CRELNM_S -
        LOGNAM  =  LOGDESC,-
        TABNAM  =  TABDESC,-
        ATTR    =  LNMATTR,-
        ITMLST  =  CRELST
```

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.

6.2.1 Duplication of 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.

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]	Exec
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 the purposes of this discussion) contains a definition of TERMINAL as the mailbox device MBA407. The multivalued logical name 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 table 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$SYSTEM_TABLE
```

In the preceding 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$JOB, and LNM$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 containing system files.

Logical Name		Equivalence Name
SYS$LIBRARY	——>	SYS$SYSROOT:[SYSLIB]
SYS$SYSTEM	——>	SYS$SYSROOT:[SYSEXE]
.		.
.		.
.		.

The Logical Names section of the *VMS DCL Dictionary* 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.

6.3 Creating Logical Name Tables—$CRELNT

The Create Logical Name Table ($CRELNT) system service creates logical name tables. Logical name tables can be created at any access mode depending on the privileges of the calling process. A user-specified logical name identifying the newly created logical name table is stored in the process directory table LNM$PROCESS_DIRECTORY.

6.3.1 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 $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.

6.3.2 $CRELNT System Service Call

The following example illustrates a call to the $CRELNT system service.

```
TABDESC:
        .ASCID                  /LOG_TABLE/
PARDESC:
        .ASCID                  /LNM$PROCESS_TABLE/
TAB_ATTR:
        .LONG                   LNM$M_CONFINE
TAB_QUOTA:
        .LONG                   5000
   .
   .
   .
$CRELNT_S -
        TABNAM = TABDESC,-          ;Table name
        PARTAB = PARDESC,-          ;Parent table
        ATTR = TAB_ATTR,-           ;Attributes
        QUOTA = TAB_QUOTA           ;Quota
```

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.

6.4 Deleting Logical Names—$DELLNM

The Delete Logical Name ($DELLNM) system service deletes entries from a logical name table. When you write a call to the $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.

```
LOGDESC:
        .ASCID          /TERMINAL/
TABDESC:
        .ASCID          /LNM$JOB/

$DELLNM_S  -
        LOGNAM = LOGDESC,-
        TABNAM = TABDESC,-
```

For information about access modes and the deletion of logical names, see Section 6.1.4.

6.5 Translating Logical Names—$TRNLNM

The Translate Logical Name ($TRNLNM) system service translates a logical name to its equivalence string. In addition, $TRNLNM returns information about the logical name and equivalence string.

The system service call to $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). These services implicitly specify the logical name table name LNM$FILE_DEV. For more information about LNM$FILE_DEV, refer to Section 6.1.8.

The following system services perform iterative logical name translation automatically:

- Allocate Device ($ALLOC)
- Assign I/O Channel ($ASSIGN)
- Broadcast ($BRDCST)
- Create Mailbox ($CREMBX)
- Deallocate Device ($DALLOC)
- Dismount Volume ($DISMOU)
- Get Device/Volume Information ($GETDVI)
- Mount Volume ($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 $TRNLNM system service searches the user-specified logical name tables for a specified logical name and returns the equivalence name. In addition, $TRNLNM returns attributes specified optionally for the logical name and equivalence string.

The following example shows a call to the $TRNLNM system service to translate the logical name ABC.

```
LOGDESC:
        .ASCID              /ABC/
TABDESC:
        .ASCID              /LNM$FILE_DEV/
EQVBUF1:
        .BLKB               LNM$C_NAMLENGTH
EQVDESC1:
        .LONG               0
        .ADDRESS            EQVBUF1
EQVBUF2:
        .BLKB               LNM$C_NAMLENGTH
EQVDESC2:
        .LONG               0
        .ADDRESS            EQVBUF2
TRNLIST:
        .WORD               LNM$C_NAMLENGTH
        .WORD               LNM$_STRING
        .ADDRESS            EQVBUF1
        .ADDRESS            EQVDESC1
        .WORD               LNM$C_NAMLENGTH
        .WORD               LNM$_STRING
        .ADDRESS            EQVBUF2
        .ADDRESS            EQVDESC2
        .LONG               0
TRNATTR:
        .LONG               LNM$M_CASE_BLIND
   .
   .
   .
$TRNLNM_S       -
        LOGNAM = LOGDESC,-
        TABNAM = TABDESC,-
        ATTR = TRNATTR,-
        ITMLST = TRNLIST
```

This call to the $TRNLNM system service results in the translation of the logical name ABC. In addition, LNM$FILE_DEV is specified in the **tabnam** argument as the search list that $TRNLNM is to use to find the logical name ABC. The logical name ABC was assigned two equivalence strings. The LNM$_STRING item code in the **itmlst** argument directs $TRNLNM to look for an equivalence string at the current index value. Note that the LNM$_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 $TRNLNM system service searches for the equivalence strings without regard to uppercase or lowercase letters. The $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.

6.6 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
```

6.7 The DECdns Clerk System Service

The DECdns Clerk ($DNS) system service supports two programming interfaces:

- The portable application programming interface
- The VMS system service and run-time library

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 the VMS and ULTRIX operating systems.

The portable interface is documented in the *Guide to Programming with DECdns*.

The VMS system service and run-time library routines can be used by applications written in the high-level and mid-level languages listed in the Preface of this document. However, applications that use these interfaces are limited to the VMS 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 VMS.
- The application is not written in the C programming language.

The $DNS system service is documented in the *VMS System Services Reference Manual*. Before using this system service, you must understand 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

Once you understand the preceding topics, you can continue 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
- Using the $DNS system service

6.7.1 Functions Provided by the DECdns System Service and Run-Time Library Routines

The $DNS system service and run-time library routines can be used together to assign, maintain, and retrieve DECdns names. This section describes the capabilities of each interface.

6.7.1.1 The $DNS System Service

DECdns provides a single system service call ($DNS) to create, delete, modify, and retrieve DECdns names from a namespace. The $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 successfully queued to the name service.

The $DNSW system service is the synchronous equivalent of $DNS. The $DNSW call is identical to $DNS in every way except that $DNSW returns to the caller after the operation completes.

The $DNS call has two main parameters:

- A function code identifying the particular service to perform
- An item list specifying 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

6.7.1.2 The Run-Time Library Routines

The DECdns run-time library routines can be used to manipulate output from the $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 doing the following:

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.

6.8 Using the $DNS System Service Call

The following sections describe how to create and modify an object; 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.

6.8.1 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 $DNS or the $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 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 created by other applications.

To create an object with $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, 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 $DNS parse function to convert the full name string into an opaque format. Specify the DNS$_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 containing 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, specifying 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 errors returned; it is not enough to check the return of the $DNS call itself. You need to check the DECdns status block to be sure no errors were returned by the DECdns server.

The following routine, written in C, shows how to create an object in the namespace with the synchronous service $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$w_itm_size = class_len;   ❶
    createitem[0].dns$w_itm_code = dns$_class;
    createitem[0].dns$a_itm_address = class_name;

    createitem[1].dns$w_itm_size = object_len;   ❷
    createitem[1].dns$w_itm_code = dns$_objectname;
    createitem[1].dns$a_itm_address = object_name;

    version.dns$b_c_major = 1;   ❸
    version.dns$b_c_minor = 0;

    createitem[2].dns$w_itm_size = sizeof(struct $dnscversdef);   ❹
    createitem[2].dns$w_itm_code = dns$_version;
    createitem[2].dns$a_itm_address = &version;

    *((int *)&createitem[3]) = 0;   ❺
```

```
    status = sys$dnsw(0, dns$_create_object, &createitem, &iosb, 0, 0); ❻
    if(status == SS$_NORMAL)
    {
        status = iosb.dns$l_dnsb_status; ❼
    }
    return(status);
}
```

❶ The first entry in the item list is the address of the opaque simple name representing the class of the object.

❷ The second entry in the item list is the address of the opaque full name for the object.

❸ 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.

❹ The third entry in the item list is the address of the version structure that was just built.

❺ A value of 0 terminates an item list.

❻ The next step is to call the system service to create the object.

❼ Then, check to see that both the system service and DECdns were able to perform the operation without error.

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 returned; it is not enough to check the return of the $DNS call itself. You need to check the DECdns status block to be sure no errors are returned by the DECdns server.

6.8.2 Modifying Objects and Their Attributes

After you create objects that identify resources, 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 time stamp 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 using 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 tasks to modify an object with $DNSW:

1. Build an item list containing the following elements:
   - The opaque name of the object you are modifying
   - The type of object
   - The operation to perform
   - The type of attribute you are modifying
   - The attribute name
   - The value being added to the attribute
2. Supply any of the optional parameters described in Section 6.8.1.
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;
{
    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;    ❶

    moditem[1].dns$w_itm_size = sizeof(char);
    moditem[1].dns$w_itm_code = dns$_lookingfor;
    moditem[1].dns$a_itm_address = &objtype;    ❷

    moditem[2].dns$w_itm_size = sizeof(char);
    moditem[2].dns$w_itm_code = dns$_modoperation;
    moditem[2].dns$a_itm_address = &opertype;   ❸

    moditem[3].dns$w_itm_size = sizeof(char);
    moditem[3].dns$w_itm_code = dns$_attributetype;
    moditem[3].dns$a_itm_address = &attype;     ❹

    moditem[4].dns$w_itm_size = att_len;
    moditem[4].dns$w_itm_code = dns$_attributename;
    moditem[4].dns$a_itm_address = att_name;    ❺

    moditem[5].dns$w_itm_size = val_len;
    moditem[5].dns$w_itm_code = dns$_modvalue;
    moditem[5].dns$a_itm_address = att_value;   ❻

    *((int *)&moditem[6]) = 0;                  ❼

    /*
     * Call $DNSW to add the attribute to the object.
     */
    status = sys$dnsw(0, dns$_modify_attribute, &moditem, &iosb, 0, 0);❽

    if(status == SS$_NORMAL)
    {
    status = iosb.dns$l_dnsb_status;            ❾
    }

    return(status);
}
```

❶ The first entry in the item list is the address of the opaque full name of the object.

❷ The second entry in the item list shows that this is an object—not a soft link or child directory pointer.

❸ The third entry in the item list is the operation to perform. The program adds an attribute with its value to the object.

❹ The fourth entry in the item list is the attribute type. The attribute has a set of values rather than a single value.

❺ The fifth entry in the item list is the opaque simple name of the attribute being added.

❻ The sixth entry in the item list is the value associated with the attribute.

❼ A value of 0 terminates the item list.

❽ Then, a call is made to the $DNSW system service to perform the operation.

❾ Finally, a check is made to see that both the system service and DECdns performed the operation without error.

6.8.3 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 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.

The VAX Distributed File Service (DFS) uses DECdns for resource naming. This section gives an example of the DNS$_READ_ATTRIBUTE call as used by DFS. The DFS application uses DECdns to give VMS users the ability to use remote VMS disks as if they were attached to their local VMS system. The DFS application creates DECdns names for VMS directory structures (a directory and all of its subdirectories). Each DFS object in the namespace references a particular file access point. DFS creates each object with a class attribute of DFS$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 VMS 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 procedures:

1. Translates the DECdns name that is supplied through the user to opaque format using the $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 $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 where the access point is located.
5. Verifies that the DECdns name is valid on the DFS server.

Then the DFS client and DFS server communicate to complete the mount function.

6.8.3.1 Reading Attributes

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 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 $DNSW service uses an item list to return a set of objects. Then, the application calls a run-time 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. */ ❶
   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  ❷
   {
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,
                          ❸                  &val_len, &cts_dsc,
                                               &cts_len, &newset_dsc,
                                               &setlen);

              if(set_status == SS$_NORMAL)
              {  ❹
                  readitem[4].dns$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: ");    ❺
                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);
}
```

❶ The item list contains five entries:

- The opaque full name of the object with the attribute the program wants to read
- The type of object to access
- The opaque simple name of the attribute to read
- The address of the buffer containing the set of values returned by the read operation
- A value of 0 to terminate the item list

❷ The loop repeatedly calls the $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.

❸ The DNS$REMOVE_FIRST_SET_VALUE routine extracts a value from the set.

❹ This attribute name might be the context the routine uses to read additional attributes. The attribute's creation timestamp (CTS), not its value, provides the context.

❺ 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.

6.8.3.2 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.

The following program, written in C, shows how an application can read the objects in a directory with the $DNS system service. The values 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 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);  ❶

    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,
              ❷    &enum_iosb, enumerate_objects_ast, setbuf);

    if(enum_status != SS$_NORMAL)   ❸
    {
        printf("Error enumerating objects = %d\n", enum_status);
        exit(enum_status);
    }
    status = sys$synch(synch_event, &enum_iosb);  ❹
```

```
    if(status != SS$_NORMAL)
    {
        printf("Synchronization with AST threads failed\n");
        exit(status);
    }
}

/* AST routine parameter:                          */
/*      outbuf : address of buffer that contains enumerated names. */

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$_NORMAL)
        {
            cvtitem[0].dns$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;
```

❺

❻

```
            *((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;    ❼
            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$_MOREDATA)
    {                                                    ❽
        enum_status = sys$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
    {                                                    ❾
        sys$setef(synch_event);
    }
}
```

❶ Get an event flag to synchronize the execution of AST threads.

❷ Use the system service to enumerate the object names.

❸ Check the status of system service itself before waiting for threads.

❹ Use the $SYNCH call to make sure the DECdns clerk has completed and that all threads have finished executing.

❺ After enumerating objects, $DNS calls an AST routine. The routine shows how DNS$REMOVE_FIRST_SET_VALUE extracts object names from the set returned by the DNS$_ENUMERATE_OBJECTS function.

❻ 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:

- The address of the opaque simple name to be converted
- The address of the buffer that will hold the string name

- A value of 0 to terminate the item list

❼ This object name could 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.

❽ Use the system service to enumerate the object names as long as there is more data.

❾ Set the event flag to indicate that all AST threads have completed and the program can terminate.

6.9 DECdns Logical Names

When the DECdns clerk is started on a VMS operating system, the VMS 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 shown in the previous section by entering the following DCL command:

```
$ DEFINE/TABLE=DNS$SYSTEM RESEARCH "ENG.RESEARCH"
```

When parsing a name, the $DNS service specifies the logical name DNS$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 $DNS, you must create a process or job table and redefine DNS$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 using the DCL command DEFINE, as follows:

```
$ DEFINE /TABLE=DNS_PROCESS_TABLE RESEARCH "ENG.RESEARCH.MYGROUP"
```

For information about logical names, see *VMS Introduction to System Services*.

7

Input/Output Services

You can use two basic methods to perform input/output operations under the VMS operating system:

- VMS Record Management Services (RMS)
- I/O system services

VMS RMS provides a set of routines for general-purpose, device-independent functions such as data storage, retrieval, and modification.

The 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 VMS RMS. Using I/O services requires more programming knowledge than using VMS RMS, but can result in more efficient input/output operations.

The following system services are Input/Output services:

- Device Scan ($DEVICE_SCAN)
- Assign I/O Channel ($ASSIGN)
- Deassign I/O Channel ($DASSGN)
- Queue I/O Request ($QIO)
- Queue I/O Request and Wait for Event Flag ($QIOW)
- Formatted ASCII Output ($FAO)
- Formatted ASCII Output with List Parameter ($FAOL)
- Allocate Device ($ALLOC)
- Deallocate Device ($DALLOC)
- Mount Volume ($MOUNT)
- Dismount Volume ($DISMOU)
- Initialize Volume ($INIT_VOL)
- Get Device and Channel Information ($GETDVI)
- Get Device and Channel Information and Wait ($GETDVIW)
- Cancel I/O on Channel ($CANCEL)
- Create Mailbox and Assign Channel ($CREMBX)
- Delete Mailbox ($DELMBX)
- Breakthrough ($BRKTH)
- Breakthrough and Wait ($BRKTHW)
- Get Queue Information ($GETQUI)

- Get Queue Information and Wait ($GETQUIW)
- Send Message to Job Controller ($SNDJBC)
- Send Message to Job Controller and Wait ($SNDJBCW)
- Send Message to Operator ($SNDOPR)
- Send Message to Error Logger ($SNDERR)
- Get Message ($GETMSG)
- Put Message ($PUTMSG)
- Get Job/Process Information ($GETJPI)
- Get Job/Process Information and Wait ($GETJPIW)
- Get Lock Information ($GETLKI)
- Get Lock Information and Wait ($GETLKIW)
- Get Systemwide Information ($GETSYI)
- Get Systemwide Information and Wait ($GETSYIW)
- Update Section File on Disk ($UPDSEC)
- Scan for Devices ($DEVICE_SCAN)

This chapter includes the following general information about how to use the I/O services:

- Assigning channels
- Queuing I/O requests
- Allocating devices
- Using mailboxes

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 *VMS I/O User's Reference Volume.*

If you want to write your own device driver or connect to a device interrupt vector, see the *VMS Device Support Manual.*

7.1 Quotas, Privileges, and Protection

To preserve the integrity of the operating system, VMS I/O operations are performed under the constraints of quotas, privileges, and protection.

Quotas establish a limit on the number and type of I/O operations that a process can perform concurrently, and on 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 ($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.

7.1.1 Buffered I/O Quota

The buffered I/O quota specifies the maximum number of concurrent buffered I/O operations a process can have active. 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 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.

7.1.2 Buffered I/O Byte Count Quota

The buffered I/O byte count quota 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 Set Resource Wait Mode 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.

7.1.3 Direct I/O Quota

The direct I/O quota 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 DMA real-time devices, and nonnetwork transfers, for example, 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 Set Resource Wait Mode 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.

7.1.4 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.

7.1.5 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. Figure 7–4 and Figure 7–5 show the use of physical I/O privilege in greater detail.

7.1.6 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 VMS software.) Figure 7–4 and Figure 7–5 show the use of logical I/O privilege in greater detail.

7.1.7 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 ACP interface operations.

7.1.8 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 $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 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 ancillary control process [ACP]), volumes that are mounted as foreign, and mailboxes (both temporary and permanent).

Figure 7–1 shows the 4-bit protection fields for volumes mounted as structured. Figure 7–2 shows the 4-bit protection fields for foreign volumes. Figure 7–3 shows the 4-bit protection fields for mailboxes.

Usually, volume protection is meaningful only for read and write operations.

Figure 7–1 Files–11 Volume Protection Fields

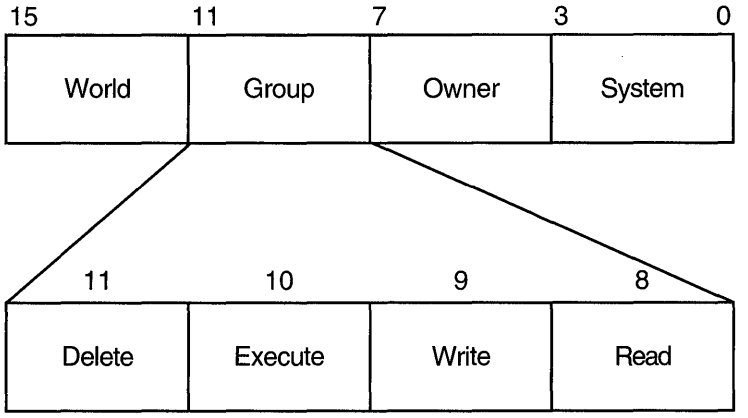
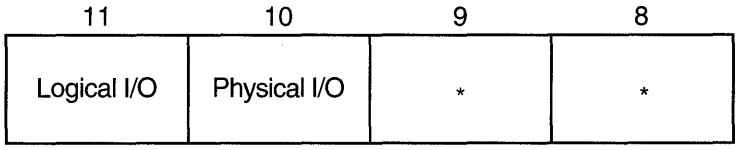
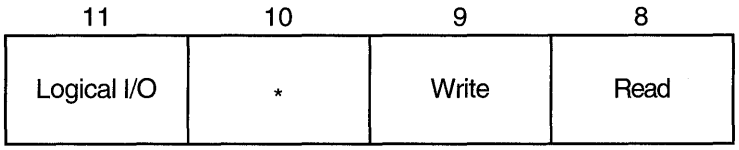


Figure 7–2 Foreign Volume Protection Fields



*Not Used ZK–0623–GE

Figure 7–3 Mailbox Protection Fields



*Not Used ZK–0624–GE

7.1.9 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 if 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.

7.1.10 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.

7.1.11 Bypass Privilege

Bypass privilege (BYPASS) allows a process to bypass volume and device protection completely.

7.2 Summary of VMS QIO Operations

The VMS 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.

7.3 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 may be necessary if the device is under ACP control. (ACP functions are described in the *VMS I/O User's Reference Volume*.)

7.3.1 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 $QIO system service, which returns a condition value of SS$_NOPRIV (no privilege). Figure 7–4 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.

7.3.2 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 $QIO system service, which returns a condition value of SS$_NOPRIV (no privilege). Figure 7–5 illustrates the logical I/O access checks in greater detail.

7.3.3 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.

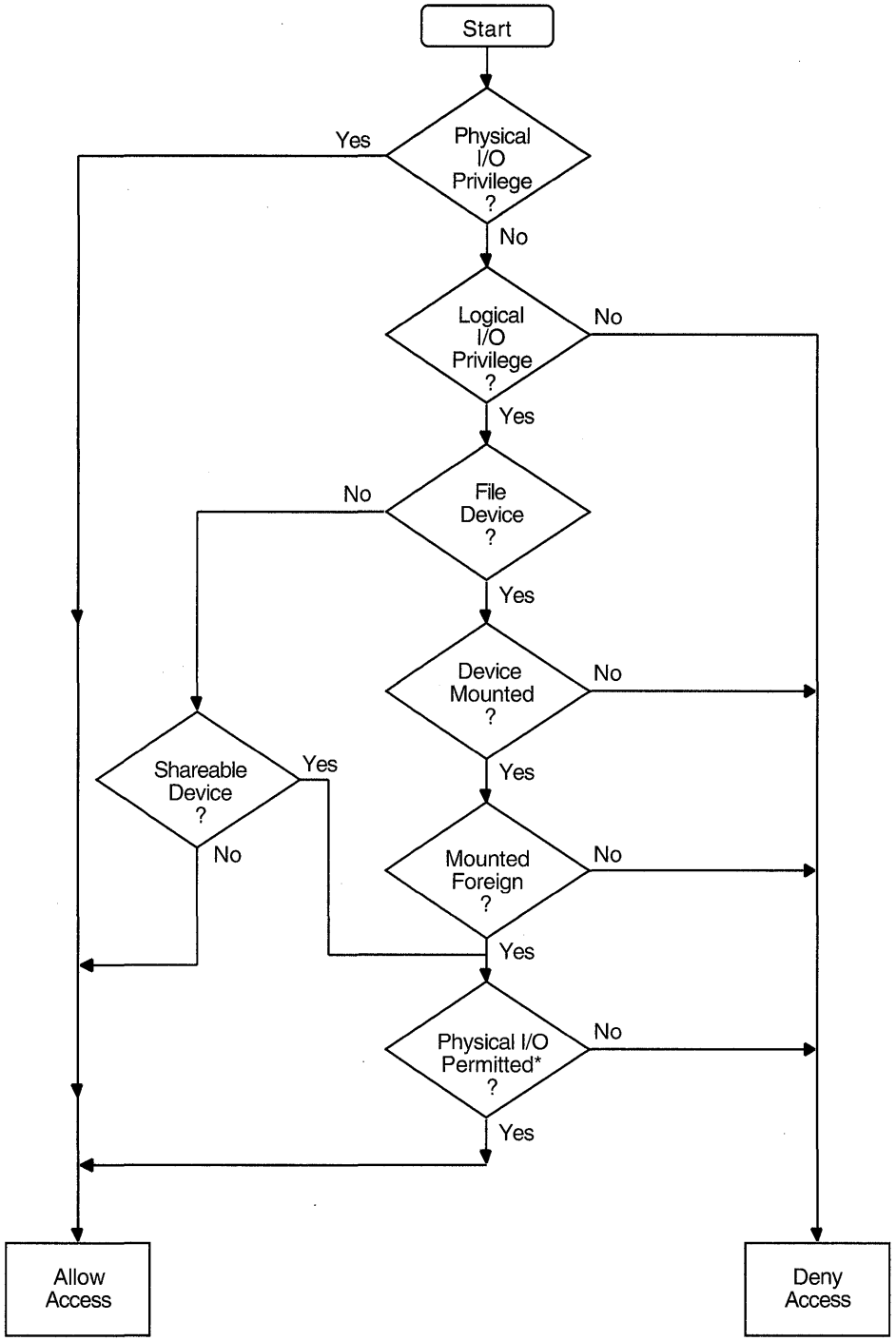
For block-addressable devices (such as disks), data is read from and written to open files. The addressable units in the file are 512-byte blocks. They are numbered starting at 1 and are relative to a file rather than to a device. Block-addressable devices must be mounted and structured and must contain 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 7–6 shows the relationship of physical, logical, and virtual I/O to the driver.

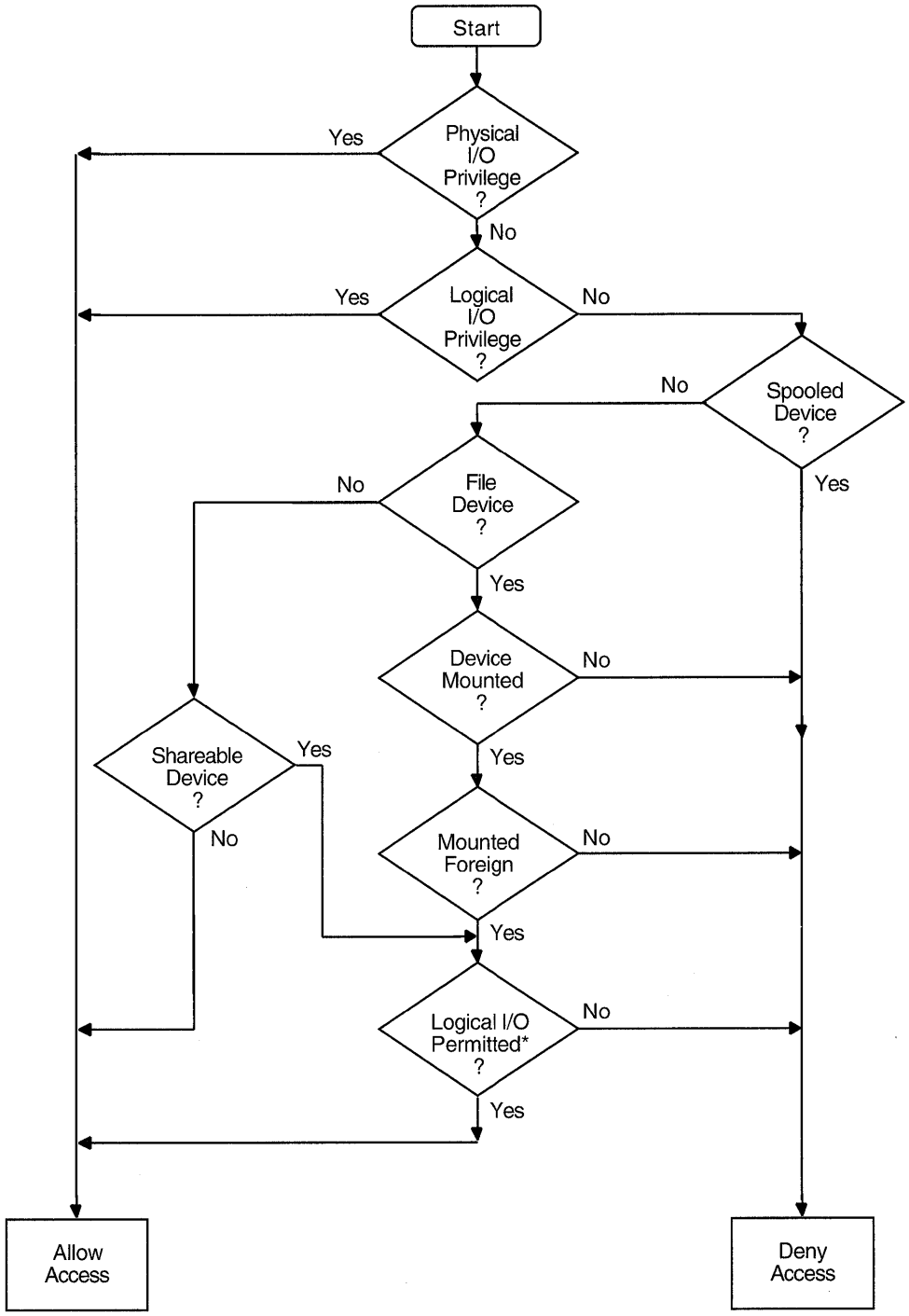
Figure 7–4 Physical I/O Access Checks



*Volume protection mask allows access.

ZK–0625–GE

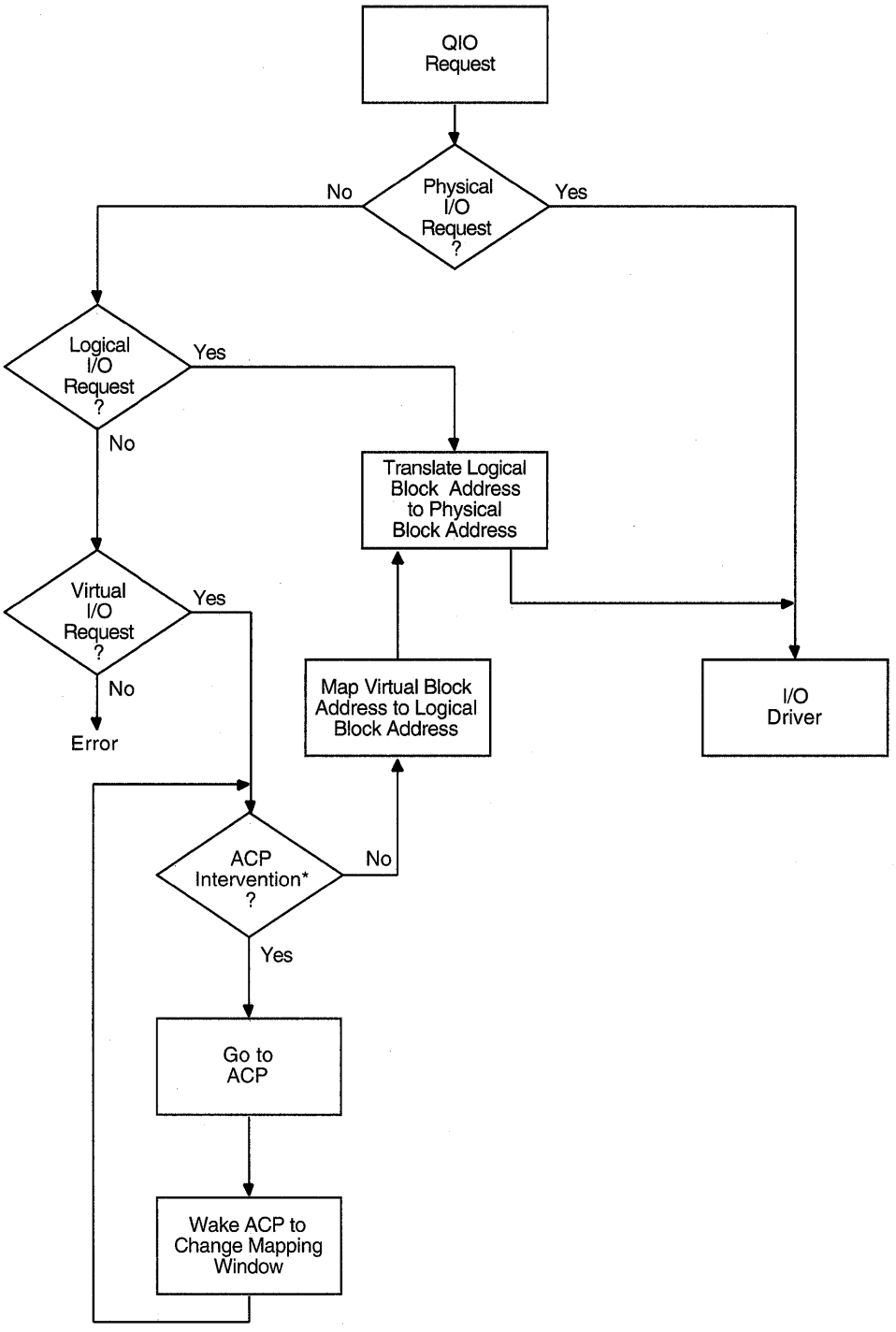
Figure 7–5 Logical I/O Access Checks



* Volume protection mask allows access.

ZK–0626–GE

Figure 7–6 Physical, Logical, and Virtual I/O



*Needed to map virtual address to logical address.

ZK–0627–GE

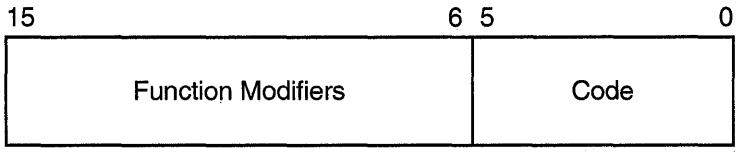
7.4 I/O Function Encoding

I/O functions fall into three groups that correspond to the three I/O device addressing modes (physical, logical, and virtual) described in Section 7.3. 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 7–7 shows the format of the 16-bit function value.

Symbolic names for I/O function codes are defined by the $IODEF macro.

Figure 7–7 I/O Function Format



ZK–0628–GE

7.4.1 Function Codes

The low-order six 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 7–1 lists the symbolic values for read and write I/O functions in the three transfer modes.

Table 7–1 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 *VMS I/O User's Reference Volume* provides complete descriptions of the functions and function codes.

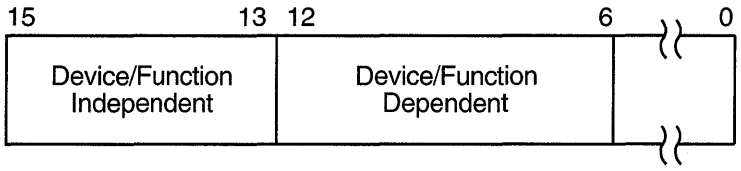
Note

You should determine the device class before performing any QIO function, because the requested function may 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.

7.4.2 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 7–8 shows the format of function modifiers.

Figure 7–8 Function Modifier Format



ZK–0629–GE

As shown in Figure 7–8, bits 13 through 15 are device- or function-independent bits, and bits 6 through 12 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$M_INHRETRY is used to inhibit all error recovery. If any error occurs, and this modifier bit is specified, the operation is terminated immediately and a failure status is returned in the I/O status block (see Section 7.10). IO$M_DATACHECK is used to compare the data in memory with that on a disk or magnetic tape.

7.5 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 ($ASSIGN) system service establishes this path.

When you write a call to the $ASSIGN service, you must supply the name of the device, which may 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.

```
TTNAME: .ASCID  /TTA2:/                 ; Terminal descriptor
TTCHAN: .BLKW   1                       ; Terminal channel number
          .
          .
          .
        $ASSIGN_S -
                DEVNAM=TTNAME, -
                CHAN=TTCHAN
```

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 $ASSIGN and other I/O services handle logical names, see Section 7.1.5.

7.6 Queuing I/O Requests

All input and output operations in VMS are initiated with the Queue I/O Request ($QIO) system service. The $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.

Required arguments to the $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 $QIO to write a line to a terminal may look like the following.

```
$QIO_S   CHAN=TTCHAN, -
         FUNC=#IO$_WRITEVBLK, -
         P1=BUFADDR, -
         P2=#BUFLEN
```

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 Section 2.3. For details on all function codes and an explanation of the parameters required by each, see the *VMS I/O User's Reference Volume*.

7.7 Synchronizing Service Completion

The $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 that the operation was queued successfully.
- Test whether the operation itself completed successfully.

Optional arguments to the $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 7.10.

The use of these three techniques is shown in the three examples that follow.

Example 1: Event Flags

```
$QIO_S  EFN=#1,...❶          ; Issue 1st I/O request
BLBC    R0,ERROR                  ; Queued successfully?
$QIO_S  EFN=#2,...❷          ; Issue 2nd I/O request
BLBC    R0,ERROR                  ; Queued successfully?
$WFLAND_S -  ❸               ; Wait till both done
        EFN=#0, - ❹
        MASK=#^B110
```

❶ When you specify an event flag number as an argument, $QIO clears the event flag when it queues the I/O request. When the I/O completes, the flag is set.

❷ In this example, the program issues two Queue I/O requests. A different event flag is specified for each request.

❸ The Wait for Logical AND of Event Flags ($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.

❹ Note that the $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 7.8 for a discussion of the recommended technique for testing I/O completion.)

Example 2: An AST Routine

```
$QIO_S  ...,ASTADR=TTAST, - ❶; I/O with AST
            ASTPRM=#1,...
BLBC        R0,ERROR              ; Queued successfully?
  .         ; Continue
  .
  .
.ENTRY      TTAST,^M<R10,R11> ❷; AST service routine entry mask
  .                             ; handle I/O completion
  .
  .
RET                             ; End of service routine
```

❶ When you specify the **astadr** argument to the $QIO system service, the system interrupts the process when the I/O completes and passes control to the specified AST service routine.

The $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.

❷ 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 $QIO, you should also specify the **iosb** argument so that the AST routine can evaluate whether the I/O completed successfully.

Example 3: The I/O Status Block ❶

```
TTIOSB: .BLKQ   1 ❷                    ; I/O status block
          .
          .
          .
❸    $QIO_S  ...,IOSB=TTIOSB,...       ; Issue I/O request
       BLBC    R0,ERROR                     ; Queued successfully?
          .                               ; Continue
          .
          .
10$:    TSTW    TTIOSB  ❹                  ; Is I/O done yet?
        BEQL    10$                       ; No, loop till done
  ❺       CMPW    TTIOSB,#SS$_NORMAL      ; I/O successful?
        BNEQ    IO_ERR                    ; No, handle the error
          .
          .
```

❶ 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.

❷ TTIOSB defines the I/O status block for this I/O operation. The **iosb** argument in the $QIO system service refers to this quadword.

❸ The $QIO system service clears the quadword when it queues the I/O request. When the request is queued, the program calls a routine to check whether the request was successfully placed on the queue; if queuing was successful, the program continues execution.

❹ The process polls the I/O status block. If the low-order word still contains 0, the I/O operation has not yet completed. In this example, the program loops until the request is complete.

❺ After the I/O operation completes, the process compares the low word of the I/O status block with the success status SS$_NORMAL. If the return status is not SS$_NORMAL, the program branches to IO_ERR.

Note

The technique shown in Example 3 wastes system time, looping until the request is complete; you should use this technique only when it is the last possible alternative.

7.8 Recommended Method for Testing Asynchronous Completion

Digital recommends that you use the Synchronize ($SYNCH) system service to wait for completion of an asynchronous event. The $SYNCH service correctly waits for the actual completion of an asynchronous event, even if some other event sets the event flag.

To use the $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 $SYNCH, you must specify the same **efn** and I/O status block that you specified in your call to the asynchronous service. The $SYNCH service waits for the event flag to be set by means of the $WAITFR system service. When the specified event flag is set, $SYNCH checks the specified I/O status block. If the I/O status block is nonzero, the system service has completed and $SYNCH returns control to your program. If the I/O status block is 0, $SYNCH clears the event flag by means of the $CLREF service and calls the $WAITFR service to wait for the event flag to be set.

The $SYNCH service sets the event flag before returning control to your program. This ensures that the call to $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 ($QIO) system service demonstrates how the $SYNCH service is used.

```
EVENT_FLAG = 1
;
Q_IOSB: .QUAD   0
          .
          .
          .
        $QIO_S  EFN=#EVENT_FLAG, -      ; Request I/O
                IOSB=Q_IOSB,...
        $SYNCH_S -
                EFN=#EVENT_FLAG
                IOSB=Q_IOSB             ; Wait until I/O completes
        BLBC    R0,ERROR                ; Test status
          .
          .
          .
```

Note

The $QIOW service provides a combination of $QIO and $SYNCH. This program segment provides only an example of how $SYNCH operates. For a more complete example, see Section 2.5.1.

7.9 Synchronous 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 ($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 $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.

The asynchronous and synchronous names of input/output services that have synchronous versions are as follows.

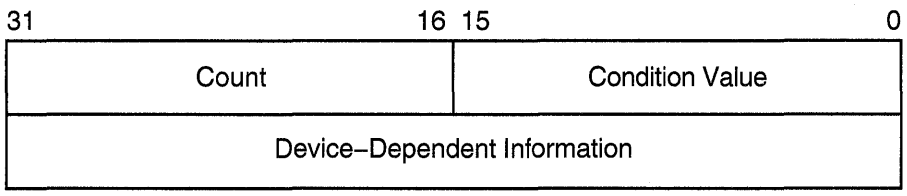
Asynchronous Name	Synchronous Name	Description
$BRKTHRU	$BRKTHRUW	Breakthrough
$GETDVI	$GETDVIW	Get Device/Volume Information
$GETJPI	$GETJPIW	Get Job/Process Information
$GETLKI	$GETLKIW	Get Lock Information
$GETQUI	$GETQUIW	Get Queue Information
$GETSYI	$GETSYIW	Get Systemwide Information
$QIO	$QIOW	Queue I/O Request
$SNDJBC	$SNDJBCW	Send to Job Controller
$UPDSEC	$UPDSECW	Update Section File on Disk

7.10 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 7–9 illustrates the format for the $QIO system service of the information written in the IOSB.

Figure 7–9 I/O Status Block



ZK–0856–GE

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.

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 *VMS I/O User's Reference Volume*.

The second longword contains device-dependent return information.

System services other than $QIO use the quadword I/O status block, but the format is different. See the description of each system service in the *VMS 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 on how to use the I/O status block, see the *VMS I/O User's Reference Volume*.

7.11 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 ($DASSGN) system service:

```
$DASSGN_S CHAN=TTCHAN
```

This service call releases the terminal channel assignment acquired in the $ASSIGN example shown in Section 7.5. The system automatically deassigns channels for a process when the image that assigned the channel exits.

7.12 Example of 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.

```
TTNAME: .ASCID  /SYS$INPUT:/ (1)        ; Descriptor for terminal name
TTCHAN: .BLKW   1                       ; Receive channel number here
TTIOSB: .BLKW   1   (2)                 ; First word of IOSB, status
TTIOLEN:
        .BLKW   1                       ; Second word, get length
        .BLKL   1                       ; Second longword of IOSB
OUTLEN: .BLKL   1   (3)                 ; Length of string to output
BUFFER: .BLKB   80                      ; Buffer to read input
        LENGTH=.-BUFFER
          .
          .
          .
(4)$ASSIGN_S -                      ; Assign channel
                DEVNAM=TTNAME, -        ; Logical name translated by $ASSIGN
                CHAN=TTCHAN
        BSBW    ERROR
(5)     $QIOW_S -
                FUNC=#IO$_READVBLK-
                CHAN=TTCHAN, -
                P2=#LENGTH, -
                P1=BUFFER, -
                IOSB=TTIOSB
        BSBW    ERROR
        MOVZWL  TTIOSB,R0               ; Move status code to R0
(6)     BSBW    ERROR
(7)     MOVZWL  TTIOLEN,OUTLEN          ; Get length out of IOSB
        $QIOW_S -
                FUNC=#IO$_WRITVBLK-
                CHAN=TTCHAN, -
                P2=OUTLEN, -
                P1=BUFFER, -
                IOSB=TTIOSB
```

```
❽  BSBW    ERROR
     MOVZWL  TTIOSB,R0              ; Move status code to R0
     BSBW    ERROR
❾  $DASSGN_S -                     ; Done, deassign channel
             CHAN=TTCHAN
     BSBW    ERROR
        .
        .
        .
ERROR:  BLBS    R0,10$             ; Check for successful
                                   ; return code
                                   ;
        $EXIT_S R0                 ; If not successful,
                                   ; exit and signal
                                   ;
10$:    RSB                        ; If successful,
                                   ; return to caller
```

❶ 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.

❷ 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).

❸ The string will be read into the buffer INBUF; the longword OUTLEN will contain the length of the string for the output operation.

❹ The $ASSIGN service assigns a channel and writes the channel number at TTCHAN.

❺ 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.

❻ 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.

❼ 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 a word long. The $QIOW macro writes the line just read to the terminal.

❽ 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.

❾ When all I/O operations on the channel are finished, the channel is deassigned.

7.13 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 ($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 $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 prior to checking the status for that operation.

For example, you can call the $CANCEL system service as follows.

```
$QIO_S      CHAN=TTCHAN, EFN=3, IOSB=IOSB1  . . .
$QIO_S      CHAN=TTCHAN, EFN=4, IOSB=IOSB2  . . .
.
.
.
$CANCEL_S   CHAN=TTCHAN
$SYNCH      EFN=3, IOSB=IOSB1
$SYNCH      EFN=4, IOSB=IOSB2
```

In this example, the $CANCEL system service initiates the cancellation of all pending I/O requests to the channel whose number is located at TTCHAN.

The $CANCEL system service returns after initiating the cancellation of the I/O requests. If the call to $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 actually completed.

7.14 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 ($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 ($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 $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, $ALLOC attempts to allocate the specified device.

If you specify a logical name, $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), $ALLOC attempts to allocate any device available of the specified type. For more information about the allocation of devices by generic names, see Section 7.17.

When you specify generic device names, you must provide fields for the $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 $ASSIGN system service.

The following example illustrates the allocation of a tape device specified by the logical name TAPE.

```
LOGDEV: .ASCID  /TAPE/                  ; Descriptor for logical name
DEVDESC:                                ; Descriptor for physical name
        .LONG   64                      ; Length of buffer
        .ADDRESS -                      ; Address of buffer
                 DEVSTR
DEVSTR: .BLKB   64                      ; Get physical name returned
TAPECHAN:
        .BLKW   1                       ; Channel for tape I/O
          .
          .
❶       $ALLOC_S -
                 DEVNAM=LOGDEV, -
                 PHYLEN=DEVDESC, -
                 PHYBUF=DEVDESC
          BSBW   ERROR
❷       $ASSIGN_S -                     ; Assign channel
                 DEVNAM=DEVDESC, -
                 CHAN=TAPECHAN
          BSBW   ERROR
            .                           ; Continue with I/O
            .
❸       $DASSGN_S -                     ; Deassign channel
                 CHAN=TAPECHAN
          BSBW   ERROR
          $DALLOC_S -                   ; Deallocate tape
                 DEVNAM=DEVDESC
```

❶ The $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 actually allocated and the length of the name string. The $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.

❷ The $ASSIGN command uses the character string returned by the $ALLOC system service as the input device name argument, and requests that the channel number be written into TAPECHAN.

❸ When I/O operations are completed, the $DASSGN system service deassigns the channel, and the $DALLOC system service deallocates the device. The channel must be deassigned before the device can be deallocated.

7.14.1 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 $ASSIGN system service when $ASSIGN is called to assign a channel to the device.

7.14.2 Deallocation

When the program has finished using an allocated device, it should release the device with the Deallocate Device ($DALLOC) system service, to make it available for other processes, as in this example:

```
$DALLOC_S DEVNAM=DEVDESC
```

At image exit, the system automatically deallocates devices allocated by the image.

7.15 Mounting, Dismounting, and Initializing Volumes

This section introduces you to using system services to mount, dismount, and initialize disk and tape volumes.

7.15.1 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 a volume or volume set programmatically using the Mount Volume ($MOUNT) system service or the Dismount Volume ($DISMOU) system service.

Mounting a volume involves two operations:

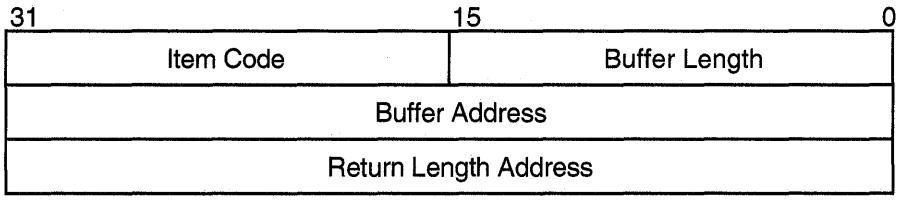
1. Place the volume on the device and start the device (by pressing the START or LOAD button).
2. Mount the volume with the $MOUNT system service.

7.15.1.1 Calling the $MOUNT System Service

The Mount Volume ($MOUNT) system service allows a process to mount a single volume or a volume set. When you call the $MOUNT system service, you must specify a device name.

The $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 7–10 shows the format of an item descriptor.

Figure 7–10 $MOUNT Item Descriptor



ZK–1705–GE

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 $MOUNT. The call is equivalent to the DCL command that precedes the example.

```
$ MOUNT/SYSTEM/NOQUOTA  DRA4:,DRA5:  USER01,USER02  USERD$

        $MNTDEF
ITEMS:  .WORD   4                   ; Length of flags
        .WORD   MNT$_FLAGS          ; Flag code
        .ADDRESS -                  ; Address of flags longword
                FLAGS
        .LONG   0                   ; Unused longword
;
        .WORD   5                   ; Length of first device name
        .WORD   MNT$_DEVNAM         ; Device code
        .ADDRESS -                  ; Address of first device name
                DEV1
        .LONG   0                   ; Unused longword
;
        .WORD   6                   ; Length of first volume name
        .WORD   MNT$_VOLNAM         ; Volume code
        .ADDRESS -                  ; Address of first volume name
                VOL1
        .LONG   0                   ; Unused longword
;
        .WORD   5                   ; Length of second device name
        .WORD   MNT$_DEVNAM         ; Device code
        .ADDRESS -                  ; Address of second device name
                DEV2
        .LONG   0                   ; Unused longword
;
        .WORD   6                   ; Length of second volume name
        .WORD   MNT$_VOLNAM         ; Volume code
        .ADDRESS -                  ; Address of second volume name
                VOL2
        .LONG   0                   ; Unused longword
;
        .WORD   6                   ; Length of volume logical name
        .WORD   MNT$_LOGNAM         ; Logical name code
        .ADDRESS -                  ; Address of volume logical name
                LOG
        .LONG   0                   ; Unused longword
        .LONG   0                   ; End of item list
;
DEV1:   .ASCII  /DRA4:/             ; First device
VOL1:   .ASCII  /USER01/            ; First volume name
DEV2:   .ASCII  /DRA5:/             ; Second device
VOL2:   .ASCII  /USER02/            ; Second volume name
LOG:    .ASCII  /USERD$/            ; Logical name
;
FLAGS:  .LONG   <MNT$M_SYSTEM!MNT$M_NODISKQ>
          .
          .
          .
        $MOUNT_S -                  ; Now call $MOUNT
                ITMLST=ITEMS
          .
          .
          .
```

7.15.1.2 Calling the $DISMOU System Service

The $DISMOU system service allows a process to dismount a volume or volume set. When you call $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 $DISMOU. The call dismounts the volume set mounted in the previous example.

```
DEV1_DESC:
        .ASCID   /DRA4:/
          .
          .
          .
        $DISMOU_S -
                DEVNAM=DEV1_DESC
          .
          .
          .
```

7.15.2 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 ($INIT_VOL) system service.

7.15.2.1 Calling the Initialize Volume System Service

You must specify a device name and a new volume name when you call the $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 *VMS System Services Reference Manual* for more information on initialization options.

Before initializing the volume with $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 is used by other Digital operating systems, including RSX–11M, RSX–11M–PLUS, RSX–11D, and IAS. For more information, see the *Guide to VMS Files and Devices*.

Here are two examples of calling $INIT_VOL programmatically: one from a C program and one from a BASIC program.

Examples

The following example illustrates a call to $INIT_VOL from VAX C.

1.
```
#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 $INIT_VOL from VAX BASIC.

2.
```
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)
```

7.16 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:

```
TTNAME: .ASCID  /_TTB3:/
```

Any string that does not begin with an underscore is considered a logical name, even though it may be a physical device name. The following system services translate a logical name iteratively until a physical device name is returned, or until the system default number of translations have been performed:

- Allocate Device ($ALLOC)
- Assign I/O Channel ($ASSIGN)
- Broadcast ($BRDCST)
- Deallocate Device ($DALLOC)
- Dismount Volume ($DISMOU)
- Get I/O Device Information ($GETDEV)
- Get Device/Volume Information ($GETDVI)
- Mount Volume ($MOUNT)

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 $ALLOC system service returns the device name of the physical device that has been allocated, the device name string returned is prefaced with an underscore character. When this name is used for the subsequent $ASSIGN system service, the $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 ($CRELNM) system service.

For details on how to create and use logical names, see Chapter 6.

7.17 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 $ASSIGN and $DALLOC system services apply default values as shown in Table 7–2.
- The $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 7–2.

Table 7–2 Default Device Names for I/O Services

Device	Device Name[1]	Generic Device
dd:	ddA0: (unit 0 on controller A)	ddxy: (any available device of the specified type)
ddc:	ddc0: (unit 0 on controller specified)	ddcy: (any available unit on the specified controller)

[1]A summary of the device names is contained in the *VMS DCL Concepts Manual.*

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

(continued on next page)

Table 7–2 (Cont.) Default Device Names for I/O Services

Device	Device Name[1]	Generic Device
ddu:	ddAu: (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]A summary of the device names is contained in the *VMS DCL Concepts Manual.*

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

7.18 Obtaining Information About Physical Devices

The Get Device/Volume Information ($GETDVI) system service returns information about devices. The information returned is specified by an item list created before the call to $GETDVI.

When you call the $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 $GETDVI in the *VMS System Services Reference Manual.* The *VMS I/O User's Reference Volume* 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 actually been used. To do this, the program should provide $GETDVI with the number of the channel to the device and request the name of the device with the DVI$_DEVNAM item identifier.

VMS 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$_ENDOFFILE).
- A write to NL immediately returns a success indication (SS$_NORMAL).

The null device functions as a virtual device to which you can direct output, but from which the data does not return.

7.19 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 ($FAO) system service performs these functions.

Input to the $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 $FAO system service to format an output string for a $QIOW macro. Complete details on how to use $FAO, with additional examples, are provided in the description of the $FAO system service in the *VMS System Services Reference Manual*.

```
FAOSTR:❶ .ASCID  /FILE !AS DOES NOT EXIST/ ; Descriptor for
                                ; FAO control string
FAODESC: ❷                    ; Descriptor for $FAO output
        .LONG   80              ; Length of buffer
        .ADDRESS -              ; Address of buffer
                 FAOBUF
FAOBUF: .BLKB   80              ; Buffer for $FAO output

FAOLEN: .LONG   0               ; Receive length of $FAO output

FILESPEC: ❸
        .ASCID  /DISK$USER:MYFILE.DAT/ ; Descriptor for FAO parameter
          .
          .
          .
❹   $FAO_S  CTRSTR=FAOSTR, -
              OUTLEN=FAOLEN, -
              OUTBUF=FAODESC,-
              P1=#FILESPEC    ; Parameter for $FAO
        BSBW    ERROR
❺   $QIOW   ...,BUFFER=FAOBUF, -
              LENGTH=FAOLEN
        BSBW    ERROR
```

❶ 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 $FAO is called to format this control string, !AS will be substituted with the string whose descriptor address is specified.

❷ FAODESC is a character string descriptor for the output buffer; $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 $QIOW macro.)

❸ FILESPEC is a character string descriptor defining an input string for the FAO directive !AS.

❹ The call to $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.

❺ When $FAO completes successfully, $QIOW writes the following output string:

```
FILE DISK$USER:MYFILE.DAT DOES NOT EXIST
```

7.20 Mailboxes

Mailboxes are virtual devices that can be used for communication among processes. You accomplish actual data transfer by using VMS RMS or I/O services. When the Create Mailbox and Assign Channel ($CREMBX) 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 $CREMBX or $ASSIGN system service.

The $CREMBX system service creates the mailbox. The $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 $CREMBX or $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 $TRNLNM system service), or it can call the Get Device/Volume Information ($GETDVI) system service to obtain the unit number and the physical device name.

Channels assigned to mailboxes can be either bidirectional or unidirectional. Bidirectional channels (read/write) allow both $QIO read and $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 ($CREMBX) and Assign I/O channel ($ASSIGN) system services use the **flags** argument to enable unidirectional channels. If the **flags** argument is not specified, or is 0, 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 *VMS I/O User's Reference Manual: Part I*.

Mailboxes are either temporary or permanent. You need the user privileges TMPMBX and PRMMBX to create temporary and permanent mailboxes.

For a temporary mailbox, the $CREMBX service enters the logical name and equivalence name in the logical name table LNM$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 $CREMBX service enters the logical name and equivalence name in the logical name table LNM$PERMANENT_MAILBOX. This logical name table name usually specifies the LNM$SYSTEM logical name table name. Permanent mailboxes continue to exist until they are specifically marked for deletion with the Delete Mailbox ($DELMBX) system service.

The following example shows how processes can communicate by means of a mailbox.

```
Process ORION

MBLOGNAM:                                   ; Mailbox logical
        .ASCID  /GROUP100_MAILBOX/          ; Name descriptor
MBUFLEN = 128
MBUFFER:
        .BLKB   MBUFLEN                     ; Input buffer for mailbox reads
MBXCHAN:
        .BLKW   1                           ; Mailbox channel number

MBXIOSB:
        .BLKW   1                           ; IOSB first word (status)
MBLEN:  .BLKW   1                           ; IOSB 2nd word (length)
        .BLKL   1                           ; Remainder of IOSB

OUTLEN: .BLKL   1                           ; Longword to get length

          .
          .
         .ENTRY ORION, ^M<R2,R3,R4>         ; Entry mask
;
❶    $CREMBX_S -
                PRMFLG= #0, -
                CHAN=MBXCHAN, -
                MAXMSG=#MBUFLEN, -
                BUFQUO= #384, -
                PROMSK= # ^X0000, -
                LOGNAM=MBLOGNAM
        BSBW    ERROR
;
❷    $QIO_S CHAN=MBXCHAN, -
                FUNC= #IO$_READVBLK, -
                IOSB=MBXIOSB,-
                ASTADR=MBXAST, -
                P1=MBUFFER, -
                P2=#MBUFLEN
        BSBW    ERROR
           .
           .
        RET
;
❸    .ENTRY  MBXAST, ^M<R2,R3,R4>    ; AST routine entry mask
;
        CMPW    MBXIOSB, #SS$_NORMAL    ; I/O successful?
        BNEQ    ASTERR                  ; Branch if not
        MOVZWL  MBLEN,OUTLEN            ; Make length a longword
        $QIOW_S ...,BUFFER=MBUFFER, -
                LENGTH=OUTLEN,...
        BSBW    ERROR
           .
           .
        RET

Process CYGNUS

MAILBOX:                                    ; Mailbox logical name descriptor
        .ASCID /GROUP100_MAILBOX/
MAILCHAN:                                   ; Mailbox channel number
        .BLKW   1
OUTBUF: .BLKB   128                         ; Buffer for output msg data
OUTLEN: .BLKL   1                           ; Will contain length of msg
          .
          .
         .ENTRY CYGNUS, ^M<R2,R3,R4>        ; Entry mask
```

```
❹   $ASSIGN_S -                    ; Assign channel
              DEVNAM=MAILBOX, -
              CHAN=MAILCHAN
      BSBW    ERROR
         .
         .
         .
      $QIOW_S CHAN=MAILCHAN, -
              BUFFER=OUTBUF, -
              LENGTH=OUTLEN,...
      BSBW    ERROR
         .
         .
      RET
```

❶ 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 $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 you want to 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.)

❷ After creating the mailbox, process ORION calls the $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.

❸ 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 $QIOW_S.

❹ Process CYGNUS assigns a channel to the mailbox, specifying the logical name the process ORION gave the mailbox. The $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 $QIOW (without the IO$M_NOW function modifier) is used to write the message, the process will not continue executing until another process reads the message.

7.20.1 Mailbox Name

The **lognam** argument to the $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.

```
MBXDESC:  .ASCID  /CHKPNT/  ; Descriptor for mailbox logical name
          .
          .
          .
          $CREMBX_S  LOGNAME=MBXDESC,...
```

The following logical name translation takes place:

1. MBX$ is prefixed to CHKPNT.
2. MBX$CHKPNT is translated to CHKPNT_001.

Because no further translation is successful, 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 VMS operating system strips the underscore and considers the resultant string to be the actual name (that is, no further translation is 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, VMS considers the resultant string to be the actual name (that is, no further translation is performed).

7.20.2 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.

7.20.3 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. Section 8.7.2 provides an example of how to create and use a termination mailbox.

You cannot use a mailbox in memory shared by multiple processors as a process termination mailbox.

7.21 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
                                    .
                                    .
                                    .
```

8

Process Control Services

When you log in to the system, a process for the execution of program images is created. You can create another process to execute an image by issuing the RUN or SPAWN command, using any of the qualifiers that pertain to process creation. You can also write a program that creates another process to execute a particular image.

The following services are process control system services:

- Create Process ($CREPRC)
- Delete Process ($DELPRC)
- Suspend Process ($SUSPND)
- Resume Process ($RESUME)
- Hibernate ($HIBER)
- Wake ($WAKE)
- Schedule Wakeup ($SCHDWK)
- Cancel Wakeup ($CANWAK)
- Exit ($EXIT)
- Force Exit ($FORCEX)
- Declare Exit Handler ($DCLEXH)
- Cancel Exit Handler ($CANEXH)
- Set Process Name ($SETPRN)
- Set Priority ($SETPRI)
- Set Privileges ($SETPRV)
- Set Resource Wait Mode ($SETRWM)

Process control services allow you to create processes and to control a process or group of processes. This chapter describes some aspects of process control services and includes discussions of the following:

- Subprocesses and detached processes
- Execution context of a process
- Process creation
- Interprocess control and communication
- Process hibernation and suspension

- Image exit and exit handlers
- Process deletion and termination messages

8.1 Subprocesses and Detached Processes

A process is either a subprocess or a detached process. A subprocess receives a portion of its creator's resource quotas and must terminate before the creator. A detached process is fully independent; for example, the process the system creates when you log in is a detached process.

The Create Process ($CREPRC) system service creates both subprocesses and detached processes. The number of subprocesses a process can create is controlled by its PRCLM quota. The DETACH privilege controls your ability to create a detached process with a UIC that is different from the UIC of the creating process.

8.2 The 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 is mapped into the created process.

When you call the $CREPRC system service to create a process, you define the context by specifying arguments to the service.

8.3 Process Creation

Sections 8.3.1 through 8.3.5 show examples of process creation and describe how the arguments to the $CREPRC system service define the context of the process.

8.3.1 Defining an Image for a Subprocess to Execute

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

```
PROGNAME:
        .ASCID  /CARRIE/          ; Descriptor for image to execute
          .
          .
          .
        $CREPRC_S -               ; Create Process to execute CARRIE
                IMAGE=PROGNAME
```

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 Section 8.7.

8.3.2 Input, Output, and Error Devices for Subprocesses

When you call the $CREPRC system service, you can provide equivalence names for the logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR. These logical name/equivalence name pairs are placed in the process logical name table for the created process.

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

```
INSTREAM: -
        .ASCID  /SUB_MAIL_BOX/  ; Descriptor for input stream
OUTSTREAM: -
        .ASCID  /COMPUTE_OUT/   ; Descriptor for output
                                ; and error stream
PROGNAME: -
        .ASCID  /COMPUTE.EXE/   ; Descriptor for image name

          .
          .
        $CREPRC_S -             ; Create process
                IMAGE=PROGNAME, -
                INPUT=INSTREAM, - ❶
                OUTPUT=OUTSTREAM, - ❷
                ERROR=OUTSTREAM ❸
```

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

❷ 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 will be written to this file.

❸ The **error** argument equates the equivalence name COMPUTE_OUT to the logical name SYS$ERROR. All system-generated error messages will be 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 $CREPRC system service does not provide default equivalence names for the logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR. If none are specified, 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 ($GETDVI) system service to obtain the device name for the logical name SYS$INPUT, SYS$OUTPUT, or SYS$ERROR.
2. Specify the address of the descriptor returned by the $GETDVI service when you specify the **input**, **output**, or **error** argument to the $CREPRC system service.

This procedure is illustrated in the following example.

```
        $PRCDEF
        $DVIDEF
DVILIST:                                ; Begin $GETDVI item list
        .WORD   64                      ; Maximum of 16 bytes long
        .WORD   DVI$_DEVNAM             ; Get terminal name
        .ADDRESS -
                TERM                    ; Destination of terminal name
        .ADDRESS -
                TERMDESC                ; Destination of length of string
        .LONG   0                       ; End item list
;
TERMDESC:                               ; Descriptor for terminal name
        .WORD   64                      ; Maximum of 16 bytes long
        .WORD   0
TERMADDR:
        .ADDRESS -
                TERM
TERM:   .BLKB   64                      ; Terminal name is placed here
;
LOGNAM: .ASCID  /SYS$INPUT/
;
IMAGENAME:
        .ASCID  /WRKD$:[ORANGE]MIRROR/  ; Image for subprocess
;
          .
          .
          .
;
; Determine terminal name
;
        $GETDVI_S -
                DEVNAM=LOGNAM, -        ; Return information on SYS$INPUT
                ITMLST=DVILIST          ; Address of the item list
;
        BLBC    R0,SSERR                ; If not success, go to error routine
;
10$:    $CREPRC_S -                     ; Create subprocess
                IMAGE=IMAGENAME,        ; Running MIRROR
                INPUT=TERMDESC, -       ; Using creating process's
                OUTPUT=TERMDESC, -      ; terminal as the input,
                ERROR=TERMDESC, -       ; output, and error device
                BASPRI=#4               ; Set base priority to 4
          .
          .
```

When the subprocess executes, 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 VMS RMS to open the file for reading or writing, or both.
- Use the Assign I/O Channel ($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.

```
OUTPUT: .ASCID  /SYS$OUTPUT/     ; Logical name descriptor
OUTCHAN:
        .BLKW   1                ; Channel number of output device
          .
          .
          .
        $ASSIGN_S -
                DEVNAM=OUTPUT, -
                CHAN=OUTCHAN
```

For more information about channel assignment for I/O operations, see Chapter 7.

8.3.3 Disk and Directory Defaults for Created Processes

When you use the $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 (or both) for the created process that is different from that of the creating process in a call to $CREPRC. The created process can, however, define an equivalence for the logical device SYS$DISK by calling the Create Logical Name ($CRELNM) system service.

If the process is a subprocess, you 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 can also set its own default directory by calling the VMS RMS default directory control routine, SYS$SETDDIR.

A process can create a process with a default directory that is different from its own by doing the following:

1. The process that is creating a new process makes a call to SYS$SETDIR to change its own default directory.
2. The creating process makes a call to $CREPRC to create the new process.
3. The creating process makes a call to SYS$SETDIR to change its own default directory back to the default directory it had before the first call to SYS$SETDIR.

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$SETDIR, see the *VMS Record Management Services Manual*.

8.3.4 Controlling Resources of Created Processes

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 $CREPRC system service that control these characteristics follow. For details, see the descriptions of arguments to the $CREPRC system service in the *VMS System Services Reference Manual*.

- **prvadr**—This argument defines the privilege list for the created process. 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 special privilege, 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 mask specifying the GRPNAM and GROUP privileges.

```
PRVMSK: .LONG   <1@PRV$V_GRPNAM>!<1@PRV$V_GROUP> ; Grpnam and group
        .LONG   0       ; quadword mask required.  No bits set in
                        ; high-order longword for these privileges.
```

- **quota**—This argument defines the quota list for a subprocess. If you do not specify this argument, the system defines default quotas for the subprocess.
- **stsflg**—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.
- **baspri**—This argument sets the base execution priority for the created process. If not specified, it 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.

8.3.5 Detached Processes

The creation of a detached process is primarily a function performed by VMS when you log in. The DETACH privilege controls the ability to create a detached process with a UIC that is different from the UIC of the creating process. The **uic** argument to the $CREPRC system service provides one way to define whether a process is a subprocess or a detached process; it provides the created process with a user identification code (UIC). If you omit the **uic** argument, the $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 DETACH privilege to create a detached process with the same UIC as the creating process.

8.4 Interprocess Control and Communication

Processes can be either wholly independent or cooperative. The sections that follow discuss considerations for developing applications that require the concurrent execution of many programs.

8.4.1 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 GROUP privilege to issue process control services for other processes executing in the same group.
- You need WORLD privilege to issue process control services for any process in the system.

You need additional privileges to perform some specific functions; for example, to set the base priority of a process to a higher level than that of the creating process.

8.4.2 Process Identification

There are two types of process identification:

- Process identification number (PID).

  The system assigns this unique 32-bit number to a process when it is created. If you provide the **pidadr** argument to the $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 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 $CREPRC system service, as follows.

```
ORION:  .ASCID  /ORION/         ; Descriptor for process name
ORIONID:
        .LONG   0               ; Process ID returned
          .
          .
          .
        $CREPRC_S -
                PRCNAM=ORION, -
                PIDADR=ORIONID,...
```

The service returns the process identification in the longword at ORIONID. You can now use either the process name (ORION) or the process identification (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.

```
CYGNUS:  .ASCID  /CYGNUS/        ; Descriptor for process name
           .
           .
           .
         $SETPRN_S -
                 PRCNAM=CYGNUS
```

Most of the process control services accept either the **prcnam** or **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 8.4.2.1).

If you specify neither the process name argument nor the process identification number argument, the service is performed for the calling process. Table 8–1 provides a summary of the possible combinations of these arguments and an explanation of how the services interpret them.

Table 8–1 Process Identification

Process Name Specified?	Process ID Address Specified?	Contents of Process ID	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	Process ID	The process identification is used and returned.

(continued on next page)

Table 8–1 (Cont.) Process Identification

Process Name Specified?	Process ID Address Specified?	Contents of Process ID	Resultant Action by Services
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	Process ID	The process identification is used and returned; the process name is ignored.

8.4.2.1 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 user privilege GROUP; 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 user privilege WORLD. 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 user privilege WORLD.

8.4.3 Techniques for Interprocess Communication

Processes can communicate in the following ways:

- Files
- Common event flag clusters
- Logical name tables
- Mailboxes
- Global sections
- Lock management system services

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

Like 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 technique.

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

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

Common Event Flag Clusters: Processes executing within the same group can use common event flag clusters to signal the occurrence or completion of particular activities. For details on event flags and event flag clusters, and an example of how cooperating processes in the same group use a common event flag, see Chapter 4.

Logical Name Tables: Processes executing in the same job can use the jobwide 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 user privilege GRPNAM to place names in the group logical name table. All processes in the system can use the system logical name table. Processes can also create and use user-defined logical name tables. For details on logical names and logical name tables, see Chapter 6.

Mailboxes: Mailboxes can be used as virtual input/output devices to pass information, messages, or data among processes. For details on how to create and use mailboxes, with an example of how cooperating processes use a mailbox, see Chapter 7. Mailboxes may also be used to provide a creating process with a way to determine when and under what condition a created subprocess was deleted. For an example of a termination mailbox, see Section 8.7.2.

Global Sections: Global sections can be either disk files or page-file sections containing 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 only page to the system default page file. Global sections are described in more detail in Section 12.7.

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. For more information about the lock management system services, see Chapter 13.

8.5 Process Hibernation and Suspension

There are two ways to halt the execution of a process temporarily: hibernation, performed by the Hibernate ($HIBER) system service, and suspension, performed by the Suspend Process ($SUSPND) system service. The process can continue execution normally only after a corresponding Wake from Hibernation ($WAKE) system service, if it is hibernating, or after a Resume Process ($RESUME) system service, if it is suspended.

Process hibernation and suspension are compared in Table 8–2.

Table 8–2 Process Hibernation and Suspension

Hibernation	Suspension
Can only cause self to hibernate	Can suspend self or another process, depending on privilege
Reversed by $WAKE system service	Reversed by $RESUME system service
Interruptible; can receive ASTs	Noninterruptible; cannot receive ASTs
Can wake self	Cannot cause self to resume
Can schedule wakeup at an absolute time or at a fixed time interval	Cannot schedule resumption
Requires little system overhead	Requires system dynamic memory

8.5.1 Process Hibernation

The hibernate/wake mechanism provides an efficient way to prepare an image for execution and then place it in a wait state until it is needed. When you issue the wakeup request, the image is reactivated with little delay or system overhead.

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

```
Process TAURUS

ORION:  .ASCID  /ORION/          ; Descriptor for subprocess name
FASTCOMP:
        .ASCID  /COMPUTE.EXE/    ; Descriptor for image name
          .
          .
❶ $CREPRC_S -              ; Create ORION
              PRCNAM=ORION, -
              IMAGE=FASTCOMP,...
        BSBW    ERROR            ; Continue
          .
          .
❷ $WAKE_S PRCNAM=ORION     ; Wake ORION
        BSBW    ERROR
          .
          .
        $WAKE_S PRCNAM=ORION     ; Wake ORION again
        BSBW    ERROR
          .
          .

Process ORION

        .ENTRY  COMPUTE,^M<> ❸; Entry mask
10$:    $HIBER_S                 ; Sleep
        BSBW    ERROR
          .                      ; Perform...
          .
        BRW     10$              ; Back to sleep
```

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

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

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

The Schedule Wakeup ($SCHDWK) system service, a variation of the $WAKE system service, schedules a wakeup for a hibernating process at a fixed time or at an elapsed (delta) time interval. Using the $SCHDWK service, a process can schedule a wakeup for itself before issuing a $HIBER call. For an example of how to use the $SCHDWK system service, see Chapter 10.

Hibernating processes can be interrupted by Asynchronous System Traps (ASTs), as long as AST delivery is enabled. The process can call $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 5.

8.5.2 Alternate Methods of Hibernation

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

- Specify the **stsflg** argument for the $CREPRC system service, setting the bit that requests $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 $CREPRC service, the creating process can control when to wake the created process. When you use the RUN command, its qualifiers control when the process is to be awakened.

If you use the /INTERVAL qualifier and the image to be executed does not call the $HIBER system service, the image is placed in a state of hibernation whenever it issues a RET instruction. Each time the image is reawakened, it begins executing at its entry point. If the image does call $HIBER, each time it is awakened it begins executing at either the point following the call to $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 ($DELPRC) or Force Exit ($FORCEX) system service, or from the command level with the STOP command. The $DELPRC and $FORCEX system services are described in Section 8.6.4 and in Section 8.7. The RUN and STOP commands are described in the *VMS DCL Dictionary*.

These techniques 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 the integrity of data areas that are modified during its execution, as well as the status of opened files.

8.5.3 Suspension

Using the Suspend Process ($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. VMS provides no system service to force a process to be swapped out, but the $SUSPND system service can accomplish the task in the following way. Suspended processes are the first processes to be selected for swapping. A suspended process cannot be interrupted by ASTs, and can resume execution only after another process issues a Resume Process ($RESUME) system service for it. 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.

8.6 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.

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$ERROR logical name, or if an invalid or nonexistent image name is specified, the error condition is signaled in the created process.
- An exception occurring during execution of the image. When an exception occurs, any user-specified condition handlers receive control to handle the exception. If there are no user-specified condition handlers, a system-declared condition handler receives control, and it initiates exit activities for the image. Condition handling is described in Chapter 11.
- A Force Exit ($FORCEX) system service issued on behalf of the process by another process.

8.6.1 Image Rundown Activities

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.

8.6.2 The $EXIT System Service

To initiate the rundown activities described in Section 8.6.1, the system calls the Exit ($EXIT) system service on behalf of the process. In some cases, a process can call $EXIT to terminate the image itself (for example, if an unrecoverable error occurs).

The $EXIT system service accepts a status code as an argument. If you use $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 $EXIT, the current value in R0 is passed as the status code when the system calls $EXIT.

This status code is used as follows:

- The command interpreter uses the status code to optionally display 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.

8.6.3 Exit Handlers

Exit handlers are procedures that can 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 ($DCLEXH) system service. Exit handlers are called 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 $EXIT.

If an image declares more than one exit handler, the control blocks are linked together on a last-in, first-out basis. After an exit handler is called and returns control, the control block is removed from the list. Exit control blocks can also be removed prior to image exit with the Cancel Exit Handler ($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 receive control after exit handlers declared from user mode are executed.

Exit handlers are called as a part of the $EXIT system service. While a call to the $EXIT system service often precedes image rundown activities, it 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.

8.6.4 Forced Exit

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

```
CYGNUS: .ASCID  /CYGNUS/        ; Process name descriptor
          .
          .
          .
        $FORCEX_S -
                PRCNAM=CYGNUS
```

Because the $FORCEX system service calls the $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 $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 5.

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

```
EXITBLOCK: ❶                          ; Exit control block
        .LONG   0                       ; System uses this for pointer
        .ADDRESS -
                EXITRTN                 ; Address of exit handler
        .LONG   1                       ; Number of args for handler
        .ADDRESS -
                STATUS                  ; Destination of status code
STATUS: .BLKL   1                       ; Status code from $EXIT
          .
          .
        .ENTRY  PEGASUS,^M<R2,R3>       ; Entry mask for PEGASUS
❷     $DCLEXH_S -
                DESBLK=EXITBLOCK        ; Declare exit handler
        BSBW    ERROR
          .
          .
        RET                             ; End of main routine
;
; exit handler
        .ENTRY  EXITRTN,^M<R2> ; Entry mask
❸     BLBS    STATUS,10$              ; Normal exit? yes, finish
          .                             ; No, clean up
          .
          .
10$:    RET                           ; Finished
```

❶ 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.

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

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

8.7 Process Deletion

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

- The Delete Process ($DELPRC) system service is called.
- A process that created a subprocess is deleted.
- An interactive process uses the DCL command LOGOUT.
- A batch job reaches the end of its command file.
- An interactive process uses the DCL command STOP/ID=pid or STOP *username*.
- A process that contains a single image calls the Exit ($EXIT) system service.

When the system is called to delete a process as a result of any of these conditions, it first locates all subprocesses, searching hierarchically. No process can be deleted until any subprocesses it has created have been deleted.

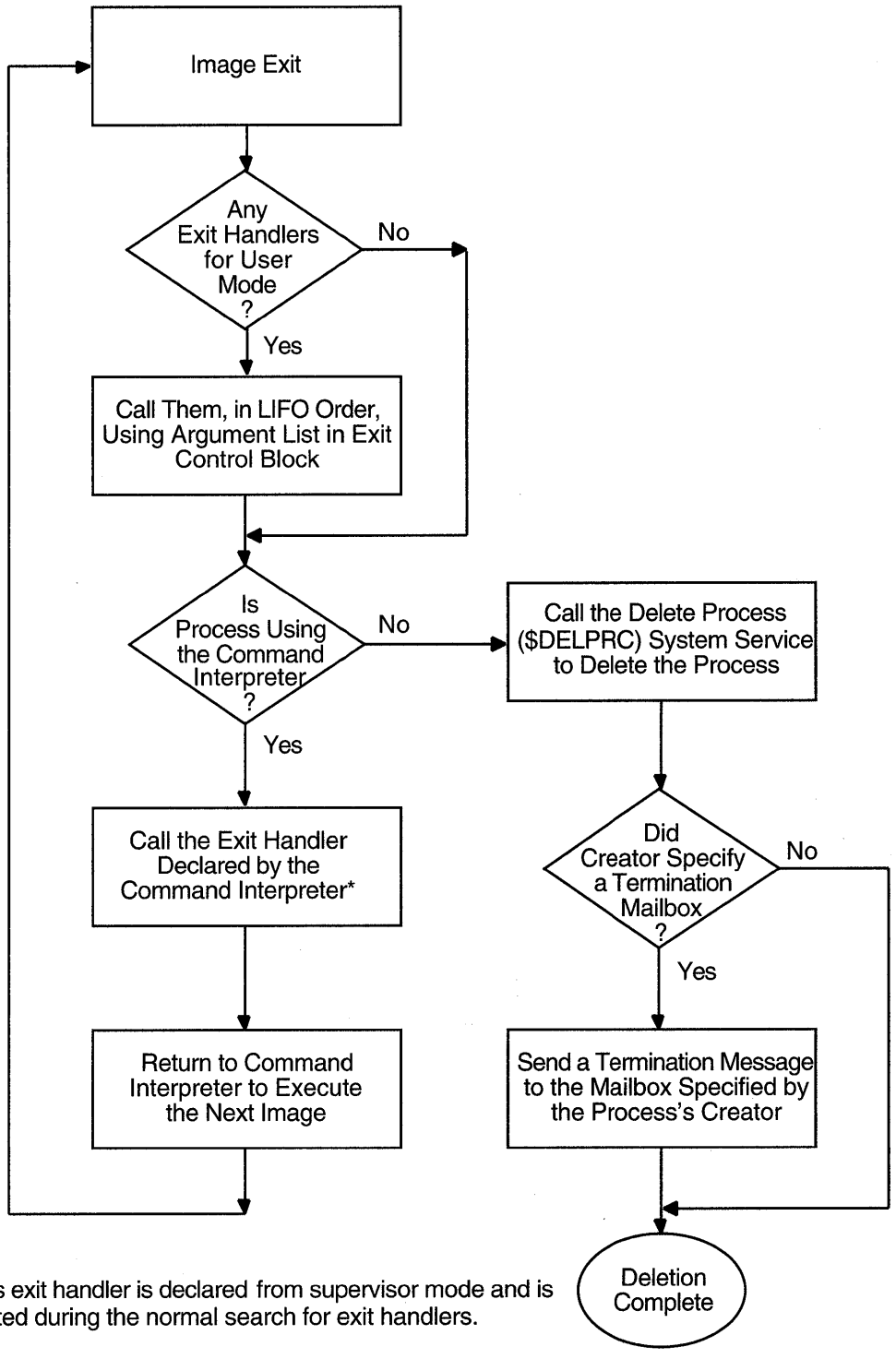
The lowest subprocess in the hierarchy is a subprocess that has no descendant subprocesses of its own. When that subprocess is deleted, its parent subprocess becomes a subprocess that has no descendant subprocesses and it can be deleted. The topmost process in the hierarchy is the process that is the ultimate parent process of all the other subprocesses.

Beginning with the lowest process in the hierarchy and ending with the topmost process, each of the following procedures is performed:

- 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 8.6.1. 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 specified 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 8–1 illustrates the flow of events from image exit through process deletion.

Figure 8–1 Image Exit and Process Deletion



*This exit handler is declared from supervisor mode and is located during the normal search for exit handlers.

ZK–0857–GE

8.7.1 The Delete Process System Service

A process can delete itself or another process at any time, depending on the restrictions outlined in Section 8.4.1. The Delete Process ($DELPRC) system service deletes a process. For example, if a process has created a subprocess named CYGNUS, it can delete CYGNUS as follows.

```
CYGNUS: .ASCID  /CYGNUS/        ;Descriptor for process name
          .
          .
          .
        $DELPRC_S-
                PRCNAM=CYGNUS
```

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 do not normally need to issue the $DELPRC system service explicitly.

As an alternative to deleting a process to stop an image, you can use the Force Exit ($FORCEX) system service to force the exit of the image executing in a process (see Section 8.6.4).

8.7.2 Termination 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 ($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 $CREPRC system service in the *VMS 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 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.

```
EXCHAN:
        .BLKW   1               ; To hold channel number of mailbox
MBXINFO:                        ; Start $GETDVI item list
        .WORD   4  ❶            ; Length of buffer
        .WORD   DVI$_UNIT
        .ADDRESS -
                UNITNUM         ; Address of buffer
        .LONG   0               ; No return length needed
        .LONG   0               ; End item list
UNITNUM:
        .WORD                   ; To receive unit number
;
EXITMSG:
        .BLKB   ACC$K_TERMLEN   ; Buffer for mailbox message
                                ; (see $SNDACC explanation
                                ;   for ACC$K_TERMLEN)
MBXIOSB:
        .BLKW   1               ; Quadword I/O status block
MBLEN:  .BLKW   1               ; Length of I/O
MBPID:  .BLKL   1               ; Receives PID of process deleted
LYRAPID:
        .LONG   0               ; Get PID of subprocess
LYREXE: .ASCID  /LYRA.EXE/      ; Name of image for subprocess
           .
           .
           .
❷             $CREMBX_S -             ; Create mailbox
                CHAN=EXCHAN, -
                MAXMSG=_#84, -
                PROMSK=_#0, -
                BUFQUO=_#240
        BSBW    ERROR
❸     $GETDVI_S -             ; Get mailbox info
                CHAN=EXCHAN, -
                ITMLST=MBXINFO
;
        BSBW    ERROR
        $CREPRC_S -             ; Create subprocess
                IMAGE=LYREXE, -
                PIDADR=LYRAPID,..., -
                MBXUNT=UNITNUM  ; Specify termination mailbox
❹     BSBW    ERROR
        $QIO_S  CHAN=EXCHAN, -  ; QIO (read) to mailbox
                FUNC=_#IO$_READVBLK, -
                ASTADR=EXITAST, -
                IOSB=MBXIOSB, -
                P1=EXITMSG, -
                P2=_#ACC$K_TERMLEN
        BSBW    ERROR
           .                    ; Continue execution
           .
           .
        RET
;
; AST routine for termination message
;
```

```
❺    .ENTRY  EXITAST,^M<>    ; Entry mask
       CMPW    MBXIOSB,_#SS$_NORMAL   ; I/O successful?
       BNEQ    20$             ; Branch if not
       CMPW    EXITMSG+ACC$W_MSGTYP,#MSG$_DELPROC
                               ; Is it a termination msg?
       BNEQ    20$             ; No, something else
       CMPL    LYRAPID,MBPID   ; Is it LYRA?
       BNEQ    20$             ; No, somebody else
       CMPL    EXITMSG+ACC$L_FINALSTS,#SS$_NORMAL
                               ; Deleted normally?
       BEQL    10$             ; Yes, return
          .                    ; No, respond to error in LYRA
          .
          .
10$:   RET                     ; AST routine finished
20$:      .                    ; Handle all other conditions
          .
          .
```

❶ The item list for the Get Device/Volume Information ($GETDVI) system service specifies that the unit number of the mailbox is to be returned.

❷ The Create Mailbox and Assign Channel ($CREMBX) system service creates the mailbox, and returns the channel number at EXCHAN.

❸ The Create Process ($CREPRC) system service creates a process to execute the image LYRA.EXE, and returns the process identification at LYRAPID. The **mbxunt** argument refers to the unit number of the mailbox, obtained from the Get Device/Volume Information ($GETDVI) system service.

❹ The Queue I/O Request queues a read request to the mailbox, specifying 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.

❺ When a message is received in the mailbox, 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. All other messages or error conditions cause a branch to the label 20$.

The Create Mailbox and Assign Channel ($CREMBX), Get Device/Volume Information ($GETDVI), and Queue I/O Request ($QIO) system services are described in greater detail in Chapter 7.

8.8 Example of Using Process Control Services

The following FORTRAN example calculates gross income and taxes, and then uses the results to calculate net income.

The INCOME.FOR program uses SYS$CREPRC, specifying a termination mailbox, to create a subprocess to calculate taxes (CALC_TAXES) while the INCOME program calculates gross income. The INCOME program issues an asynchronous read to the termination mailbox. The asynchronous read specifies 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.

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

```
                        INSTALLED.FOR
! Installed common to be linked with INCOME.FOR and
! CALC_TAXES.FOR.

! Unless the shareable image created from this file is
! in SYS$SHARE, you must define a group logical name
! INSTALLED and equate it to the full file specification
! of the shareable image.

INTEGER*4 INCOME (200),
2         TAXES (200),
2         NET (200)
COMMON /CALC/ INCOME,
2             TAXES,
2             NET

END
```

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

! Set up for SYS$GETDVI
INTEGER*4 UNIT_BUF,
2         UNIT_LEN
INTEGER*2 UNIT_BUF_LEN,
2         UNIT_BUF_CODE
INTEGER*4 UNIT_BUF_ADDR,
2         UNIT_LEN_ADDR,
2         END_LIST /0/
EXTERNAL DVI$_UNIT
COMMON /GETDVI_LIST/ UNIT_BUF_LEN,
2                    UNIT_BUF_CODE,
2                    UNIT_BUF_ADDR,
2                    UNIT_LEN_ADDR,
2                    END_LIST
```

```
! Name and I/O channel for mailbox
CHARACTER*(*) MBX_NAME
PARAMETER (MBX_NAME = 'MAIL_BOX')
INTEGER*2 MBX_CHAN
! Logical unit number for I/O
INTEGER*4 MBX_LUN
! Mailbox message
CHARACTER*84 MBX_MESSAGE
INTEGER*4 READ_CODE,
2         LENGTH
! I/O status block
INTEGER*2 IOSTAT,
2         MSG_LEN
INTEGER*4 READER_PID
COMMON /IOBLOCK/ IOSTAT,
2                MSG_LEN,
2                READER_PID

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

! Flag to indicate taxes calculated
INTEGER*4 TAX_DONE


! Get and clear an event flag.
STATUS = LIB$GET_EF (TAX_DONE)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
STATUS = SYS$CLREF (%VAL(TAX_DONE))
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! Create the mailbox.
STATUS = SYS$CREMBX (,
2                    MBX_CHAN,
2                    ,,,,
2                    MBX_NAME)
IF (.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

! Get unit number of the mailbox.
UNIT_BUF_LEN = 4
UNIT_BUF_CODE = %LOC(DVI$_UNIT)
UNIT_BUF_ADDR = %LOC(UNIT_BUF)
UNIT_LEN_ADDR = %LOC(UNIT_LEN)
STATUS = SYS$GETDVIW (,
2                     %VAL(MBX_CHAN),
2                     MBX_NAME,          ! device
2                     UNIT_BUF_LEN,      ! common
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                 IOSTAT,,,          ! 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. IOSTAT) CALL LIB$SIGNAL (%VAL(IOSTAT))

! Calculate net income after taxes.
                  .
                  .
                  .
END

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

! Calculate taxes.
                        .
                        .
                        .
END
```

9

Process Information Services

The VMS process information services enable you to gather information about processes. You can obtain information about one process or a group of processes on the local system or on remote nodes in a VAXcluster system. DCL commands such as SHOW SYSTEM and SHOW PROCESS use the process information services to display information about processes. You can use the process information services within your programs.

The following are process information system services:

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

For detailed information about $GETJPI and $PROCESS_SCAN, see the *VMS System Services Reference Manual.*

9.1 Overview of $GETJPI and $GETJPI with $PROCESS_SCAN

$GETJPI returns information about processes. $GETJPI uses the PID or the process name to obtain information about one process and the –1 wildcard to obtain information about all processes. $GETJPI cannot perform a selective search—it can only search for one process 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 $GETJPI with $PROCESS_SCAN.

$PROCESS_SCAN provides a process context that is used by $GETJPI to return information about processes on the local system or across the cluster. $PROCESS_SCAN can be used only with $GETJPI; it cannot be used alone. The process context generated by $PROCESS_SCAN is used like the –1 wildcard except that it is initialized by calling the $PROCESS_SCAN service instead of by a simple assignment statement. However, the $PROCESS_SCAN context is more powerful and more flexible than the –1 wildcard. $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 $GETJPI.

Using $GETJPI with $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 $GETJPI returns only the processes that match the specified user name. You can specify a search for all batch processes and $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 $GETJPI returns only information about processes owned by that user name that are running as batch jobs.

9.1.1 Using the Process ID to Obtain Information

$GETJPI returns 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.

9.1.2 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.

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

9.2 Using $GETJPI Alone

Using $GETJPI alone limits you to obtaining information about one process at a time or information about all processes on the local system. To obtain information about one process (either a local or a remote process), specify the PID or the process name. To obtain information about all processes on the local system, use the –1 wildcard as the **pidadr**. If no PID or process name is specified, $GETJPI returns information about the calling process.

9.2.1 Requesting Information About a Single Process

Example 9–1 is a FORTRAN program that displays the process name and the PID of the calling program.

Example 9–1 Using $GETJPI to Obtain Information About the Calling Process

```
! 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
   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                     %VAL(1),    ! Event flag 1
 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
```

(continued on next page)

Example 9–1 (Cont.) Using $GETJPI to Obtain Information About the Calling Process

```
! 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 9–2 demonstrates how to use the process name to obtain information about a process.

Example 9–2 Using $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 9-1

! Call $GETJPI to get data for a named process

STATUS = SYS$GETJPIW (
2                   %VAL(1),     ! Event flag 1
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
```

9.2.2 Requesting Information About All Processes on the Local System

You can use $GETJPI to perform a wildcard search on all processes on the local system. When the **pidadr** argument is specified as –1, $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 $GETJPI.

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

When performing wildcard searches, $GETJPI returns an error status for processes that are inaccessible. When a program that uses a –1 wildcard checks the status value returned by $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 9–3 is a MACRO program that demonstrates how to use the $GETJPI –1 wildcard to search for all processes on the local system.

Example 9–3 Using $GETJPI to Request Information About All Processes on the Local System

```
          .TITLE   WILDJPI - Wildcard $GETJPI example program

          $JPIDEF                        ; Define $GETJPI item codes

          .PSECT   DATA  RD,WRT,NOEXE

IOSB:     .QUAD    0                     ; Completion status
PID:      .LONG    -1                    ; Wildcard PID initialized to -1

ITEMS:    .WORD    32                    ; Size of user name buffer
          .WORD    JPI$_USERNAME         ; User name item code
          .ADDRESS UNAME                 ; Address of user name buffer
          .ADDRESS UNAMESIZ              ; Address to return user name size
          .LONG    0                     ; End of list

UNAMEDSC:                                ; Length and address form a string
                                         ;  descriptor for LIB$PUT_OUTPUT
UNAMESIZ: .LONG    0                     ; Buffer for size of user name
          .ADDRESS UNAME                 ; Address of user name buffer

UNAME:    .BLKB    32                    ; User name buffer

          .PSECT   CODE  EXE,NOWRT

          .ENTRY   START, ^M<>

LOOP:     $GETJPIW_S -                   ; Get information and wait
                   EFN=#1, -             ;  - use event flag 1
                   PIDADR=PID, -         ;  - use wildcard pid
                   ITMLST=ITEMS, -       ;  - address of item list
                   IOSB=IOSB             ;  - always use IOSB for status check
          BLBC     R0,10$                ; If failure in R0, check that status
          MOVZWL   IOSB,R0               ; If success in R0, then move status
                                         ;  from IOSB to R0 for checks
10$:      BLBS     R0,DISPLAY            ; If success in both R0 and IOSB,
                                         ;  then display this user name
          CMPW     R0,#SS$_NOPRIV        ; No privilege for this process?
          BEQL     LOOP                  ; If no privilege, try next process
          CMPW     R0,#SS$_SUSPENDED     ; Process suspended?
          BEQL     LOOP                  ; If yes, try next process
          CMPW     R0,#SS$_NOMOREPROC    ; No more processes?
          BEQL     DONE                  ; If yes, finished
          BRB      ERROR                 ; Otherwise, exit with error code in R0
```

(continued on next page)

Example 9–3 (Cont.) Using $GETJPI to Request Information About All Processes on the Local System

```
DISPLAY:  PUSHAL   UNAMEDSC            ; Pass address of the user name descriptor
          CALLS    #1,G^LIB$PUT_OUTPUT ; Display name on SYS$OUTPUT
          BRB      LOOP                ; Get the next process

DONE:     MOVL     #SS$_NORMAL,R0      ; Put success status into R0
ERROR:    $EXIT_S  R0                  ; Exit with status in R0

          .END     START
```

9.3 Using $GETJPI with $PROCESS_SCAN

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

9.3.1 Using the $PROCESS_SCAN Item List and Item-Specific Flags

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

Each entry in the $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 compare to the value in the item list in the following ways.

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$M_GTR	Match if value is greater than
PSCAN$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 in the following FORTRAN example.

```
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$M_WILDCARD flag to specify that a character string is to be treated as a wildcard. For example, if you want to search for all process names that begin with the letter *A* and end with the string *ER*, use the string *A*ER* with the PSCAN$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*.

9.3.2 Requesting Information About Processes That Match One Criterion

You can use $GETJPI with $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, $GETJPI returns only those processes that match the specified user name.

Example 9–4 demonstrates how to perform a $PROCESS_SCAN search on the local node to select all processes that are owned by user SMITH.

Example 9–4 Using $GETJPI and $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
```

(continued on next page)

Example 9–4 (Cont.) Using $GETJPI and $PROCESS_SCAN to Select Process Information by User Name

```
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

!*****************************************
!*  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
```

(continued on next page)

Example 9–4 (Cont.) Using $GETJPI and $PROCESS_SCAN to Select Process Information by User Name

```
        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                   %VAL(1),     ! Event flag 1
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

 END DO

 END
```

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

$PROCESS_SCAN can also search for processes that match one of a number of values for a single criterion, for example, 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. $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 9–5 to initialize the item list in Example 9–4.

Example 9–5 Using $GETJPI and $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     *
!*****************************************
```

9.3.4 Requesting Information About Processes That Match Multiple Criteria

$PROCESS_SCAN can be used 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 9–6 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 9–6 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
```

(continued on next page)

Example 9–6 (Cont.) Selecting Processes That Match Multiple Criteria

```
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).END_LIST = 0

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

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

9.3.5 Specifying a Node as Selection Criterion

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

$PROCESS_SCAN ignores a node specification if it is running on a node that is not part of a VAXcluster system. For example, if you request that $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 that system is a MicroVAX.

A remote $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 $GETJPI operations are asynchronous, and must be properly synchronized. Many applications that are not correctly synchronized might seem to work on a single node because some $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 $GETJPI operations, see the section on synchronizing system service completion in the *Introduction to VMS System Services*.
- The CLUSTER_SERVER process is always a current process, because it is executing on behalf of $GETJPI.
- Attempts by $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 $GETJPI status of SS$_UNREACHABLE.
- SS$_NOMOREPROC is returned after all processes on all specified nodes have been scanned.

9.3.6 Scanning All Nodes on the Cluster for Processes

$PROCESS_SCAN 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 9–7 to initialize the item list to find all processes with a nonzero cluster system identifier (CSID) and a user name of SMITH.

Example 9–7 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     *
!*****************************************
```

9.3.7 Scanning Specific Nodes on the Cluster for Processes

You can specify a list of nodes as well. Example 9–8 demonstrates how to design an item list to search for batch processes on the nodes TIGNES, VALTHO, or 2ALPES.

Example 9–8 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

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     *
!*****************************************
```

9.3.8 Conducting Multiple Simultaneous Searches with $PROCESS_SCAN

Only one asynchronous remote $GETJPI request per $PROCESS_SCAN context is permitted at a time. If you issue a second $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 $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 $PROCESS_SCAN once for each context. For example, you can design a program that calls $GETSYI in a loop to find the nodes in the VAXcluster system and creates a separate $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 $PROCESS_SCAN context specifies the local node, any number of $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 $GETJPI to reference remote processes, you must properly synchronize all $GETJPI calls. Before VMS 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 $GETJPI with $PROCESS_SCAN with a remote process, your program might attempt to use the data before $GETJPI has returned it.

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

9.4 Programming Considerations

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

9.4.1 Using Item Lists Correctly

When $GETJPI collects data, it makes multiple passes through the item list. If the item list is self-modifying—that is, if the addresses for the output buffers in the item list point back at the item list—$GETJPI replaces the item list information with the returned data. Therefore, incorrect data might be read or unexpected errors might occur when $GETJPI reads the item list again.

The number of passes needed by $GETJPI 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.

The results from $GETJPI are unpredictable when an item list has buffer pointers that point back at the item list itself. To prevent confusing errors, Digital recommends that you do not use self-modifying item lists.

9.4.2 Improving Performance by Using Buffered $GETJPI Operations

To request information about a process located on a remote node, $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 $GETJPI locates.

To reduce the overhead of such a remote search, use $PROCESS_SCAN with the PSCAN$_GETJPI_BUFFER_SIZE item code to specify a buffer size for $GETJPI. When the buffer size is specified by $PROCESS_SCAN, $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 $GETJPI item list requests 100 bytes of information, you might specify a PSCAN$_GETJPI_BUFFER_SIZE of 1000 bytes so that the service can place information for at least 10 processes in each message. ($GETJPI does not send fill data in the message buffer; therefore, it is possible that information for more than 10 processes can be packed into the buffer.)

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

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

If you use PSCAN$_GETJPI_BUFFER_SIZE with $PROCESS_SCAN, all calls to $GETJPI using that context must request the same item code information. Because $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 $GETJPI item list between calls. $GETJPI returns the error SS$_BADPARAM if any item code or buffer length changes between $GETJPI calls. However, you can change the buffer addresses in the $GETJPI item list from call to call.

The $GETJPI buffered operation is not used for searching the local node. When a search specifies both multiple nodes and $GETJPI buffering, the buffering is used on remote nodes but is ignored on the local node. Example 9–9 demonstrates how to use a $GETJPI buffer to improve performance.

Example 9–9 Using a $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     *
!*****************************************
```

9.4.3 Meeting Remote $GETJPI Quota Requirements

A remote $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 $GETJPI request:

1. Add the following:
   - The size of the $PROCESS_SCAN item list
   - The total size of all reference buffers for $PROCESS_SCAN (the sum of all buffer length fields in the item list)
   - The size of the $GETJPI item list
   - The size of the $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 $GETJPI requests. For buffered requests, use the value specified in the $PROCESS_SCAN item, PSCAN$_GETJPI_BUFFER_SIZE, as the size of the buffer. For nonbuffered requests, use the total length of all data buffers specified in the $GETJPI item list as the size of the buffer.

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

9.4.4 Using $GETJPI Control Flags

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

If $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 $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 $GETJPI item list when any of the JPI$_GETJPI_CONTROL_FLAGS flags is used.

Unlike other $GETJPI item codes, the JPI$_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 $GETJPI how to interpret the item list, it must be the first entry in the $GETJPI item list. The error code SS$_BADPARAM is returned if it is not the first item in the list.

The following are the $GETJPI control flags.

JPI$M_NO_TARGET_INSWAP

When JPI$M_NO_TARGET_INSWAP is specified, $GETJPI does not retrieve a process that has been swapped out of the balance set. JPI$M_NO_TARGET_INSWAP is used to avoid the additional load of swapping processes into a system. For example, this flag is used 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:

- Any data stored in the virtual address space of the process is not accessible.
- Any data stored in the process header (PHD) may not be accessible.
- Any 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 VMS.

JPI$M_NO_TARGET_AST

When JPI$M_NO_TARGET_AST is specified, $GETJPI does not deliver a kernel-mode AST to the target process. JPI$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:

- Any 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 VMS.

The use of the flag JPI$M_NO_TARGET_AST also implies that $GETJPI does not swap in a process, because $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, $GETJPI attempts to retrieve as much information as possible, even if the process is suspended or being deleted. JPI$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 9–10 demonstrates how to use $GETJPI control flags to avoid swapping processes during a $GETJPI call.

Example 9–10 Using $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

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
```

(continued on next page)

Example 9–10 (Cont.) Using $GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
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

!*****************************************
!*  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
```

(continued on next page)

Example 9–10 (Cont.) Using $GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
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)

        ! Call $GETJPI to get the next process

        STATUS = SYS$GETJPIW (
2                   %VAL(1),    ! Event flag 1
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
```

10

Timer and Time Conversion Services

Many applications require the scheduling of program activities based on clock time. Under the VMS operating system, an image can schedule events for a specific time of day or after a specified time interval. The timer and time conversion services are as follows:

- Get Time ($GETTIM)
- Convert Binary Time to Numeric Time ($NUMTIM)
- Convert Binary Time to ASCII String ($ASCTIM)
- Convert ASCII String to Binary Time ($BINTIM)
- Set Timer ($SETIMR)
- Cancel Timer Request ($CANTIM)
- Schedule Wakeup ($SCHDWK)
- Cancel Wakeup ($CANWAK)
- Set System Time ($SETIME)

You may use timer services to schedule, convert, or cancel events. For example, you can use the timer 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.

The timer services require you to specify the time in a 64-bit format. To work with the time in different formats, you can use time conversion services to 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.

This chapter describes the system time format and the services that use it, with examples of how to schedule program activities using the timer services.

10.1 The System Time Format

The VMS operating system maintains the current date and time in 64-bit format. The time value is a binary number in 100-nanosecond 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 astronomical 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 and time of day. 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, VMS supplies the current date and time.

10.2 Obtaining the Current Date and Time

You obtain the current time in system format by using the Get Time ($GETTIM) system service, which places the time into a quadword buffer. For example:

```
TIME:   .BLKQ   1               ; Buffer for time
          .
          .
          .
        $GETTIM_S -
                TIMADR=TIME     ; Get time
```

This call to $GETTIM returns the current date and time in system format in the quadword buffer TIME.

The Convert Binary Time to ASCII String ($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 $ASCTIM system service as follows.

```
ATIMENOW:                       ; Descriptor for ASCII time
        .LONG   23              ; Length of buffer
        .ADDRESS -
                TIMESTR         ; Address of buffer
TIME_VALUE:                     ; 64-bit time value to be converted
        .BLKQ   1
TIMESTR:
        .BLKB   23              ; 23 bytes returned
          .
          .
          .
        $ASCTIM_S -
                TIMBUF=ATIMENOW, -
                TIMADR=TIME_VALUE
```

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

where:

dd	Is the day of the month.
mmm	Is the month (a 3-character alphabetic abbreviation).
yyyy	Is the year.
hh:mm:ss.cc	Is the time in hours, minutes, seconds, and hundredths of seconds.

10.3 Obtaining an Absolute Time in System Format

The converse of the $ASCTIM system service is the Convert ASCII String to Binary Time ($BINTIM) system service. You provide the service with the time in the ASCII format shown in Section 10.2. 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 $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.

```
ASCII_NOON:
        .ASCID  /-- 12:00:00.00/        ; Descriptor for ASCII 12 noon
BINARY_NOON:
        .BLKQ   1                       ; Buffer for binary 12 noon
          .
          .
          .
        $BINTIM_S -                     ; Convert time
                TIMBUF=ASCII_NOON, -
                TIMADR=BINARY_NOON
```

When the $BINTIM service completes, a 64-bit time value representing "noon today" is returned in the quadword at BINARY_NOON.

10.4 Obtaining a Delta Time in System Format

The $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 $BINTIM service to obtain a delta time in system format.

```
ATENMIN:
        .ASCID  /0 00:10:00.00/ ; Descriptor for ASCII ten minutes
BTENMIN:
        .BLKQ   1               ; Buffer for binary ten minutes
          .
          .
          .
        $BINTIM_S -             ; Convert time
                TIMBUF=ATENMIN, -
                TIMADR=BTENMIN
```

If you are a VAX MACRO programmer, you can also specify approximate delta time values when you assemble a program, using two MACRO .LONG directives to represent a time value in terms of 100-nanosecond units. The arithmetic is based on the following formula:

```
1 second = 10 million * 100 nanoseconds
```

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-nanosecond units that can be expressed in a longword. In terms of time values, this is slightly more than 7 minutes.

10.5 Timer Requests

Timer requests made with the Set Timer ($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 $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.

Examples 1 and 2 show timer requests using event flags and ASTs. Event flags and event flag services are described in more detail in Chapter 4. ASTs are described in more detail in Chapter 5.

Example 1: Setting an Event Flag

```
A30SEC: .ASCID  /0 00:00:30.00/ ; Descriptor for ASCII 30
                                ;   seconds, delta time
B30SEC: .BLKQ   1               ; Quadword to hold converted
                                ;   (binary) delta time
          .
          .
          .
        $BINTIM_S -             ; Convert to binary
                TIMBUF=A30SEC, -
                TIMADR=B30SEC
        BSBW    ERROR
❶     $SETIMR_S -             ; Set time to wait
                EFN=#4, -
                DAYTIM=B30SEC
        BSBW    ERROR           ; Call error routine
❷     $WAITFR_S -             ; Wait 30 seconds
        EFN=#4
        BSBW    ERROR
          .
          .
          .
```

❶ The call to $SETIMR requests that event flag 4 be set in 30 seconds (expressed in the quadword B30SEC).

❷ The Wait for Single Event Flag ($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 2: Using an AST Service Routine

```
ANOON:  .ASCID  /-- 12:00:00.00/ ; Descriptor for ASCII 12 noon
BNOON:  .BLKQ   1               ; To hold converted (binary) noon
          .
          .
          .
❶     $BINTIM_S -             ; Convert to binary
                TIMBUF=ANOON, -
                TIMADR=BNOON
        BSBW    ERROR
❷     $SETIMR_S -
                DAYTIM=BNOON, - ; Set timer for noon,
                ASTADR=ASTSERV, - ; Specify AST routine,
                REQIDT=#12      ; Request ID of 12 as AST parameter
        BSBW    ERROR
          .
          .
          .
        RET
;
❸     .ENTRY  ASTSERV,^M<> ; Entry mask for AST routine
        CMPL    #12,4(AP)       ; Is this a "noon" AST request?
        BNEQ    10$             ; If not, handle other type(s)
          .                     ; Handle "noon" AST request
          .
          .
        RET
10$:      .                     ; Handle other types of requests
          .
          .
        RET
```

❶ The call to $BINTIM converts the ASCII string representing 12:00 noon to format. The value returned in BNOON is used as input to the $SETIMR system service.

❷ The AST routine specified in the $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 5.) 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.

❸ This AST service routine checks the parameter passed by the **reqidt** argument and checks 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.

Canceling Timer Requests

The Cancel Timer Request ($CANTIM) system service cancels timer requests that have not been processed. The $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 2, you would use the following call to $CANTIM:

```
$CANTIM_S REQIDT=#12
```

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.

10.6 Scheduled Wakeups

Example 1 in Section 10.5 shows a process placing itself in a wait state using the $SETIMR and $WAITFR services. A process can also make itself inactive by hibernating. A process hibernates by issuing the Hibernate ($HIBER) system service; hibernation is reversed by a wakeup request, which can be put into effect immediately with the $WAKE system service or scheduled with the Schedule Wakeup ($SCHDWK) system service. For more information about the $HIBER and $WAKE system services, see Section 8.5.

The following example shows a process scheduling a wakeup for itself prior to hibernating.

```
ATENSEC:
        .ASCID  /0 00:00:10.00/ ; Descriptor for
                                ;    10-second wait time

BTENSEC:
        .BLKQ   1               ; To hold binary ten-second value
          .
          .
          .
        $BINTIM_S -             ; Convert time
                TIMBUF=ATENSEC, -
                TIMADR=BTENSEC
        $SCHDWK_S -             ; Schedule wakeup
                DAYTIM=BTENSEC
        $HIBER_S                ; Sleep ten seconds
```

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 $SCHDWK system service in a program, you can specify that the wakeup request be repeated at fixed time intervals. See Chapter 8 for more information on hibernation and wakeup.

Canceling Scheduled Wakeups

You can cancel scheduled wakeup requests that are pending but have not yet been processed with the Cancel Wakeup ($CANWAK) system service.

The following example shows the scheduling of wakeup requests for a process, CYGNUS, and the subsequent cancellation of the wakeups. The $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.

```
CYGNUS: .ASCID  /CYGNUS/         ; Descriptor for process name
ONE_MIN:
        .ASCID  /0 00:01:00.00/ ; Descriptor for 1 min (delta)
INTERVAL:
        .BLKQ   1                ; 8 bytes to hold binary 1 min
          .
          .
          .
        $BINTIM_S -              ; Convert to binary
                TIMBUF=ONE_MIN, -
                TIMADR=INTERVAL
          .
          .
          .
        $SCHDWK_S -              ; Wake up every minute
                PRCNAM=CYGNUS, -
                DAYTIM=INTERVAL, -
                REPTIM=INTERVAL
          .
          .
          .
        $CANWAK_S -              ; Cancel wakeups
                PRCNAM=CYGNUS
          .
          .
          .
```

10.7 Numeric and ASCII Time

The Convert Binary Time to Numeric Time ($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 $NUMTIM system service and the format of the information returned are described in the *VMS System Services Reference Manual*.

You use the $ASCTIM system service to format the time in ASCII for inclusion in an output string. The $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 ($FAO) system service. The $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 *VMS System Services Reference Manual*.

10.8 Setting the System Time

The Set System Time ($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. There is normally no 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. You may also 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 $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; the other specifies an absolute time of 9:00.

```
        .TITLE  SCORPIO Show scheduled wakeups
        .PSECT  READ_ONLY_DATA,NOEXE,RD,NOWRT
ABS_TIME:
        .ASCID  /-- 9:00:00.00/         ; Absolute time of 9:00 AM
DELTA_TIME:
        .ASCID  /0 :05:00/              ; Delta time of 5 minutes
;
        .PSECT  WRITEABLE_DATA,NOEXE,RD,WRT
;
ABS_BINARY:
        .BLKQ   1                       ; Absolute time in 64-bit format
DELTA_BINARY:
        .BLKQ   1                       ; Delta time in 64-bit format
;
```

```
        .PSECT  CODE,EXE,PIC,NOSHR,RD,NOWRT
        .ENTRY  SCORPIO,^M<>
        $BINTIM_S -                     ; Convert absolute time to
                TIMBUF=ABS_TIME, -      ; Binary
                TIMADR=ABS_BINARY
        BLBS    R0,10$                  ; Check for error
        BRW     ERR                     ; If so, exit
;
10$:    $SETIMR_S -                     ; Set timer to wake AST routine
                DAYTIM=ABS_BINARY, -    ; at 9:00 AM
                ASTADR=GEMINI, -        ; Routine is GEMINI
                REQIDT=#1               ; Request ID number 1
        BLBS    R0,20$                  ; Check for error
        BRW     ERR
;
20$:    $BINTIM_S -                     ; Convert delta time to
                TIMBUF=DELTA_TIME, -    ; binary
                TIMADR=DELTA_BINARY
        BLBS    R0,30$                  ; Check for error
        BRW     ERR                     ; If so, exit
;
30$:    $SETIMR_S -                     ; Set timer to wake AST routine
                DAYTIM=DELTA_BINARY, -  ; in 15 minutes
                ASTADR=GEMINI, -        ; Routine is GEMINI
                REQIDT=#2               ; Request ID number 2
        BLBS    R0,40$
        BRW     ERR
;
40$:    $HIBER_S                        ; Hibernate process
;
EXIT:   $EXIT_S
;
ERR:    PUSHL   R0
        CALLS   #1,G^LIB$SIGNAL
        BRW     EXIT
;
;
        .PSECT  READ_ONLY_DATA,NOEXE,RD,NOWRT
FAO_IN: .ASCID  "Request ID !UB answered at !AS."
        .PSECT  WRITEABLE_DATA,NOEXE,RD,WRT
NOWDESC:
        .LONG   12
        .ADDRESS -
                TIMENOW
TIMENOW:
        .BLKB   12
FAO_OUT:
        .LONG   80
        .ADDRESS -
                FAO_STR
FAO_STR:
        .BLKB   80
;
        .PSECT  CODE,EXE,PIC,NOSHR,RD,NOWRT
        .ENTRY  GEMINI,^M<R6,R7,R8,R9,R10,R11>
        $ASCTIM_S -                     ; Find out the current time
                TIMBUF=NOWDESC, -
                CVTFLG=#1               ; Hours, mins, secs, only
        BLBS    R0,10$
        BRW     ERR
;
```

```
10$:    $FAO_S  CTRSTR=FAO_IN, -        ; Format string
                OUTBUF=FAO_OUT, -       ; Place in FAO_OUT
                OUTLEN=FAO_OUT, -
                P1=4(AP), -             ; Request ID
                P2=#NOWDESC             ; Current time
        BLBS    R0,20$
        BRW     ERR
;
20$:    PUSHAL  FAO_OUT
        CALLS   #1,G^LIB$PUT_OUTPUT
        RET
;
        .END    SCORPIO
```

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-1990 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
$
```

10.9 Example of Using the Timer Service

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

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

1. 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$CREPRC system service to indicate that the created process should hibernate before executing the program.
2. Use the SYS$SCHDWK system service to specify the time at which the system should wake the subprocess and a time interval at which the system should repeat the wakeup call.

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

```
INCLUDE '($SYSSRVNAM)'   !Declare system services
INCLUDE '($PRCDEF)'      !Declare process options

! SYS$CREPRC options and values
INTEGER OPTIONS,
2       STATUS

! 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.

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))

END
```

11

Condition-Handling Services

A **condition handler** is a procedure that is given control when an exception occurs. An **exception** is an event that is detected by the hardware or software and that interrupts the execution of an image. Examples of exceptions include arithmetic overflow or underflow and reserved opcode or operand faults.

If you determine that a program needs to be informed of particular exceptions so that 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.

This section describes how the VMS condition-handling mechanism works and explains how to write a condition handler. You use the following system services in writing a condition handler:

- Set Exception Vector ($SETEXV)
- Set System Service Failure Exception Mode ($SETSFM)
- Unwind from Condition Handler Frame ($UNWIND)
- Declare Change Mode or Compatibility Mode Handler ($DCLCMH)

11.1 Types of Exception

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 messages are documented in the *VMS System Messages and Recovery Procedures Reference Manual* or, for certain software routines, in the manual associated with that routine. (VAX MACRO error messages appear in the *VAX MACRO User's Guide*.)

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 ($SETSFM) system service. For example:

```
$SETSFM_S ENBFLG=#1
```

System service failure exception mode is initially disabled, and may be enabled or disabled at any time during the execution of an image.

Table 11–1 provides a summary of common conditions caused by exceptions. The condition names are listed in the first column. The second column explains the condition more fully by giving information about the type, meaning, and arguments relating to the condition. The condition type is either trap or fault. Because the explanation of types is complicated, you should refer to the *VAX Architecture Handbook* for more detailed information. The meaning of the exception condition is a short description of each condition. The arguments for the condition handler are listed, if any apply; they give specific information about the condition.

Table 11–1 Summary of Exception Conditions

Condition Name	Explanation	
SS$_ACCVIO	Type:	Fault.
	Description:	Access violation.
	Arguments:	1. Reason for access violation. This is a mask with the following format: Bit 0 = type of access violation 0 = page table entry protection code did not permit intended access 1 = P0LR, P1LR, or SLR length violation Bit 1 = page table entry reference 0 = specified virtual address not accessible 1 = associated page table entry not accessible Bit 2 = intended access 0 = read 1 = modify 2. Virtual address to which access was attempted or, on some processors, virtual address within the page to which access was attempted.
SS$_ARTRES	Type:	Trap.
	Description:	Reserved arithmetic trap.
	Arguments:	None.

(continued on next page)

Table 11–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS$_ASTFLT	Type:	Trap.
	Description:	Stack invalid during attempt to deliver an AST.
	Arguments:	1. Stack pointer value when fault occurred. 2. AST parameter of failed AST. 3. Program counter (PC) at AST delivery interrupt. 4. Processor status longword (PSL) at AST delivery interrupt.[1] 5. Program counter (PC) to which AST would have been delivered.[1] 6. Processor status longword (PSL) to which AST would have been delivered.[1]
SS$_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.
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

[1]The PC and PSL 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 modes with the Declare Change Mode or Compatibility Mode Handler ($DCLCMH) system service, that routine receives control when the associated trap occurs.

[3]If a compatibility mode handler has been declared with the Declare Change Mode or Compatibility Mode Handler ($DCLCMH) system service, that routine receives control when this fault occurs.

(continued on next page)

Table 11–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS$_DECOVF	Type:	Trap.
	Description:	Decimal overflow.
	Arguments:	None.
SS$_FLTDIV	Type:	Trap.
	Description:	Floating/decimal divide by zero.
	Arguments:	None.
SS$_FLTDIV_F	Type:	Fault.
	Description:	Floating divide by zero fault.
	Arguments:	None.
SS$_FLTOVF	Type:	Trap.
	Description:	Floating overflow.
	Arguments:	None.
SS$_FLTOVF_F	Type:	Fault.
	Description:	Floating overflow fault.
	Arguments:	None.
SS$_FLTUND	Type:	Trap.
	Description:	Floating underflow.
	Arguments:	None.
SS$_FLTUND_F	Type:	Fault.
	Description:	Floating underflow fault.
	Arguments:	None.
SS$_INTDIV	Type:	Trap.
	Description:	Integer divide by zero.
	Arguments:	None.
SS$_INTOVF	Type:	Trap.
	Description:	Integer overflow.
	Arguments:	None.
SS$_OPCCUS	Type:	Fault.
	Description:	Opcode reserved to customer fault.
	Arguments:	None.
SS$_OPCDEC	Type:	Fault.
	Description:	Opcode reserved by Digital fault.
	Arguments:	None.

(continued on next page)

Table 11–1 (Cont.) Summary of Exception Conditions

Condition Name	Explanation	
SS$_PAGRDERR	Type:	Fault.
	Description:	Read error occurred during an attempt to read a faulted page from disk.
	Arguments:	1. Translation not valid reason. This is a mask with the following format: Bit 0 = 0 Bit 1 = page table entry reference 0 = specified virtual address not valid 1 = associated page table entry not valid Bit 2 = intended access 0 = read 1 = modify 2. Virtual address of referenced page.
SS$_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$_SUBRNG	Type:	Trap.
	Description:	Subscript range trap.
	Arguments:	None.
SS$_TBIT	Type:	Fault.
	Description:	Trace bit is pending following an instruction.
	Arguments:	None.

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:

- 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 ($DCLCMH) system service to establish procedures to receive control when one of these conditions occurs. The $DCLCMH system service is described in the *VMS System Services Reference Manual*.

11.2 Specifying Condition Handlers

You can establish condition handlers to receive control in the event of an exception in two ways:

- By specifying the address of the entry mask of a condition handler in the first longword of a procedure call frame
- By establishing exception handlers with the Set Exception Vector ($SETEXV) system service

The first of these methods is the preferred way to specify a condition handler for a particular image. The use of call frame 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)
```

The high-level language programmer can call the common Run-Time Library routine LIB$ESTABLISH (see the *VMS Run-Time Library Routines Volume*); however, some languages provide access to condition handling as part of the language.

Each procedure on the call stack can declare a condition handler.

The $SETEXV system service allows you to specify addresses for a primary exception handler, a secondary exception handler, and a last-chance exception handler. Handlers may be specified for each access mode. The primary exception vector is reserved for the debugger. In general, you should avoid using the vectored handlers unless absolutely necessary. If you use a vectored handler, it must be prepared for all exceptions occurring in that access mode.

An address of 0 in the first longword of a procedure call frame or in an exception vector indicates that no condition handler exists for that call frame or vector.

11.3 The 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 in the following order:

1. The primary exception vector for the access mode at which the program was executing when the exception occurred.
2. The secondary exception vector for the access mode at which the program was executing when the exception occurred.
3. The condition handler address specified in the procedure call stack of the access mode at which the program was executing when the exception occurred. The exception dispatcher scans call frames on the stack backwards, using the saved frame pointer in each call frame to refer to the previous call frame.

4. The last-chance exception vector for the access mode at which the program was executing when the exception occurred.

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 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 11–1 illustrates the exception dispatcher's search of the call stack for a condition handler.

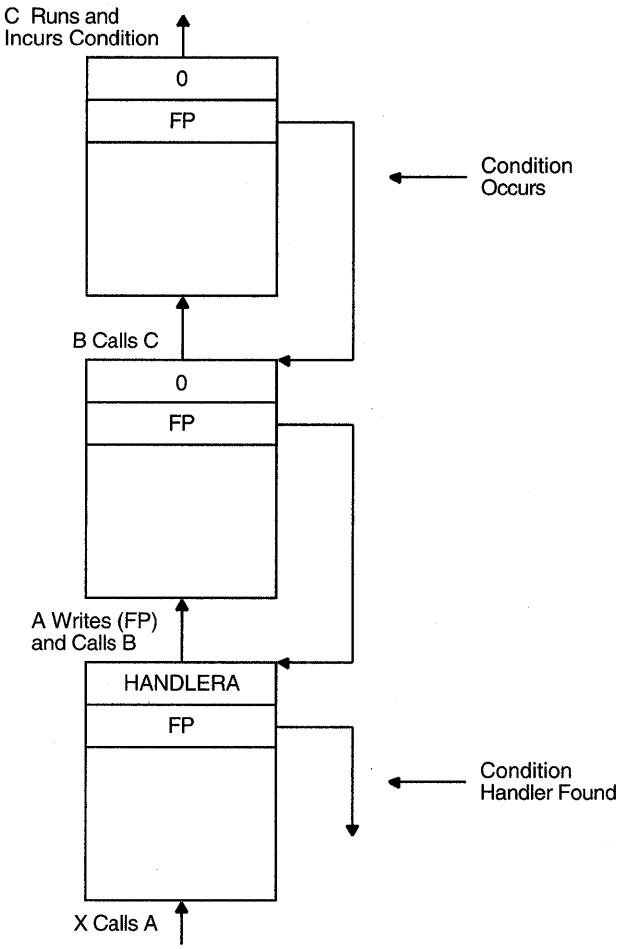
11.4 The Argument List Passed to a Condition Handler

When the dispatcher finds a condition handler, it passes control to it using a CALLG instruction. The argument list passed to the condition handler is constructed on the stack and consists of the addresses of two argument arrays, as illustrated in Figure 11–2; these arguments are described in detail in Sections 11.4.1 and 11.4.2.

Using the $CHFDEF macro instruction, you can define the following symbolic names to refer to these arguments.

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

Figure 11–1 Search of Stack for 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.

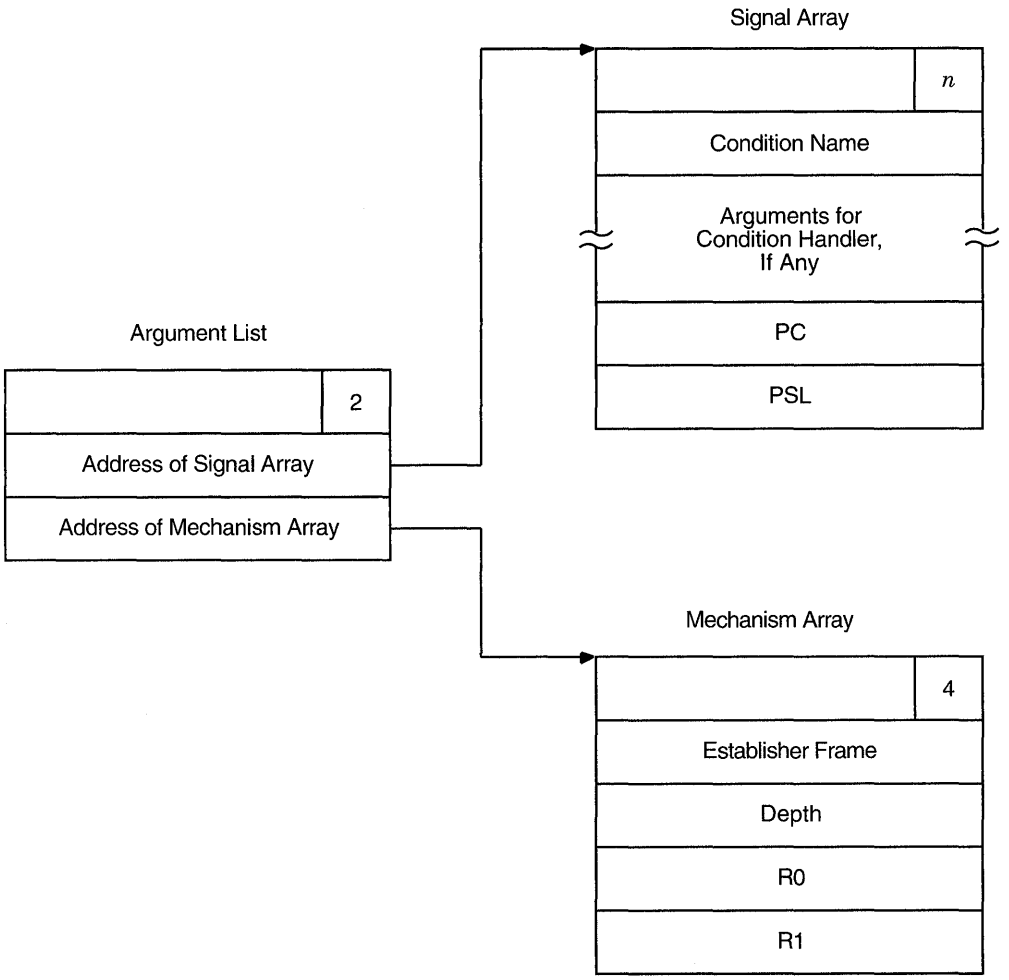
2 An exception occurs while Procedure C is executing. The exception dispatcher searches for a condition handler.

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 HANDLERA.

ZK–0858–GE

Figure 11–2 Argument List and Arrays Passed to Condition Handler



You can define symbolic names to refer to these arguments using the $CHFDEF macro instruction. The symbolic names are as follows:

Symbolic Offset	Value
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

ZK–0859–GE

11.4.1 Signal Array Arguments

The signal array contains the following values describing the condition:

- Condition name—The symbolic value assigned to the specific condition. The possible exception conditions and their symbolic definitions are listed in Table 11–1.
- Arguments—Specific information relating to the condition (see Table 11–1).
- PC—The program counter at the time of the exception. Depending on the type of exception (fault or trap), this can be the address of the instruction that caused the exception (for a fault), or of the following instruction (for a trap).
- PSL—The processor status longword at the time of the exception.

11.4.2 Mechanism Array Arguments

The mechanism array describes the context in which the exception occurred. The exception dispatcher supplies the following arguments:

- Establisher frame—The frame pointer (FP) registers contents of the call frame that established the condition handler. This is the address of the longword containing the condition handler address. For example, if the call stack is as shown in Figure 11–1, 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 FP of the procedure.
- Depth—The frame number of the procedure that established the condition handler, relative to the frame of the procedure that incurred the exception. The depth is determined as follows.

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
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 11–1, the depth argument passed to HANDLERA would have a value of 2.

The condition handler can use this argument to determine whether it wants to handle the condition. For example, the handler may not want to handle the condition if the exception that caused the condition did not occur in the establisher frame.

- R0—The contents of register 0 when the exception occurred.
- R1—The contents of register 1 when the exception occurred.

11.5 Courses of Action for the Condition Handler

After the condition-handling routine determines the nature of the exception, it can take one of the three following courses of action:

- 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.

- 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.

- Unwind

  The condition handler cannot fix the problem, and execution cannot continue while using the current flow. The handler issues the Unwind Call Stack ($UNWIND) system service to unwind the call stack. Call frames may then be removed from the stack and the flow of execution modified, depending on the arguments to the $UNWIND service.

11.5.1 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.

```
        .ENTRY  PGMA,^M<>               ; Entry mask for
                                        ; procedure A
❶ MOVAB  HANDLERA,(FP)              ; Declare condition handler
       $SETSFM_S -
                ENBFLG=#1               ; Enable SSFAIL
                                        ; exceptions
❷ CALLG   ARGLIST,PGMB              ; Call procedure B
            .
            .
            .

❸.ENTRY   HANDLERA,^M<R2,R3,R4>  ; Entry mask of HANDLERA
       MOVL     CHF$L_SIGARGLST(AP),R4  ; Get addr of signal args
       CMPL     #SS$_SSFAIL,CHF$L_SIG_NAME(R4)
                                        ; System service failure?
       BNEQ     10$                     ; No - resignal
         .                              ; handle SSFAIL exception
❹        .
         .
       MOVZWL   #SS$_CONTINUE,R0        ; Signal to continue
       RET                              ; Return to exception
                                        ; dispatcher
```

```
10$:    MOVZWL   #SS$_RESIGNAL,R0        ; Signal to resignal
        RET                              ; Return to dispatcher

        .ENTRY   PGMB,^M<R2,R3,R4>       ; Entry mask of procedure B
❺MOVAB     HANDLERB,(FP)            ; Declare condition handler
          .
          .
          .      <-- System service failure occurs ❻
          . ❼

❽.ENTRY   HANDLERB,^M<R2,R3,R4>  ; Entry mask of HANDLERB
;
        MOVL     CHF$L_SIGARGLST(AP),R4  ; Get addr of signal args
        CMPL     #SS$_BREAK,CHF$L_SIG_NAME(R4) ; Breakpoint fault?
        BNEQ     10$                     ; No, resignal
          .                              ; Yes, handle exception
          .
          .
        MOVZWL   #SS$_CONTINUE,R0        ; Signal to continue
        RET                              ; Return to exception
                                         ; dispatcher
10$:    MOVZWL   #SS$_RESIGNAL,R0        ; Signal to resignal ❾
        RET                              ; Return to dispatcher
```

❶ Procedure A executes and establishes condition handler HANDLERA. HANDLERA is set up to respond to exceptions caused by failures in system service calls.

❷ During its execution, Procedure A calls Procedure B.

❸ The exception dispatcher resumes its search for a condition handler and calls HANDLERA.

❹ 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.

❺ Procedure B establishes condition handler HANDLERB. HANDLERB is set up to respond to breakpoint faults.

❻ While Procedure B is executing, an exception occurs caused by a system service failure.

❼ The dispatcher returns control to Procedure B, and execution of Procedure B resumes at the instruction following the system service failure.

❽ The exception dispatcher searches the exception vectors for a condition handler (assume there are none defined), and then searches the call stack. HANDLERB is called with the condition SS$_SSFAIL.

❾ Because HANDLERB handles only breakpoint faults, it places the return value SS$_RESIGNAL in R0 and returns control to the exception dispatcher.

11.5.2 Unwinding the Call Stack

The third course 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 $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.

The $UNWIND system service 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 $UNWIND, the unwind is performed to the caller of the procedure that established the condition handler that is issuing the $UNWIND service. Control is returned to the address specified in the return PC for that procedure. Note that this is the default and normal case for unwinding.

Another common case of unwinding is to unwind to the procedure that declared the handler. This is done by using the depth value from the exception mechanism array (CHF$L_MCH_DEPTH) as the depth argument to $UNWIND.

It therefore follows that the default unwind (no depth specified) is equivalent to specifying CHF$L_MCH_DEPTH plus 1. In certain cases 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 11–3 illustrates an unwind situation and describes some of the possible results.

The unwind operation consists of two parts:

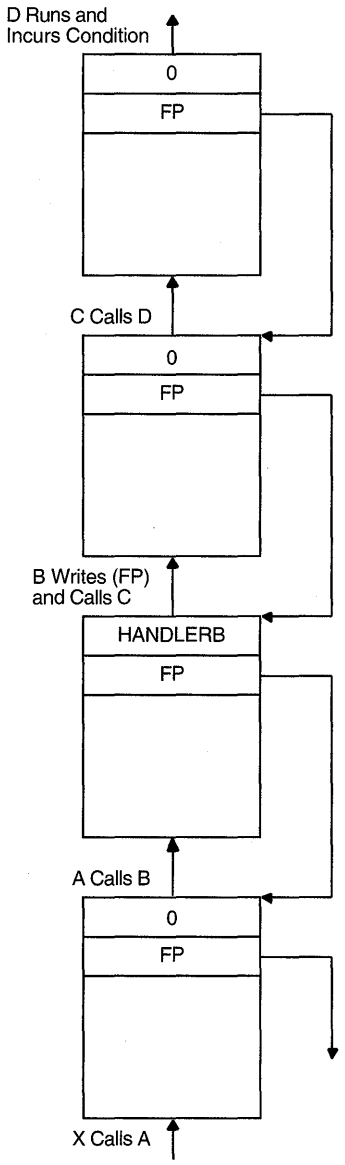
1. In the call to $UNWIND, the return PCs saved in the stack are modified to point into a routine within the $UNWIND service, but the entire stack remains present.
2. When the handler returns, control is directed to this routine by the modified PCs. It proceeds to return to itself, removing the modified 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 $UNWIND. Handlers should in general return immediately after calling $UNWIND.

During the actual unwinding of the call stack, the unwind routine examines each frame in the call stack to see if 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. After the handler returns, the call frame is removed from the stack.

Thus, in Figure 11–3, HANDLERB may be called a second time, during the unwind operation. Note that HANDLERB 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.

Figure 11–3 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, HANDLERB receives control.

3 If HANDLERB 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 HANDLERB itself), and control returns to Procedure A. Procedure A receives control at the point following its call to Procedure B.

4 If HANDLERB 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

11.6 Multiple Exceptions

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 11.3.

If the $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 established in the primary or secondary exception vector, that handler must handle the additional condition. Failure to do so correctly may result in a recursive exception loop in which the vectored handler is repeatedly called until the user stack is exhausted.

11.7 Example of Using Condition-Handling Services

This section contains an example of how to use condition-handling services.

You should 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.

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.

Because in the following example the cleanup operation is to be performed whether the program completes successfully or not, 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))
```

12

Memory Management Services

The VMS 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. Memory management system services are as follows:

- Expand Program/Control Region ($EXPREG)
- Create Virtual Address Space ($CRETVA)
- Delete Virtual Address Space ($DELTVA)
- Create and Map Section ($CRMPSC)
- Map Global Section ($MGBLSC)
- Delete Global Section ($DGBLSC)
- Update Section File on Disk ($UPDSEC)
- Lock Pages in Working Set ($LKWSET)
- Unlock Pages from Working Set ($ULWSET)
- Adjust Working Set Limit ($ADJWSL)
- Purge Working Set ($PURGWS)
- Lock Page in Memory ($LCKPAG)
- Unlock Page in Memory ($ULKPAG)
- Set Protection on Pages ($SETPRT)
- Set Process Swap Mode ($SETSWM)
- Set Stack Limits ($SETSTK)

Memory management services allow you to control the size of virtual and physical memory address space available to a program. For example, they allow you to do the following:

- Increase or decrease the virtual address space available in the program or control region of a process.
- Control the process's working set size and the exchange of pages between physical memory and the paging device.
- Define disk files containing data or shareable images and map the files into the virtual address space of a process.

This chapter discusses the services that provide these capabilities. However, before you use any of these services, you should have an understanding of the VAX memory structure and memory management routines. Where pertinent, virtual memory concepts related to the use of particular services are discussed in this section.

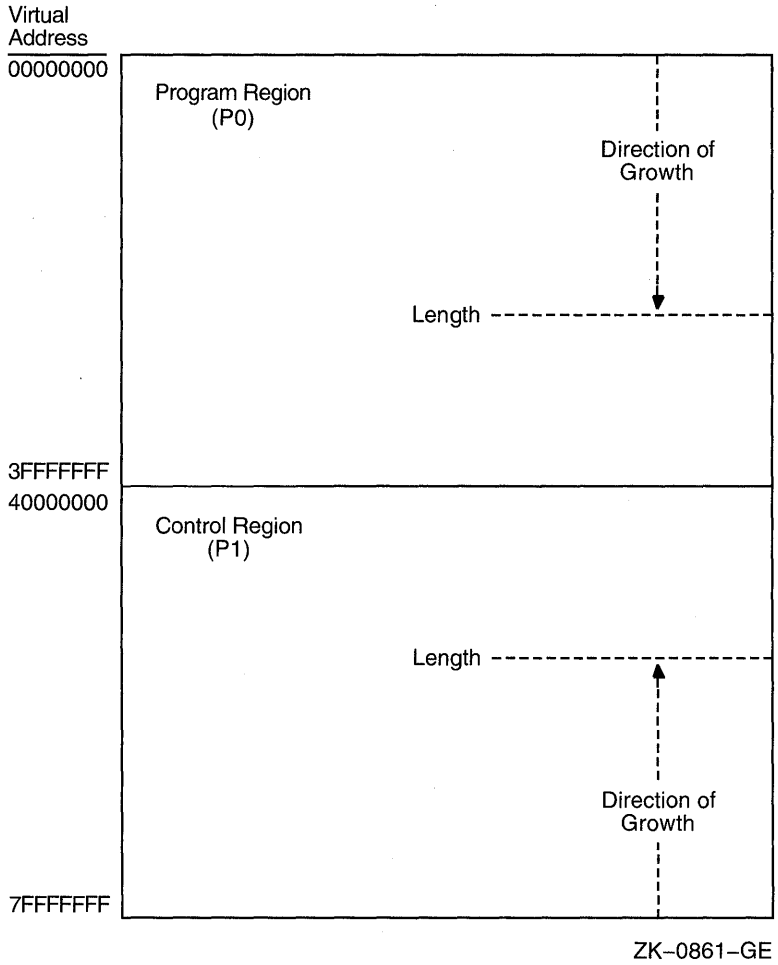
12.1 Virtual Address Space

The virtual address space of a process is divided into two regions:

- The program region (P0), which contains the image currently being executed.
- The control region (P1), which contains the information maintained by the system on behalf of the process. It also contains the user stack, which expands toward the lower-addressed end of the control region.

Figure 12–1 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 12–1 Layout of Process Virtual Address Space



To facilitate memory protection and mapping, the virtual address space is subdivided into 512-byte units called **pages.** Using memory management 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. (By using the STACK= option in a linker options file, you can also expand the user stack when you link the image.)

The maximum size to which a process can increase its address space is controlled by the SYSGEN parameter VIRTUALPAGECNT.

12.2 Increasing and Decreasing Virtual Address Space

The Expand Program/Control Region ($EXPREG) system service adds pages to the end of either the program or control region, and optionally returns the range of virtual addresses of the new pages. For example, if you want to add four pages to the program region of a process, you can write a call to the $EXPREG system service, as follows.

```
BEGSPACE:
        .BLKL   2               ; 2 longwords to hold start
                                ; and end of new pages
          .
          .
          .
        $EXPREG_S -             ; Get 4 pages
                PAGCNT=#4, -
                RETADR=BEGSPACE, -
                REGION=#0
```

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 $EXPREG service is optional; if it is not specified, the pages are added to or deleted from the program region by default.

The $EXPREG service can add pages only to the end of a particular region. When you need to add pages that are not at the end of these regions, you can use the Create Virtual Address Space ($CRETVA) system service. Likewise, when you need to delete pages created by either $EXPREG or $CRETVA, you can use the Delete Virtual Address Space ($DELTVA) system service. For example, if you have used the $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 $DELTVA system service as shown in the following example.

```
BEGSPACEA:
        .BLKL   2               ; Start and end of 1st area
BEGSPACEB:
        .BLKL   2               ; Start and end of 2nd area
          .
          .
          .
        $EXPREG_S -             ; Four pages
                PAGCNT=#4, -
                RETADR=BEGSPACEA, -
                REGION=#0
        BSBW    ERROR
          .
          .
          .
        $EXPREG_S -             ; Three more
                PAGCNT=#3, -
                RETADR=BEGSPACEB, -
                REGION=#0
        BSBW    ERROR
          .
          .
        $DELTVA_S -             ; Delete first 4 pages
                INADR=BEGSPACEA
        BSBW    ERROR
```

In this example, the first call to $EXPREG adds four pages to the program region; the virtual addresses of the created pages are returned in the two-longword array at BEGSPACEA. The second call adds three pages, and returns the addresses at BEGSPACEB. The call to $DELTVA deletes the first four pages that were added.

12.3 Input Address Arrays and Return Address Arrays

When the $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 or Delete Virtual Address Space ($CRETVA and $DELTVA) system service, 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.

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 nine bits of the specified virtual address (the low-order nine bits contain the byte offset within the page). The virtual addresses returned indicate the byte offsets.

Table 12–1 shows some sample virtual addresses that may be specified as input to $CRETVA or $DELTVA and shows the return address arrays, if all pages are successfully added or deleted.

Table 12–1 Sample Virtual Address Arrays

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 12–1, 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.

12.4 Page Ownership and Page 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 ($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, or 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 –1 in both longwords of the return address array.

If the **retadr** argument is not specified, no information is returned.

12.5 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.

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 following system services allow a process, within limits, to counteract these potential problems:

- The Adjust Working Set Limit ($ADJWSL) system service increases or decreases the maximum number of pages that a process can have in its working set.
- The Purge Working Set ($PURGWS) system service removes one or more pages from the working set.
- The Lock Pages in Working Set ($LKWSET) system service makes one or more pages in the working set ineligible for paging.

The initial size of a process's working set is defined by the process's working set default (WSDEFAULT) quota. Because some programs may have larger memory requirements than others, a program can call the $ADJWSL system service to dynamically increase the process's working set limit. When the additional pages are no longer needed in the working set, the program can call the $ADJWSL system service to decrease the working set limit. It can also call the $PURGWS system service to remove from the working set pages that are no longer in use. The maximum size of a process's working set is defined by the process's working set quota (WSQUOTA).

Under some circumstances, an image may not want certain pages to be paged out at all; in this case, the image can lock these pages in the working set with the Lock Pages in Working Set ($LKWSET) system service. As long as the process's working set is in memory, these pages cannot be paged out until they are explicitly unlocked with the Unlock Pages in Working Set ($ULWSET) system service.

12.6 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 ($SETSWM) system service, as follows:

```
$SETSWM_S SWPFLG=#1
```

This call to $SETSWM disables process swap mode. You can also disable swap mode by setting the appropriate bit in the STSFLG argument to the Create Process ($CREPRC) system service; however, you need the PSWAPM privilege to alter process swap mode.

A process can also lock pages in memory with the Lock Pages in Memory ($LCKPAG) system service. When a page is locked in memory with this service, the page remains in memory even when the remainder of the process's working set is swapped out of the balance set. This system service can be useful in special circumstances, for example, for routines that perform I/O operations to devices without using the VMS I/O system.

You can unlock pages locked in memory with the Unlock Pages in Memory ($ULKPAG) system service. However, you need the PSWAPM privilege to issue the $LCKPAG or $ULKPAG system service.

12.7 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 mapping, are discussed in Section 12.7.15.

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:

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 ($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 creating, mapping, and using 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 12.7.9 discusses global page-file sections.

12.7.1 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.

12.7.2 Opening the Disk File

Before you can use a file as a section, you must open it using VMS RMS. The following example shows the VMS RMS file access block ($FAB) and $OPEN macros used to open the file, and the channel specification to the $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 VMS RMS assigns the channel using the access mode of the caller. VMS RMS returns the channel number on which the file is accessed; this channel number is specified as input to the $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 $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, you should 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 $CREATE to open an existing file; if the file does not exist, it will be 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.

12.7.3 Defining the Section Extents

After the file is opened successfully, the $CRMPSC system service can create a section from the entire file, or from only certain portions of it. The following arguments to $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 mapping, the **vbn** argument specifies the starting page frame number.

12.7.4 Defining the Section Characteristics

The **flags** argument to the $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 12.7.10)
- Whether the section is to be mapped to a disk file or to specific physical page frames (see Section 12.7.15).

Table 12–2 shows the flag bits that must be set for specific characteristics.

Table 12–2 Flag Bits to Set for Specific Section Characteristics

	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.

12.7.5 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 12.7.6.

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 ($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.

12.7.6 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 SYSGEN 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.

```
NAMEDESC:
        .ASCID   /GSDATA/        ; Descriptor for logical name
          .                      ; of section
          .
          .
        $CRMPSC_S -
                GSDNAM=NAMEDESC,...
```

The following logical name translation takes place:

1. GBL$ is prefixed to GDSDATA.
2. GBL$GSDATA is translated to GSDATA_001. (No further translation is 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 VMS operating system strips the underscore and considers the resultant string to be the actual name (that is, no further translation is 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, VMS considers the resultant string to be the actual name (that is, no further translation is performed).
- If the global section has a name in the format *name_nnn*, VMS 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.

12.7.7 Mapping Sections

When you call the $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.

If you know specifically which pages the section should be mapped into, you provide these addresses in a two-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.

```
MAPRANGE:
        .LONG   ^X1400          ; Address (hex) of page 10
        .LONG   ^X2300          ; 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 (P0) or control (P1) region, 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 program or control region. 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.

```
MAPRANGE:
        .LONG   ^X200           ; Any program (P0) region address
        .LONG   ^X200           ; Any P0 address (can be same)
RETRANGE:
        .BLKL   2               ; Address range returned here
          .
          .
          .
        $CRMPSC_S -
                INADR=MAPRANGE, -
                RETADR=RETRANGE,-
                FLAGS=<SEC$M_EXPREG>,...
          .
          .
          .
```

The addresses specified do not have to be currently in the virtual address space of the process. The $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.

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.

```
SECFAB: $FAB    FNM=<SECTION.TST>, -
                FOP=UFO, -
                FAC=PUT, -
                SHR=<GET,PUT,UPI>
;
MAPRANGE:
        .LONG   ^X1400                  ; First page
        .LONG   ^X2300                  ; 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$:    $CRMPSC_S -
                INADR=MAPRANGE,-        ; Input address array
                RETADR=RETRANGE,-       ; Output array
                PAGCNT=#4,-             ; Map four pages
                FLAGS=#SEC$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 $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 four 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 $CRMPSC completes, the addresses of the four 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.

12.7.8 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 ($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 (**indent** 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), specifying 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 12.7.6). A demand-zero global section in memory shared by multiple processors must be mapped when it is created.

Cooperating processes can both issue a $CRMPSC system service to create and map the same global section. The first process to call the service actually creates the global section; subsequent attempts to create and map the section result only in mapping the section for the caller. The successful return status code SS$_CREATED indicates that the section did not already exist when the $CRMPSC system service was called. If the section did exist, the status code SS$_NORMAL is returned.

The example in Section 12.7.10 shows one process (ORION) creating a global section and a second process (CYGNUS) mapping the section.

12.7.9 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 pages where no initialization is necessary, but the pages are saved in a file.) The SYSGEN 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 ($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.

12.7.10 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 $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 $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
FLGCLUSTER:     ; Descriptor for common event flag cluster name
        .ASCID  /FLAG_CLUSTER/
GLOBALSEC:                      ; Descriptor for global section name
        .ASCID  /GLOBAL_SECTION/
;
FLGSET = 65                     ; Flag number to associate and set

GBLFAB: $FAB    FNM=<GLOBAL.TST>, -
                FOP=<UFO,CIF,CBT>,-
                ALQ=4, -
                FAC=PUT
          .
          .
          .
❶     $ASCEFC_S -
                EFN=#FLGSET, -
                NAME=FLGCLUSTER
        BLBS    R0,10$
        BSBW    ERROR
❷     $CRMPSC_S -             ; Create global section
10$:            GSDNAM=GLOBALSEC,-
                FLAGS=#SEC$M_WRT!SEC$M_GBL,...
        BLBS    R0,20$
        BSBW    ERROR
        $SETEF_S -              ; Set common event flag
20$:            EFN=#FLGSET

Process CYGNUS

CLUSTER:
        .ASCID  /FLAG_CLUSTER/   ; Cluster name descriptor
SECTION:
        .ASCID  /GLOBAL_SECTION/  ; Section name descriptor
FLGSET = 65

          .
          .
          .
```

```
❸       $ASCEFC_S -
                EFN=#FLGSET, -
                NAME=CLUSTER
        BLBS    R0,10$
        BSBW    ERROR
        $WAITFR_S -
10$:            EFN=#FLGSET
        BLBS    R0,20$
        BSBW    ERROR
        $MGBLSC_S -
20$:            INADR=MAPRANGE, -
                RETADR=RETRANGE,-
                FLAGS=#SEC$M_GBL,- ; Global section
                GSDNAM=SECTION  ; Section name
        BSBW    ERROR
```

❶ 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.

❷ 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.

❸ 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.

12.7.11 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. Or, 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.

12.7.12 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 ($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 ($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 system service, as follows:

```
$DASSGN_S CHAN=GBLFAB+FAB$L_STV
```

12.7.13 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 ($UPDSEC) system service writes the modified pages in a section into the disk file. The $UPDSEC system service is described in the *VMS System Services Reference Manual.*

12.7.14 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 VMS 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 *VMS Linker Utility Manual*. For information about how to install shareable images and make them available for sharing as global sections, see the *Guide to Maintaining a VMS System*.

12.7.15 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 a disk file section.

To create a page frame section, you must specify page frame number mapping by setting the SEC$M_PFNMAP flag bit in the **flags** argument to the Create and Map Section ($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 $CRMPSC service to create or map this type of section. For the same reason, the SEC$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.

12.8 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 the $CRMPSC system service), 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 the $MGBLSC system service), 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

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
```

13

Lock Management Services

The VMS lock management system services allow cooperating processes to synchronize their access to shared resources. This synchronization can be accomplished by providing a common data area in which processes can lock a specified resource by name. All processes that access the resources must use the VMS lock management services, or they are not effective.

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 Enqueue Lock Request ($ENQ) system service is used to make lock requests and the Dequeue Lock Request ($DEQ) system service is used to cancel lock requests. The Get Lock Information ($GETLKI) system service is used to get information about existing locks.

13.1 Concepts of Resources and Locks

A resource can be any entity on the VMS operating system (for example, files, data structures, databases, and 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; 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 ($ENQ) system service. There are three required arguments to the $ENQ system service for new locks:

- A resource name. The lock management services use the resource name to look for other lock requests that use the same name.
- The lock mode to be associated with the requested lock. The lock mode indicates how the process wants to share the resource with other processes.
- The 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 lock modes 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 it can access the resource.

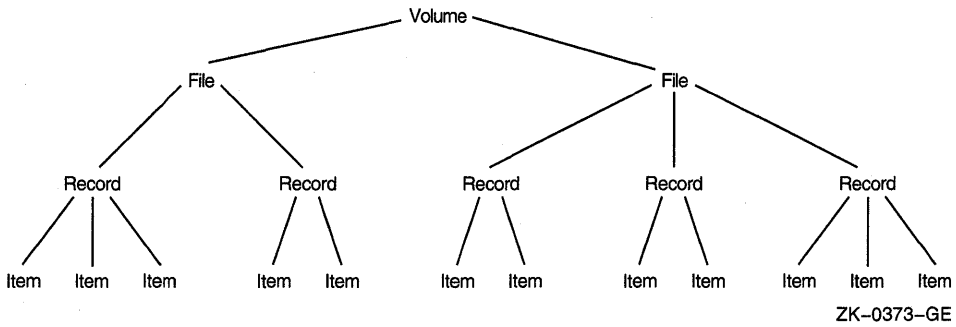
Processes can also use the $ENQ system service to change the lock mode of a lock. This is called a **lock conversion.**

13.1.1 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 **granularity** describes the part of the resource being locked.

Figure 13–1 depicts a model of a database. The database is divided into areas, which in turn are subdivided into records. The records are further divided into items.

Figure 13–1 Model Database



The processes that request locks on the database shown in Figure 13–1 must 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.

13.1.2 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 $ENQ system service. Access modes, their numeric values, and symbolic names are discussed in Section 2.1.3.

Resources can be group-specific or systemwide. The default is for resource names to be qualified by the group number of the calling process's UIC. You define systemwide locks by setting a flag bit in the call to the $ENQ system service. You need the user privilege SYSLCK 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. 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.

13.1.3 Choosing a Lock Mode

The mode of a lock determines whether the resource can be shared with other lock requests. The six lock modes are as follows.

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 the resource to be shared with other readers. No writers are allowed access to the resource. This is the traditional "share lock."
LCK$K_PWMODE	Protected write. This mode grants write access to the resource and allows the resource to be shared with concurrent read mode readers. No other writers are allowed access to the resource. This is the traditional "update lock."
LCK$K_EXMODE	Exclusive. The exclusive mode grants write access to the resource and prevents the resource from being shared with any other readers or writers. This is the traditional "exclusive lock."

13.1.4 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 modes are null, concurrent read, concurrent write, protected read, protected write, and exclusive. 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. Higher-level lock modes are less compatible with other lock modes than are lower-level lock modes. Table 13–1 shows the compatibility of the lock modes.

Table 13–1 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 lock
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive lock

13.1.5 Lock Management Queues

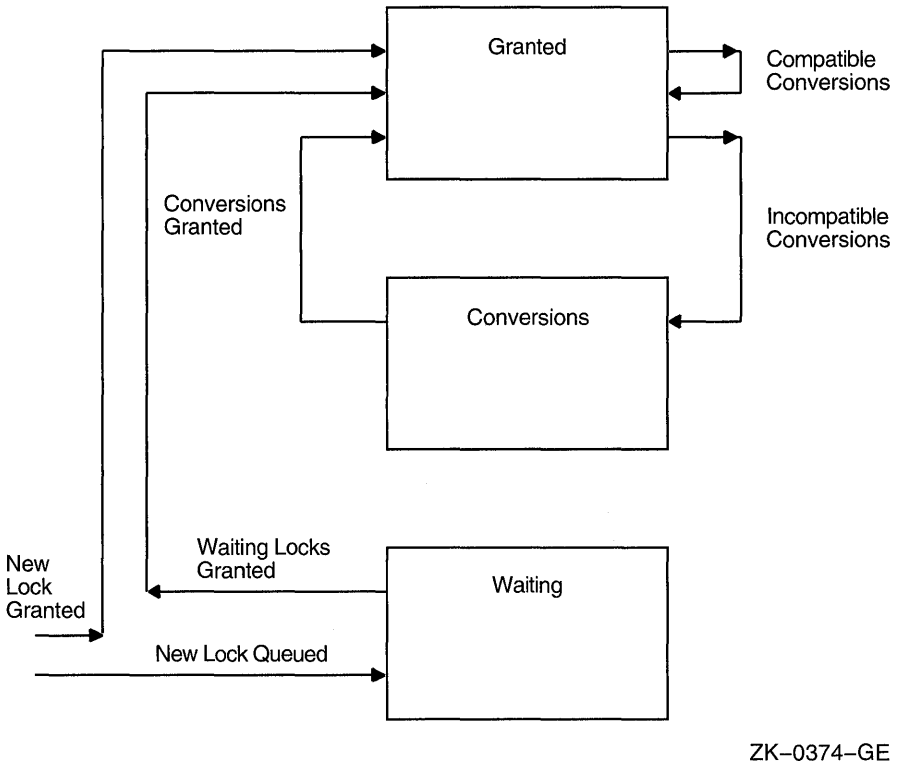
A lock on a resource can be in one of the following three states.

- 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 higher lock mode.

A queue is associated with each of the three states (see Figure 13–2).

When you request a new lock, the lock management services first determine if the resource is currently known (that is, if any other processes have locks on that resource). If the resource is new (that is, 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 if any other locks are waiting in either the conversion or 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 if 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 on 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 13–2 Three Lock Queues



13.1.6 Lock Conversion Concepts

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 13.3.6) 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 on the conversion queue. The lock retains its old lock mode and does not receive its new lock mode until the request is granted.

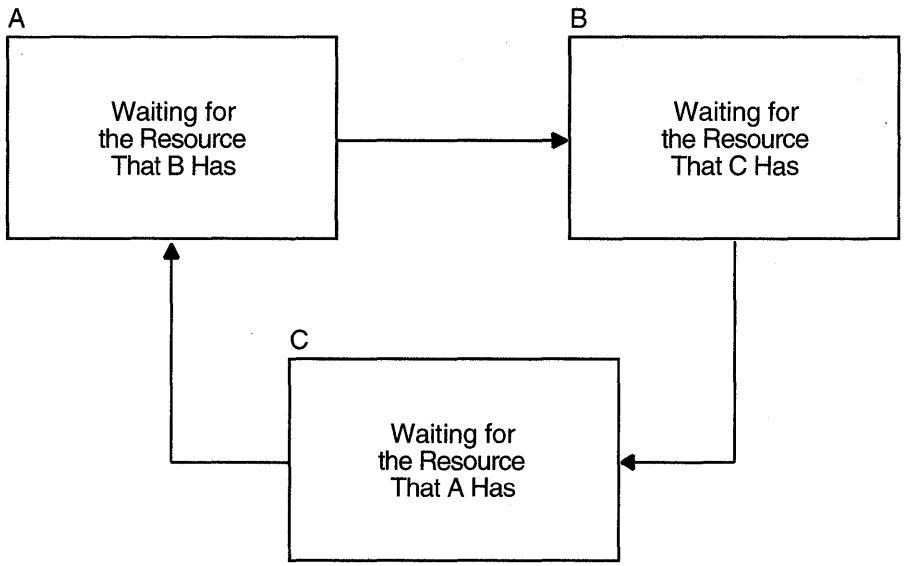
When a lock is dequeued or converted to a lower lock mode, the lock management services inspect the first conversion request on the conversion queue. The conversion request is granted if it is compatible with the locks currently granted. Any compatible conversion requests immediately following are also 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.

13.1.7 Deadlock Detection

A deadlock occurs when any group of locks are waiting for each other in a circular fashion.

In Figure 13–3, 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 13–3 A Deadlock



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.

13.2 Queuing Lock Requests

You use the $ENQ system service to queue lock requests. When you request new locks, the system service call must specify the lock mode, address of the lock status block, and resource name. The following example illustrates a call to $ENQ.

```
LKSB:   .BLKQ   0                       ; To contain lock status block
RESOURCE:
        .ASCID  /STRUCTURE_1/           ; STRUCTURE_1 is the name of
                                        ; the resource being locked
          .
          .
          .
        $ENQW_S  LKMODE=#LCK$K_PRMODE, - ; Protected read mode
                 LKSB=LKSB, -
                 RESNAM=RESOURCE
```

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 13–1 shows the compatibility of lock modes.

13.3 Advanced Locking Techniques

The previous sections discussed locking techniques and concepts useful to all applications. The following sections discuss specialized features of the VMS lock manager.

13.3.1 Synchronizing Locks

The $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 $QIO system services, described in Section 7.7.

The $ENQW macro performs synchronization by combining the functions of the $ENQ system service and the Synchronize ($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.

13.3.2 Notification of Synchronous Completion

The lock management services provide a mechanism that allows processes to determine if 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).

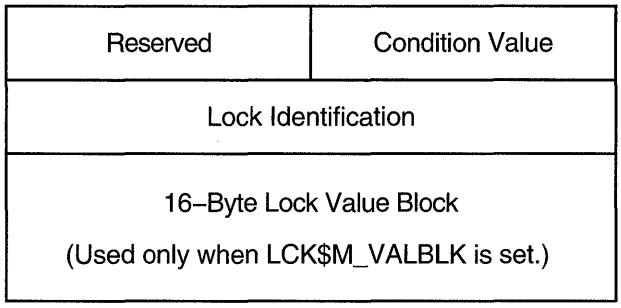
13.3.3 Expediting Lock Requests

A request can be expedited (granted immediately) if its requested mode (when granted) would not block any currently queued requests from being granted. The LCK$M_EXPEDITE flag is specified in the $ENQ operation to expedite a request. Currently, only NLMODE requests can be expedited. A request to expedite any other lock mode will fail with SS$_UNSUPPORTED status.

13.3.4 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 13–4). 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 ($DEQ) system service, see Section 13.4.

Figure 13–4 The Lock Status Block



ZK–0376–GE

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 13.5.1.

13.3.5 Blocking ASTs

In some applications that use the lock management services, a process must know if it is preventing another process from locking a resource. The lock management services inform processes of this through the use of blocking ASTs. To enable blocking ASTs, the **blkast** argument of the $ENQ system service must contain the address of a blocking AST service routine. When the lock prevents another lock from being granted, a blocking AST is delivered and the blocking AST service routine is executed. The **astprm** argument is used to pass a parameter to the blocking AST. For more information about ASTs and AST service routines, see Chapter 5. Some uses of blocking ASTs are described in Section 13.5.2.

13.3.6 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 may 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 higher-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 technique 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 $ENQ request.

Queuing Lock Conversions

To perform a lock conversion, a procedure calls the $ENQ system service with the flag bit LCK$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 example shows a simple lock conversion. Note that the lock must be granted before it can be converted.

```
LKSB:   .BLKQ   1
RESOURCE:
        .ASCID  /STRUCTURE_1/
          .
          .
          .
        $ENQW_S LKMODE=#LCK$K_NLMODE, - ; Null lock
                LKSB=LKSB, -
                RESNAM=RESOURCE
          .                        -----------
          . <--------------------- | Lock is |
          .                        | granted |
          .                        -----------
        $ENQW_S LKMODE=#LCK$K_PWMODE, - ; Protected write
                LKSB=LKSB, -            ; Lock ID is in LKSB
                FLAGS=#LCK$M_CONVERT    ; Conversion
          .
          .
          .
```

13.3.7 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 there are no conversions already queued.

The QUECVT behavior is valid only for a subset of all possible conversions. Table 13–2 defines the legal set of conversion requests for LCK$M_QUECVT. Illegal conversion requests are failed with SS$_BADPARAM returned.

Table 13–2 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 lock
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive lock

13.3.8 Parent Locks

When a lock request is queued, declaring a parent lock for the new lock is possible. When a lock has a parent, it is called a **sublock.** To specify a parent lock, the lock identification of the parent lock is passed in the **parid** argument to the $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 (files, for example), while higher-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. When updating all the records in a file, locking the whole file and updating the records without additional locking is faster and more efficient. But, when 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.

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 on SYSGEN and INTSTKPAGES, see the *Guide to Maintaining a VMS System*.

13.3.9 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 $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 lock 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 13–3 shows how lock conversions affect the contents of the process's and the resource's lock value block.

Table 13–3 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 lock
CR—Concurrent read
CW—Concurrent write
PR—Protected read
PW—Protected write
EX—Exclusive lock

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 ($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.

13.4 Dequeuing Locks

When a process no longer needs a lock on a resource, you can dequeue the lock by using the Dequeue Lock Request ($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. 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 $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 value lock block. If the lock being dequeued is in protected write or exclusive mode, the contents of the value block are stored in the value block associated with the resource. If the lock being dequeued is in any other mode, the value block is not used. The lock value block can be used only if a particular lock is being dequeued.

Three flags are available: LCKM_DEQALL, LCKM_CANCEL, and LCK$M_INVVALBLK.

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).

When LCK$M_CANCEL is specified, $DEQ attempts to cancel a lock conversion request that was queued by $ENQ. This attempt can succeed only if the lock request has not yet been granted, in which case the request is in the conversion queue. The LCK$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 $DEQ service in the *VMS System Services Reference Manual.*

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 $DEQ and $ENQ in the *VMS System Services Reference Manual* for more information on the LCK$M_INVVALBLK flag.

The following is an example of dequeuing locks.

```
LKSB:    .QUAD   0
RESOURCE:                               ; Resource is STRUCTURE_1
         .ASCID  /STRUCTURE_1/
           .
           .
           .
         $ENQ_S  LKMODE=#LCK$K_CRMODE, - ; Concurrent read mode
                 LKSB=LKSB, -
                 RESNAM=RESOURCE, -
                 ASTADR=READ_UPDATES
           .
           .
           .
         $DEQ_S  LKID=LKSB+4             ; LKSB+4 contains the lock ID
```

User mode locks are automatically dequeued when the image exits.

13.5 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 due to modifications by another process. In applications where modifications are infrequent, substantial I/O may be saved by maintaining local copies of buffers—hence, the names local buffer caching or distributed buffer management. Either the lock value block or blocking ASTs (or both) can be used to perform buffer caching.

13.5.1 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 prior 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 will find a version number mismatch and must read the latest copy from disk, rather than use its cached (now invalid) buffer.

13.5.2 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 alternate method of local buffer caching without using value blocks.

13.5.2.1 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 writing the buffer once. However, this prevents other processes from using the same disk block during this interval. This can be avoided if the process holding the exclusive mode lock has a blocking AST. The AST will notify 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.

13.5.2.2 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.

13.5.3 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, while 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 speaking, blocking ASTs are preferred in a low-contention environment, while version numbers are preferred in a high-contention environment. You may 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, blocking ASTs are used; if there is no reason to expect a quick reuse, version numbers are used.

In an **adaptive** strategy, an application makes evaluations 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, consider the case where one process continually displays the state of a database, while another occasionally updates it. If version numbers are used, the displaying process must always check to see 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.

13.6 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.

14

DECdtm Services

DECdtm services provide for complete and consistent execution of distributed transactions on the VMS operating system. In transaction processing applications, there can be many users simultaneously making inquiries and updating a collection of shared data, generally a database. Transactions typically involve communication between a network of systems distributed at various geographic locations; therefore, the operations can be collectively referred to as distributed transaction processing.

The following are DECdtm services:

- Start Transaction ($START_TRANS)
- Start Transaction and Wait ($START_TRANSW)
- End Transaction ($END_TRANS)
- End Transaction and Wait ($END_TRANSW)
- Abort Transaction ($ABORT_TRANS)
- Abort Transaction and Wait ($ABORT_TRANSW)

14.1 Using Transaction Management System Services

Application programs can call the VMS transaction management system services to delimit the set of operations that make up distributed transaction. These system services can then guarantee consistent execution of the transaction. See the *VMS System Services Reference Manual* for a complete description of DECdtm services.

You must call these services in your application program according to the syntax rules for the programming language that you are using. Refer to the appropriate language reference manual for more information on using system services.

14.1.1 Transaction Processing System Model

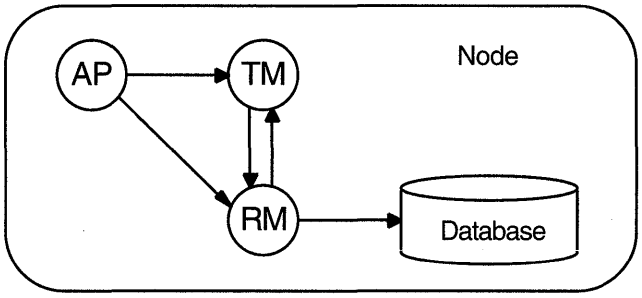
In Digital's model for transaction processing, several components work together to execute atomic transactions.

At the end-user level, user-written **application programs** (AP) define the task to be accomplished, such as query, update, and insertion, as well as how transactions are to be executed. The application programs initiate transaction execution using calls to VMS system services.

At the system level, the execution of the transaction depends on the interaction of **resource managers** (RMs) and **transaction managers** (TMs).

The interaction of these components is shown in Figure 14–1.

Figure 14–1 Transaction Processing Components



ZK-1869A-GE

The key function of the DECdtm services is to act as transaction manager. A transaction manager supports the transaction management system services that are issued from application programs to delimit transactions. To complete or abort a transaction, the transaction manager sends instructions to resource managers and other transaction managers involved in the transaction. In this way, a transaction manager coordinates the actions of a transaction.

Through calls to system services, application programs communicate directly with the DECdtm services. Additionally, these programs can use the services provided by a resource manager.

A **resource manager** is a software product that manages shared access to a set of recoverable resources on behalf of application programs. In this context, recoverable means that all updates to the resources on behalf of the transaction can be made permanent or can be undone. Recoverable resources typically include files or databases. The resource manager participates in the two-phase commit protocol to commit or abort a transaction.

14.1.2 Transaction Management

The responsibilities of a transaction manager include the following:

- Delimiting transactions
- Tracking participating transaction managers and resource managers
- Ensuring that transactions either commit or abort
- Assisting in recovery of resources after failures

For every transaction that it coordinates, the transaction manager in the DECdtm services maintains a list of the transaction's **participants**. Participants can include:

- Resource managers on a local node, spanning one or more processes
- Transaction managers on other nodes within a network, which might also have associated resource manager and transaction manager participants

The transaction manager uses this list of participants to execute the two-phase commit protocol. During the execution of this protocol, each participating transaction manager writes transaction information to a log file. A log file contains a permanent record of transaction states. By having access to a log file, a transaction manager can resume the execution of the two-phase commit

protocol after recovering from a system failure. When executing the two-phase commit protocol, the transaction manager tells the transaction's participants whether to commit or abort a transaction.

14.1.3 Starting a Transaction

Transaction management services demarcate transactions. To indicate the start of transaction operations, an application program calls $START_TRANS or its synchronous equivalent, $START_TRANSW.

The application program should make a call to $START_TRANS prior to the code making up the transaction operations and prior to any code that accesses recoverable resources or remote nodes. In response to a call to $START_TRANS, the transaction manager component of the DECdtm services generates a unique **transaction identifier** (TID) for the transaction so that it can keep track of the transaction. The transaction manager uses the TID to identify all actions performed by resource managers and transaction managers on behalf of the transaction.

For each process on which they are used, the DECdtm services maintain the concept of a current transaction. The transaction that is started using $START_TRANS is considered the process default, or current, transaction. Alternatively, the NONDEFAULT flag can be set when $START_TRANS is called to establish a nondefault transaction.

Thus, when an application program that is using a resource manager such as RMS Journaling makes a call to $START_TRANS, the TID of the current transaction is used by default. For RMS Journaling, unless a specific TID is specified (using the XAB$_TID item code), RMS associates the record stream with the default, current transaction.

The following FORTRAN code fragment demonstrates the use of $START_TRANSW. The program first determines the accounts to be credited and debited and the amount to be transferred. It then calls $START_TRANSW to indicate to the transaction manager that it is beginning the set of debit and credit operations that make up the distributed transaction.

```
INTEGER*4 STATUS, TID (4)
INTEGER*2 IOSB (4)

INTEGER*4 SYS$START_TRANSW
   .
   .
   .
GET_INPUT('Account to debit', DEBIT_ACCT)
GET_INPUT('Account to credit', CREDIT_ACCT)
GET_INPUT('Amount to transfer', TRANSFER_AMT)

STATUS=SYS$START_TRANSW (%VAL (0),
1                        %VAL (0),
2                        IOSB,
3                        %VAL (0),
4                        %VAL (0),
5                        TID)

IF (STATUS) STATUS = IOSB (1)

IF (.NOT.STATUS) GOTO 100

STATUS = DEBIT_ACCOUNT (
1       DEBIT_ACCT, TRANSFER_AMT, %REF(0))

STATUS = CREDIT_ACCOUNT (
1       CREDIT_ACCT, TRANSFER_AMT, %REF(0))
   .
   .
   .
```

14.1.4 Completing a Transaction

The processing of a transaction completes when a call is made to the DECdtm system services to either commit or abort. The system services that end a transaction and begin commit processing are $END_TRANS and its synchronous equivalent, $END_TRANSW. The services that abort a transaction are $ABORT_TRANS and its synchronous equivalent, $ABORT_TRANSW.

In response to an $END_TRANS call, the DECdtm transaction manager initiates a commit protocol to inform all the transaction's participants to start commit processing.

$END_TRANS can be called only by the same process that called the $START_TRANS service.

Note that a call to $END_TRANS does not guarantee transaction commitment. A transaction could fail for a number of reasons. For example, if the **timout** argument has been specified when calling the $START_TRANS service, then the transaction will be aborted if the transaction exceeds the time specified.

$END_TRANS returns a failure status (SS$_ABORT) if a condition occurs that makes it impossible to commit the transaction. In this event an abort reason code is returned as a second longword in the I/O status block (IOSB).

The following FORTRAN code fragment demonstrates the use of $END_TRANSW. After the final operation of the program is issued, the program calls $END_TRANSW to commit the transaction.

```
   .
   .
   .
STATUS = CREDIT_ACCOUNT (
1       CREDIT_ACCT, TRANSFER_AMT, %REF(0))

STATUS = SYS$END_TRANSW (%VAL (0),
1                        %VAL (0),
2                        IOSB,
3                        %VAL (0),
4                        %VAL (0),
5                        TID)

IF (STATUS) STATUS = IOSB (1)

IF (.NOT.STATUS) GOTO 100

   .
   .
   .
END
```

14.1.5 Calling a Planned Abort

$ABORT_TRANS and its synchronous equivalent, $ABORT_TRANSW, enable applications to implement a planned abort. If errors occur during the execution of the transaction processing, a call can be made to $ABORT_TRANS to end the transaction so that previous changes do not become permanent in the accessed database.

When an application calls the $ABORT_TRANS system service to abort a transaction, it can supply an abort reason code in the **reason** parameter to specify the reason why the transaction is to be aborted. Similarly, a resource manager which casts a "veto" vote may specify an abort reason code. The abort reason code is returned in the I/O status block (IOSB) for $ABORT_TRANS and $END_TRANS.

Note that the IOSB is not filled in when a return code of SS$_SYNCH, indicating successful synchronous completion, is returned. The SS$_SYNCH return code is returned only if the DDTM$M_SYNC flag is set. Therefore, if you want the abort reason code to be stored in the IOSB, do not set the DDTM$M_SYNC flag.

The following code fragment is from a COBOL application that calls $ABORT_TRANSW as part of its error handling:

```
   .
   .
   .
    DISPLAY "Calling subtransaction to FETCH record from database." LINE PLUS 1.

    CALL "ERASE_EAST" USING WS-EMP-KEY WS-EMP-RECORD WS-STATUS TID.
    IF WS-STATUS IS NOT EQUAL TO "SUCCESS"
       PERFORM ABORT-GLOBAL-TRANSACTION
       GO TO END-MOVE-EAST-WEST.
   .
   .
   .
ABORT-GLOBAL-TRANSACTION.

*   The employee name field contains information about the error
*   detected in the subprogram.

    DISPLAY WS-EMP-NAME LINE PLUS 1.
    DISPLAY "Aborting global transaction." LINE PLUS 1.
```

```
*   abort (rollback) global transaction
    CALL "SYS$ABORT_TRANSW" USING
      OMITTED
      OMITTED
      BY REFERENCE IOSB
      OMITTED
      OMITTED
      BY REFERENCE TID
          GIVING WS-SYS-STATUS.
```

14.1.6 Example of Using Transaction Management System Services

Example 14–1 is a BLISS program that uses the transaction management services to create two simple transactions. The first transaction is committed, using $END_TRANS. The second transaction is aborted, using $ABORT_TRANS.

Example 14–1 Using Transaction Management Services

```
MODULE EXAMPLE (MAIN=EXAMPLE) =
BEGIN

    LIBRARY'SYS$LIBRARY:STARLET';

ROUTINE EXAMPLE =
BEGIN

    LOCAL
        STATUS,
        IOSB : VECTOR [4, WORD],
        TID  : $BBLOCK [DTI$S_TID],
        BINARY_TIMEOUT : VECTOR [8, BYTE] ;

    !+
    ! Convert 10 seconds to a VMS time
    !-

❶   $BINTIM (TIMBUF = %ASCID '0000 00:00:10.00',
             TIMADR = BINARY_TIMEOUT);

    IF NOT .STATUS THEN RETURN (.STATUS);

    !+
    ! Start a nondefault process transaction
    !-

❷    STATUS = $START_TRANSW (EFN    = 1,
                             FLAGS  = (DDTM$M_NONDEFAULT OR
                                       DDTM$M_SYNC),
                             IOSB   = IOSB,
                             ASTADR = 0,
                             ASTPRM = 0,
                             TID    = TID,
                             TIMOUT = BINARY_TIMEOUT);

    IF .STATUS AND (.STATUS NEQU SS$_SYNCH) THEN

        STATUS = .IOSB [0];

    IF NOT .STATUS THEN RETURN (.STATUS);

    !+
    ! Commit the transaction
    !-
```

(continued on next page)

Example 14–1 (Cont.) Using Transaction Management Services

```
❸   STATUS = $END_TRANSW (EFN    = 1,
                          FLAGS  = DDTM$M_SYNC,
                          IOSB   = IOSB,
                          ASTADR = 0,
                          ASTPRM = 0,
                          TID    = TID);

    IF .STATUS AND (.STATUS NEQU SS$_SYNCH) THEN

        STATUS = .IOSB [0];


    !+
    ! If the transaction aborted, extract the abort reason code and
    ! return that.
    !-

❹   IF .STATUS EQLU SS$_ABORT THEN

          STATUS = .IOSB<32,32,0> ;

    IF NOT .STATUS THEN RETURN (.STATUS);

    !+
    ! Start another nondefault process transaction
    !-

❺     STATUS = $START_TRANSW (EFN    = 1,
                              FLAGS  = (DDTM$M_NONDEFAULT OR
                                        DDTM$M_SYNC),
                              IOSB   = IOSB,
                              ASTADR = 0,
                              ASTPRM = 0,
                              TID    = TID);

    IF .STATUS AND (.STATUS NEQU SS$_SYNCH) THEN

        STATUS = .IOSB [0];

    IF NOT .STATUS THEN RETURN (.STATUS);

    !+
    ! Abort the transaction
    !-

❻         STATUS = $ABORT_TRANSW (EFN    = 1,
                                  FLAGS  = DDTM$M_SYNC,
                                  IOSB   = IOSB,
                                  ASTADR = 0,
                                  ASTPRM = 0,
                                  TID    = TID);

    IF .STATUS AND (.STATUS NEQU SS$_SYNCH) THEN

        STATUS = .IOSB [0];

    !+
    ! If the transaction aborted, extract the abort reason code and
    ! return that.
    !-
```

(continued on next page)

Example 14–1 (Cont.) Using Transaction Management Services

```
❼  IF .STATUS EQLU SS$_NORMAL THEN
       STATUS = .IOSB<32,32,0> ;
   RETURN (.STATUS);
END;
END
ELUDOM
```

❶ This code sets the value if BINARY_TIMEOUT at 10 seconds.

❷ A call to $START_TRANS to start a transaction and set the transaction time-out value. The DECdtm transaction manager responds to this call by creating a transaction identifier.

❸ To commit the transaction, the application calls $END_TRANS.

❹ For $END_TRANS, a return value of SS$_ABORT indicates that the transaction aborted during processing. When this value is returned, the abort reason code is available in the IOSB. This code retrieves the abort reason code from the IOSB if the transaction is aborted.

❺ To start another transaction, the application makes another call to $START_TRANS.

❻ To abort the transaction, the application calls $ABORT_TRANS.

❼ For $ABORT_TRANS(W), a return value of SS$_NORMAL indicates that the service completed successfully, that is, the transaction was aborted. This code retrieves the abort reason code from the IOSB upon successful completion of $ABORT_TRANSW. You must not set the DDTM$M_SYNC flag if you want the abort reason code to be stored in the IOSB.

15

Programming Examples

This chapter presents three VAX MACRO programs: ORION, CYGNUS, and LYRA. These programs do not perform any practical operations; they are intended only to illustrate how to call various system services.

Each program is preceded by an introduction identifying the services it uses and the main functions it performs. The programs themselves contain many comments related to specific data definitions and portions of code.

15.1 ORION Program Example

The program ORION uses the following system services:

$ASSIGN	Assign I/O Channel
$QIOW	Form of Queue I/O Request and Synchronize
$NUMTIM	Convert Binary Time to Numeric Time
$BINTIM	Convert ASCII String to Binary Time
$SETIMR	Set Timer
$WAITFR	Wait for Single Event Flag
$READEF	Read Event Flags
$SETPRN	Set Process Name

This sample program illustrates the following:

1. Assigning an I/O channel to a terminal and writing messages to the terminal. The device name is specified by the logical name TERMINAL. Before ORION is run, the logical name must be assigned an equivalence device name.
2. Using the $NUMTIM system service to determine whether the current time is before or after noon. A call to $SETIMR is made conditionally if the time is prior to noon.
3. Obtaining a delta time value in the system format to use as input to the Set Timer ($SETIMR) system service.
4. Calling the Set Timer system service.
   a. Event flag—The $SETIMR call is followed by a wait for the specified event flag. When the timer expires, the program calls $READEF and displays the current status of the event flag cluster.
   b. AST routine—One AST routine is for a delta time interval. The other (conditional) is for an absolute time. In either case, the program continues execution and will be interrupted when the timer requests are processed.
5. An example of terminal input. The program prompts for a character string to be used as the process name of the current process. Then it uses this name as input to the $SETPRN system service.

```
        .TITLE  ORION SYSTEM SERVICES TEST
        .IDENT  /01/

; Macro library calls

        $IODEF                          ; Define I/O function codes
        $SSDEF                          ; Define system status values
        $READEFDEF                      ; Define offsets for $READEF

; Local macro defined in private macro library

;       MESSAGE Output messages formatted by FAO

        .MACRO  MESSAGE
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32
        BSBW    ERROR
        .ENDM   MESSAGE

; Read-only data program section

        .PSECT  RODATA,NOWRT,NOEXE

; Local Read/Write Data

TTNAME: .ASCID  /TERMINAL/              ; Terminal logical name

; FAO control strings and data for timer (AST and event flag) tests

ASCNOON:
        .ASCID  /-- 12:00:00.00/        ; Noon in ASCII format
TENSEC: .ASCID  /0 00:00:10/            ; Ten seconds delta time in ASCII format
DISPLAYEFN:                             ; Display cluster contents
        .ASCID  /CLUSTER 2 CONTENTS: !XL/
TIMSTR:                                 ; Display message after event flag wait
        .ASCID  "!/TIMER ENTRY PROCESSED; CLUSTER 2 = !XL"
NOONMSG:                                ; Display message at noon
        .ASCIC  /I'M YOUR TIME AST ROUTINE; IT'S NOON.../
SECMSGDESC:                             ; Display message from AST routine
        .ASCID  "!/TIME AST ROUTINE; DELTA TIME !%T"
TWENTY: .LONG   -10*1000*1000*20,-1     ; 20 seconds delta time

; Announcement messages

FAOSTR:                                 ; Master control string
        .ASCID  "!/ORION: !AC "         ; Name, message

; Announcement messages and lengths for outputting
```

```
HELLO:  .ASCII  /HELLO...MY NAME IS ORION.../
HELLOLEN:
        .LONG   HELLOLEN-HELLO
;
TIMERMSG:
        .ASCII  /BEGINNING TIMER TESTS.../
TIMERLEN:
        .LONG   TIMERLEN-TIMERMSG
;
EFNWAITMSG:
        .ASCII  /TIMER SET; WAIT TEN SECONDS/
EFNWAITLEN:
        .LONG   EFNWAITLEN-EFNWAITMSG
;
ASTWAITMSG:
        .ASCII  /TIMER SET; AST IN 20 SECONDS/
ASTWAITLEN:
        .LONG   ASTWAITLEN-ASTWAITMSG

; Prompt for terminal input

PROMPT: .ASCII  /ENTER 1-15 CHARACTER NAME FOR PROCESS:/
PROMPTLEN:
        .LONG   PROMPTLEN-PROMPT

; Error message control strings

; ERRSTR    formats error message if a system service fails
; IOERRSTR  formats error message if I/O fails
; BADASTSTR formats error message if error in AST routine

ERRSTR:
        .ASCID  "!/SYSTEM SERVICE ERROR AT APP. !XL R0=!XL"

IOERRSTR:
        .ASCID  "!/I/O ERROR; IOSB !XW"

BADASTSTR:
        .ASCID  /BAD AST PARAMETER  !UL/

WAKEUP: .ASCII  /AWAKENED.../
WAKEUPLEN:
        .LONG   WAKEUPLEN-WAKEUP

; Read/write data

        .PSECT  RWDATA,RD,WRT,NOEXE

; FAO control string and buffer for all announcement messages

FAODESC:
        .LONG   80                      ; Descriptor for FAO output buffer
        .ADDRESS -
                FAOBUF                  ; Address of buffer
FAOBUF: .BLKB   80                      ; FAO buffer
FAOLEN: .WORD   0                       ; Length of final string, always
        .WORD   0                       ; need longword for $QIOW

; Buffer to format messages from AST routine; a separate output buffer
; ensures that if the AST is delivered while another message is being
; written into the FAO output buffer, no data or message will be lost.
```

```
FASTDESC:
        .LONG   80                      ; Descriptor for FAO output buffer
        .ADDRESS -
                FASTBUF                 ; Address of buffer
FASTBUF:
        .BLKB   80                      ; FAO buffer
FASTLEN:
        .WORD   0                       ; Length of final string, always
        .WORD   0                       ; need longword for $QIOW

; Receive channel number assigned to terminal and I/O status here

TTCHAN: .BLKW   1                       ; Terminal channel

TTIOSB:                                 ; IOSB for terminal input
        .BLKW   1                       ; Return status
TTLEN:  .BLKW   1                       ; Length of I/O
        .BLKL   1                       ; Device char

; Argument list for $NAME_G form of a system service call

READLST:
        $READEF EFN=32, -
                STATE=EFNTEST

; Buffer to obtain numeric values of components of time. Since
; the only field of interest is the hours field, the remaining
; fields in the buffer are not formatted.

TIMES:  .BLKW   3                       ; Year, month, day
HOURS:  .BLKW   1                       ; Current time in hours
        .BLKW   3                       ; Remainder of buffer

; Buffer for terminal input (will create input descriptor for
; $SETPRN system service)

NAMEDESC:                               ; Descriptor setup
        .LONG   15                      ; Initial size of buffer
        .ADDRESS -
                NAME                    ; Address of buffer
NAME:   .BLKB   15                      ; Name string here


; Fields for timer tests

NOON:   .BLKQ   1                       ; Will contain 12:00 in system format
TEN:    .BLKQ   1                       ; Will contain 10 second delta time
EFNTEST:
        .LONG   0                       ; Receive status of event flags
EFNTEST2:
        .LONG   0                       ; Status after timer test

; Longword to save PC on entry to error handling subroutine

SAVEPC: .BLKL   1

; Code begins here.

        .PSECT  TIMER,EXE,NOWRT
        .ENTRY  ORION,^M<R2,R3,R4,R5,R6> ; Entry mask

; Assign an I/O channel to the device specified by the logical name
; TERMINAL and issue a message indicating we're off and running.
; Do not perform normal error checking here; instead, let the
; command interpreter issue a message based on the status in R0
; if the channel assignment fails.
```

```
SETUP:
        $ASSIGN_S -
                DEVNAM=TTNAME, -
                CHAN=TTCHAN
        BLBS    R0,10$                  ; All okay, continue
        RET                             ; Otherwise exit with status in R0
;
10$:    $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=HELLO, -
                P2=HELLOLEN, -
                P4=#32
        BSBW    ERROR

; Call Read Event Flags to get status of event flags before beginning
; tests and use FAO to output the contents of local event flag cluster 2

        $READEF_G -
                READLST
        BSBW    ERROR
        $FAO_S  CTRSTR=DISPLAYEFN, -
                OUTBUF=FAODESC, -
                OUTLEN=FAOLEN,-
                P1=EFNTEST
        BSBW    ERROR
        MESSAGE                         ; Use MESSAGE MACRO

; Announce start of timer tests

TIMETEST:
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=TIMERMSG, -
                P2=TIMERLEN, -
                P4=#32
        BSBW    ERROR

; Call $NUMTIM to find out if it is currently AM or PM. If
; the program is being run in the AM (any time), we'll call
; $SETIMR to notify us via an AST when the time rolls over
; to afternoon. If it's already PM, skip this setting of
; the timer.

        $NUMTIM_S -
                TIMBUF=TIMES
        BSBW    ERROR
        CMPW    HOURS,#12               ; Before or after noon?
        BGEQ    10$                     ; After or noon, skip setting timer

; Fall through here: format ASCII string representing 12 noon
; into system quadword time format and call $SETIMR with
; the address of AST service routine to handle timer requests.

        $BINTIM_S -                     ; Get binary noon time
                TIMBUF=ASCNOON, -
                TIMADR=NOON
        BSBW    ERROR                   ; Error check

        $SETIMR_S -
                DAYTIM=NOON, -
                ASTADR=TIMEAST, -
                REQIDT=#12
        BSBW    ERROR                   ; Error check

; Now, get a delta time of 10 seconds formatted into a quadword
```

```
10$:    $BINTIM_S -                     ; Get binary delta time
                TIMBUF=TENSEC, -
                TIMADR=TEN
        BSBW    ERROR                   ; Error check
        $SETIMR_S -                     ; Set timer (ten seconds)
                EFN=#33, -
                DAYTIM=TEN
        BSBW    ERROR                   ; Error check

; Announce wait for event flag and wait; then read the
; event flag cluster and output its contents

        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=EFNWAITMSG, -
                P2=EFNWAITLEN, -
                P4=#32
        $WAITFR_S -
                EFN=#33                 ; Now wait
        BSBW    ERROR                   ; Error check

; Update argument list for $READEF and then call it with new address
; to write the cluster into. When complete, format a message and
; display the contents of the cluster.

        MOVAL   EFNTEST2,READLST+READEF$_STATE
        $READEF_G -
                READLST
        BSBW    ERROR                   ; Error check
        $FAO_S CTRSTR=TIMSTR, -
                OUTLEN=FAOLEN, -
                OUTBUF=FAODESC,-
                P1=EFNTEST2
        BSBW    ERROR                   ; Error check
        MESSAGE

; Announce setting of timer with AST in 20 seconds (using
; alternate method of specifying delta time). Then, set timer
; and continue.

        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=ASTWAITMSG, -
                P2=ASTWAITLEN, -
                P4=#32

        $SETIMR_S -
                DAYTIM=TWENTY, -
                ASTADR=TIMEAST, -
                REQIDT=#20
        BSBW    ERROR                   ; Error check

; Issue a prompt for terminal input: request a name for the current
; process and then use the character string entered as the process
; name.

RDNAME:
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=PROMPT, -
                P2=PROMPTLEN, -
                P4=#32
        BSBW    ERROR                   ; Error check
```

```
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_READVBLK, -
                IOSB=TTIOSB, -
                P1=NAME, -
                P2=NAMEDESC
        BSBW    ERROR

        CMPW    TTIOSB,#SS$_NORMAL      ; I/O successful?
        BEQL    10$                     ; Yes, go on
        $FAO_S  CTRSTR=IOERRSTR, -
                OUTLEN=FAOLEN, -
                OUTBUF=FAODESC,-
                P1=TTIOSB
        MESSAGE
        BRW     RDNAME                  ; Go try again
10$:    MOVZWL  TTLEN,NAMEDESC          ; Update descriptor length
        $SETPRN_S -
                PRCNAM=NAMEDESC         ; Set process name
        BSBW    ERROR

; Hibernate. When ORION is run interactively, the terminal is dormant.
; When the AST for the Set Timer service is delivered, ORION
; will awaken long enough to execute the AST service routine and
; then resume execution.

; If ORION is run in a subprocess, wakeups can be scheduled for
; delta time intervals. Each time it is awakened, ORION displays a
; message and then resumes hibernating.

HIB:    $HIBER_S                        ; Hibernate for now
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=WAKEUP, -
                P2=WAKEUPLEN, -
                P4=#32
        BRB     HIB
        RET

; AST routine to handle timer requests

        .ENTRY  TIMEAST,^M<>            ; Entry mask for timer AST routine
        CMPL    #12,4(AP)               ; Is it noon AST?
        BEQL    10$                     ; Yes, go do it
        CMPL    #20,4(AP)               ; Is it delta time AST?
        BEQL    20$                     ; Yes, go do that
        BRW     30$                     ; Neither, issue error message

; Format message for noon AST

10$:    $FAO_S  CTRSTR=FAOSTR, -
                OUTBUF=FASTDESC, -
                OUTLEN=FASTLEN, -
                P1=#NOONMSG
        BSBW    ERROR                   ; Error  check
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
        BSBW    ERROR                   ; Error check
        RET

; Format message for 20 second AST
```

```
20$:    $FAO_S  CTRSTR=SECMSGDESC, -
                OUTBUF=FASTDESC, -
                OUTLEN=FASTLEN,-
                P1=#TWENTY
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
        RET

; Format message if spurious AST

30$:    $FAO_S  CTRSTR=BADASTSTR, -
                OUTLEN=FASTLEN, -
                OUTBUF=FASTDESC,-
                P1=4(AP)
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
        RET

; Error-handling routine: checks status code in R0.
; If low bit set, returns to mainline routine. Otherwise,
; displays approximate PC and R0 when system service call
; encounters an error and issues RET that causes image exit.

ERROR:
        BLBC    R0,10$                  ; If error, branch
        RSB                             ; Otherwise, continue

; Use FAO to format output error message

10$:    MOVL    (SP),SAVEPC
        $FAO_S  CTRSTR=ERRSTR, -
                OUTLEN=FAOLEN, -
                OUTBUF=FAODESC,-
                P1=SAVEPC, -
                P2=R0
        BLBC    R0,END
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32
END:    RET
        .END    ORION
```

15.2 CYGNUS Program Example

The program CYGNUS uses the following system services:

$ASSIGN	Assign I/O Channel
$DCLEXH	Declare Exit Handler
$CREMBX	Create Mailbox
$GETDVI	Get Device/Volume Information
$CREPRC	Create Process
$FAO	Formatted ASCII Output
$QIO	Queue I/O Request

$CRELNM	Create Logical Name
$WAKE	Wake Process
$SETSFM	Set System Service Failure Exception Mode
$WAITFR	Wait for Single Event Flag
$DELLNM	Delete Logical Name
$DASSGN	Deassign I/O Channel

This sample program illustrates the following:

1. Assigning a channel to the current output device assigned to the logical name SYS$OUTPUT.
2. Declaring an exit handler to receive control at image exit. The exit handler ensures that the image exits efficiently.
3. Creating a mailbox and using the $GETDVI system service to obtain the unit number.
4. Obtaining the logical name translation of SYS$OUTPUT, and checking for a concealed device name, by using the $GETDVI system service.
5. Creating a subprocess and using the mailbox created as a termination mailbox. When the subprocess terminates, an AST service routine interprets the message.
6. Placing names in the group logical name table.
7. Waking a hibernating subprocess. The subprocess created by this program places itself in hibernation after starting up. When awakened, it translates the logical names placed in the group logical name table.

```
        .IDENT /01/

; System macro definitions required by CYGNUS

        $SSDEF                  ; Define status codes for returns
        $IODEF                  ; Define I/O function codes for $QIO
        $MSGDEF                 ; Define names for mailbox messages
        $PQLDEF                 ; Define names for quota list
        $ACCDEF                 ; Define names for termination message
        $DIBDEF                 ; Define names for device information buffer
        $DVIDEF                 ; Define item codes for device information
        $LNMDEF                 ; Define item codes for logical names

; Local macros:

;        MESSAGE,  to output messages formatted by FAO

        .MACRO  MESSAGE
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN
                P4=#32
        BSBW    ERROR
        .ENDM   MESSAGE

; GRPNAME, to place logical name/equivalence name
; pairs in the group logical name table with $CRELOG and
; to do error checking.
```

```
        .MACRO  GRPNAME LOGICAL,EQUAL
        MOVW    EQUAL,CREITM
        MOVL    EQUAL+4,CREBF
        $CRELNM_S -
                TABNAM=GRPTBL, -
                LOGNAM=LOGICAL, -
                ITMLST=CREITM
        BSBW    ERROR
        .ENDM   GRPNAME

; Read-only data program section

        .PSECT  RODATA,NOWRT,NOEXE

; Descriptor for input logical name

OUTPUT: .ASCID  /SYS$OUTPUT/

; Descriptor for group logical name table
GRPTBL: .ASCID      /LNM$GROUP/

; Buffers for announcement messages and lengths

HELLO:  .ASCID  /CYGNUS...HELLO/
HELLOLEN:
        .LONG   HELLOLEN-HELLO
;
BYE:    .ASCII  /CYGNUS EXIT HANDLER.../
BYELEN: .LONG   BYELEN-BYE

; Control strings for output messages formatted by FAO and associated
; counted ASCII strings to insert in messages

PRCSTR:
        .ASCID  /LYRA CREATED, PID !XL/ ; Display PID of subprocess
ASTERRSTR:
        .ASCID  "!/MAILBOX MESSAGE HAS !AC !XW"
IOERR:  .ASCIC  'I/O ERROR'             ; I/O error in AST routine
IDERR:  .ASCIC  /BAD MSG ID/            ; Mailbox message not
                                        ;  termination message
PIDERRSTR:
        .ASCID  "!/SPURIOUS PROCESS ID !XL IN DELETION MAILBOX"
DONESTR:
        .ASCID  "!/LYRA COMPLETED; STATUS !XL TIME !%T"
BADEXSTR:
        .ASCID  "!/EXIT DUE TO ERROR !XL"

; Item list for $GETDVI to find unit number of mailbox
;
MBX_DVILIST:                            ; Begin $GETDVI item list
        .WORD   4                       ; Maximum of 4 bytes long
        .WORD   DVI$_UNIT               ; Item code for unit number
        .ADDRESS -
                UNIT_NUMBER             ; Address of buffer
        .LONG   0                       ; No return length needed
        .LONG   0                       ; End item list

; Item list for $GETDVI finding logical name translation of SYS$OUTPUT

TERM_DVILIST:                           ; Begin $GETDVI item list
        .WORD   64                      ; Maximum of 64 bytes long
        .WORD   DVI$_DEVNAM             ; Item code for device name
        .ADDRESS -
                TERM                    ; Destination of terminal name
        .ADDRESS -
                TERM_DESC               ; Destination of length of string
        .LONG   0                       ; End item list
```

```
; Descriptor to define name of image for subprocess to execute.

LYRAEXE:
        .ASCID  /LYRA.EXE/

; Quota list for subprocess: defines minimal quotas required
; for the subprocess to execute and ensures that the creating
; image will have sufficient quotas to continue.

QLIST:  .BYTE   PQL$_BYTLM              ; Buffer quota
        .LONG   1024
        .BYTE   PQL$_FILLM              ; Open file quota
        .LONG   3
        .BYTE   PQL$_PGFLQUOTA          ; Paging file quota
        .LONG   256
        .BYTE   PQL$_PRCLM              ; Subprocess quota
        .LONG   1
        .BYTE   PQL$_TQELM              ; Timer queue quota
        .LONG   3
        .BYTE   PQL$_LISTEND


; Logical name/equivalence name pairs for group table.
; Note that one of the names in the table is nested.

ORION:  .ASCID  /ORION/
HUNTER: .ASCID  /HUNTER/
PEGASUS:.ASCID  /PEGASUS/
HORSE:  .ASCID  /HORSE/
LYRA:   .ASCID  /LYRA/
HARP:   .ASCID  /HARP/
CYG:    .ASCID  /CYGNUS/
SWAN:   .ASCID  /SWAN/
DUCK:   .ASCID  /UGLY DUCKLING/
TALE:   .ASCID  /FAIRY TALE!/


; Read/write data program section

        .PSECT  RWDATA,RD,WRT,NOEXE

UNIT_NUMBER:
        .LONG   0                       ; Destination of unit number

TERM_DESC:
        .LONG   64                      ; Maximum of 64 bytes
TERM_ADDRESS:
        .ADDRESS -
                TERM
;
CONC_TERM:
        .ASCII  /_/                     ; 2nd underscore for concealed device
TERM:   .BLKB   64                      ; Terminal name is placed here


TTCHAN: .BLKW   1                       ; Channel number of terminal

; $CRELNM item list
; This list is filled in for each invocation of the GRPNAME macro

CREITM: .WORD   0                      ; Equivalence length
        .WORD   LNM$_STRING            ; Item code
CREBF:  .LONG   0                      ; Equivalence buffer
        .LONG   0                      ; No return length
        .LONG   0                      ; List terminator

; Termination control block
```

```
EXITBLOCK:                              ; Exit control block
        .BLKL   1                       ; System uses this for pointer
        .ADDRESS -
                EXITRTN                 ; Address of routine
        .LONG   2                       ; Number of arguments for handler
        .ADDRESS -
                STATUS                  ; Address to store status
ERRPC:  .BLKL   1                       ; Store PC (if error)
STATUS: .BLKL   1                       ; Status code at exit

; Fields used for termination mailbox creation, message buffering

EXCHAN: .BLKW   1                       ; Channel number of mailbox
MBXIOSB:                                ; I/O status block
        .BLKW   1                       ; Status of I/O completion
MBLEN:  .BLKW   1                       ; Length of operation here
MBPID:  .BLKL   1                       ; PID of process deleted

EXITMSG:                                ; Buffer for mailbox message
        .BLKB   ACC$K_TERMLEN

; Receive PID of subprocess here

LYRAPID:
        .BLKL   1

; Output buffers for strings formatted by FAO

FAODESC:                                ; Descriptor for output buffer
        .LONG   80                      ; 80-character buffer
        .ADDRESS -
                FAOBUF                  ; Address
FAOBUF: .BLKB   80                      ; Buffer
FAOLEN: .BLKW   1                       ; Receive length here
        .BLKW   1                       ; Need longword for $QIO

; Need separate FAO buffers for use in AST routine to ensure
; that data doesn't get clobbered asynchronously

FASTDESC:
        .LONG   80                      ; Length
        .ADDRESS -
                FASTBUF                 ; Address
FASTBUF:
        .BLKB   80                      ; Buffer
FASTLEN:
        .BLKW   1                       ; Get length
        .BLKW   1                       ; Need longword for $QIO


; Program code begins here.

        .PSECT  CODE,EXE,RD,NOWRT
        .ENTRY  CYGNUS,^M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>

; Call $ASSIGN to assign an I/O channel to device assigned to SYS$OUTPUT
; and issue message verifying successful initialization

10$:    $ASSIGN_S -
                DEVNAM=OUTPUT, -
                CHAN=TTCHAN
        BSBW    ERROR                   ; Error check

        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=HELLO, -
                P2=HELLOLEN, -
                P4=#32
        BSBW    ERROR
```

```
; Declare exit handler to do cleanup operations

        $DCLEXH_S -
                DESBLK=EXITBLOCK
        BSBW    ERROR

; Create a mailbox for subprocess termination message
;
MAILBOX:
        $CREMBX_S -
                CHAN=EXCHAN, -
                MAXMSG=#120, -
                BUFQUO=#240, -
                PROMSK=#0
        BSBW    ERROR

;
; Use $GETDVI to determine the unit number of the mailbox
;
        $GETDVI_S -
                EFN=#2, -               ; Specify event flag
                CHAN=EXCHAN, -          ; Channel just assigned
                ITMLST=MBX_DVILIST      ; List of information
        BSBW    ERROR

        $WAITFR_S -                     ; Wait for synchronous completion
                EFN=#2
        BSBW    ERROR

; Translate the logical name SYS$OUTPUT, using $GETDVI
;
        $GETDVI_S -
                EFN=#2, -               ; Specify event flag
                DEVNAM=OUTPUT, -        ; Descriptor for SYS$OUTPUT
                ITMLST=TERM_DVILIST     ; List of information
        BSBW    ERROR

        $WAITFR_S -
                EFN=#2                  ; Wait for synchronous completion
        BSBW    ERROR

        CMPL    R0,#SS$_CONCEALED       ; Was the device concealed?
        BNEQ    PROCESS                 ; No, branch
        INCW    TERM_DESC               ; Yes, add one to length of name...
        DECL    TERM_ADDRESS            ; and change pointer to CONC_TERM
;
; Create the subprocess.  The logical name SYS$OUTPUT will be
; equated to the same device as SYS$OUTPUT of the creating process.
; The MBXUNT argument specifies the name of the mailbox just
; created; the mailbox will receive a message when LYRA exits.

PROCESS:
        $CREPRC_S -
                IMAGE=LYRAEXE, -
                PIDADR=LYRAPID,-
                MBXUNT=UNIT_NUMBER,-
                OUTPUT=TERM_DESC, -
                QUOTA=QLIST
        BSBW    ERROR

; If okay, format an output message showing the process ID.
```

```
        $FAO_S CTRSTR=PRCSTR, -
               OUTLEN=FAOLEN, -
               OUTBUF=FAODESC,-
               P1=LYRAPID
        BSBW   ERROR
        $QIOW_S CHAN=TTCHAN, -
               FUNC=#IO$_WRITEVBLK, -
               P1=FAOBUF, -
               P2=FAOLEN, -
               P4=#32
        BSBW   ERROR

; Queue an I/O request to the mailbox with an AST
; to receive notification when LYRA completes.

        $QIO_S  EFN=#4, -
               CHAN=EXCHAN, -
               FUNC=#IO$_READVBLK,-
               ASTADR=EXITAST, -
               IOSB=MBXIOSB,-
               P1=EXITMSG, -
               P2=#ACC$K_TERMLEN
        BSBW   ERROR

; Place names in the group logical name table using the macro GRPNAME.
; It will be LYRA's task, when awakened, to translate these
; names and display the results at the terminal.
; Note that translation of the name CYGNUS will require
; iterative translation.

PUT_NAMES:
        GRPNAME ORION,HUNTER

        GRPNAME PEGASUS,HORSE

        GRPNAME LYRA,HARP

        GRPNAME CYG,SWAN

        GRPNAME SWAN,DUCK

        GRPNAME DUCK,TALE

; After placing names in the table, wake LYRA, which has been hibernating,
; to perform the logical name translation.

        $WAKE_S PIDADR=LYRAPID
        BSBW   ERROR
        RET                             ; All finished

; AST service routine to read the termination mailbox.
; In this example, only one message is actually expected in the mailbox
; but the program performs all the following checks:

;  1. That the I/O completed successfully.
;  2. That the message in the mailbox is a process termination message.
;  3. That the process being deleted is the subprocess created.

; This service routine enables system service failure exception
; mode as an error-handling device: if a system service
; call fails, an exception condition will occur. CYGNUS
; does not declare a condition handler, so the image
; will be forced to terminate, and the system will display
; pertinent information about the exception condition.

        .ENTRY  EXITAST,^M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
        $SETSFM_S -
               ENBFLG=#1                ; Enable SSFAIL exceptions

; Check IOSB to ensure that I/O completed successfully
```

```
        CMPW    MBXIOSB,#SS$_NORMAL     ; Check that I/O was successful
        BEQL    20$                     ; Okay, go on
        $FAO_S  CTRSTR=ASTERRSTR,-      ; Otherwise, format error msg
                OUTLEN=FASTLEN, -
                OUTBUF=FASTDESC-
                P1=#IOERR, -            ; I/O error
                P2=MBXIOSB              ; Display IOSB
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
        BRW     50$                     ; Return

; Check message type field in mailbox message to ensure that the message
; is a process termination message.

20$:    CMPW    EXITMSG+ACC$W_MSGTYP,#MSG$_DELPROC      ; Check message type
        BEQL    30$                     ; Okay, go on
        $FAO_S  CTRSTR=ASTERRSTR,-      ; Otherwise, format error message
                OUTLEN=FASTLEN, -
                OUTBUF=FASTDESC,-
                P1=#IDERR, -            ; Invalid PID error
                P2=EXITMSG+ACC$W_MSGTYP ; Print message type code
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
        BRW     50$                     ; Return

; Compare the second longword in the IOSB with the PID returned
; by $CREPRC to ensure that the termination message is for LYRA.

30$:    CMPL    LYRAPID,MBPID           ; LYRA deletion?
        BNEQ    35$                     ; Yes, go on
        BRW     40$
35$:
        $FAO_S  CTRSTR=PIDERRSTR,-      ; Otherwise, format error message
                OUTLEN=FASTLEN, -
                OUTBUF=FASTDESC,-
                P1=MBPID                ; Display spurious PID
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
        BRW     50$                     ; Return

; Format an output message indicating LYRA's final exit status
; and the time of day at which LYRA terminated.

40$:    $FAO_S  CTRSTR=DONESTR, -       ; Format message telling
                OUTLEN=FASTLEN, -       ; of LYRA's demise
                OUTBUF=FASTDESC,-
                P1=EXITMSG+ACC$L_FINALSTS, -    ; Get status code
                P2=#EXITMSG+ACC$Q_TERMTIME      ; and time of deletion
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FASTBUF, -
                P2=FASTLEN, -
                P4=#32
50$:    $SETSFM_S -
                ENBFLG=#0               ; Disable exceptions
        RET                             ; Return
```

```
; This is the exit handler for CYGNUS. It receives control
; when CYGNUS exits, either normally, or as a result of
; an error condition.

        .ENTRY  EXITRTN,^M<>            ; Entry mask
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=BYE, -
                P2=BYELEN, -
                P4=#32
        BSBW    ERROR
        BLBS    STATUS,20$              ; Normal exit, continue

; If error, format error message using argument list in
; exit control block

10$:    $FAO_S  CTRSTR=BADEXSTR, -
                OUTLEN=FAOLEN, -
                OUTBUF=FAODESC,-
                P1=STATUS, -
                P2=ERRPC
        BSBW    ERROR
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32

; Common code for both normal and error exit: wait for subprocess
; to terminate (if it hasn't already), then delete all names
; from the group logical name table.

20$:    $WAITFR_S -
                EFN=#4                  ; Wait for termination message
        BSBW    ERROR
30$:    $DELLNM_S -
                TABNAM=GRPTBL           ; Delete all names
        BSBW    ERROR
        $DASSGN_S -
                CHAN=EXCHAN             ; Deassign mailbox channel
        BSBW    ERROR
        MOVL    STATUS,R0               ; Restore saved status code
        RET                             ; Exit with status

; Common error handling routine. This routine checks the
; status code in R0; if success, returns to main
; program. If there is an error, the PC is placed in the exit
; control block so that exit routine can format and display
; an error message.

ERROR:
        BLBC    R0,10$                  ; Check status code
        RSB                             ; Low bit set, go back
10$:    MOVL    (SP),ERRPC              ; Store PC
        RET                             ; RET will cause image exit
        .END    CYGNUS
```

15.3 LYRA Program Example

The program LYRA uses the following system services:

$TRNLNM	Translate Logical Name
$ASSIGN	Assign I/O Channel
$HIBER	Hibernate
$FAOL	Formatted ASCII Output with List Parameter

The program LYRA is a subprocess created by CYGNUS. After assigning a channel to its current output device, LYRA hibernates. When awakened by CYGNUS, LYRA translates the logical names placed in the group logical name table by CYGNUS, and displays the results of the translations on the terminal.

When LYRA exits, a termination message is sent to the mailbox specified by CYGNUS.

```
        .IDENT  /01/

; Macro library call

        $SSDEF                          ; Define system status values
        $LNMDEF                         ; Define logical name item codes

; Local macro

; MESSAGE,  to output messages formatted by FAO

        .MACRO  MESSAGE
        $QIOW_S CHAN=TTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32
        BSBW    ERROR
        .ENDM   MESSAGE

; Local data program section starts here

        .PSECT  RODATA,NOWRT,NOEXE

; Logical name of logical output device

OUTPUT: .ASCID  /SYS$OUTPUT/

; Group logical name table

GRPTBL: .ASCID  /LNM$GROUP/

; Announcement messages

HELLO:  .ASCII  /LYRA: INITIALIZING...AND SO TO SLEEP/
HELLOLEN:
        .LONG   HELLOLEN-HELLO

WAKEMSG:
        .ASCII  /LYRA: OKAY, WILL DO LOGICAL NAME TRANSLATION.../
WAKELEN:
        .LONG   WAKELEN-WAKEMSG

; FAO control string for logical name output message

LOGNAMSTR:
        .ASCID  "!/LYRA: !AS IS A !AS"

; Error message control string
```

```
ERRSTR:
        .ASCID  "!/LYRA: SYSTEM SERVICE ERROR AT APP. !XL R0=!XL"

; Logical names to be translated

ORIONLOG:
        .ASCID  /ORION/
CYGNUSLOG:
        .ASCID  /CYGNUS/
LYRALOG:
        .ASCID  /LYRA/
PEGASUSLOG:
        .ASCID  /PEGASUS/

; Read/write data program section starts here

        .PSECT  RWDATA,RD,WRT,NOEXE

; Item list for $TRNLNM

TRNITM: .WORD   255                     ; Buffer length
        .WORD   LNM$_STRING             ; Item code
        .LONG   0                       ; Buffer address
        .ADDRESS -                      ; Returned string length
                LOGLEN
        .LONG   0                       ; List terminator

; Output buffer for all output formatted by FAO

FAOLEN: .WORD   0                        ; Length of final string, always
        .WORD   0                        ; need longword for $OUTPUT
FAODESC:
        .LONG   80
        .ADDRESS -                       ; Address of buffer
                FAOBUF
FAOBUF: .BLKB   80

; Word to receive channel number of terminal

OUTCHAN:
        .BLKW   1

; Buffers to maintain logical name/equivalence name pairs
; in routine that performs logical name translation

LOGBUFA:
        .LONG   255
        .ADDRESS -
                BUFA
BUFA:   .BLKB   255
LOGBUFB:
        .LONG   255
        .ADDRESS -
                BUFB
BUFB:   .BLKB   255

LOGLEN: .LONG   0                        ; Save length of equivalence name

; Parameter list for call to FAOL (used by translate routine)

TLIST:
TLOGNAM:
        .LONG   0                        ; Address of logical name descriptor
TEQLNAM:
        .LONG   0                        ; Address of equivalence descriptor
SAVER3: .LONG   0                        ; Save register contents for switch
```

```
; Longword to store the PC when a system service call results in an
; error. LYRA checks the low bit of R0 following each service call.
; If set, LYRA continues; otherwise, it saves the PC and branches
; to an error-handling routine that displays the saved PC and the
; contents of R0.

ERRPC:  .LONG   0                       ; For address of SSFAIL

; Code begins here.

        .PSECT  CODE,EXE,RD,NOWRT
        .ENABL  LSB
        .ENTRY  LYRA,^M<R2,R3,R4,R5,R6> ; Entry mask

; Assign channel to device referred to by logical name
; SYS$OUTPUT. This name was placed in the logical name
; table by CYGNUS (it is also CYGNUS's logical output device).

20$:    $ASSIGN_S -
                DEVNAM=OUTPUT, -
                CHAN=OUTCHAN
        BLBS    R0,30$
        RET                             ; Exit with status if ASSIGN fails
30$:    $QIOW_S CHAN=OUTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=HELLO, -
                P2=HELLOLEN, -
                P4=#32
        BLBS    R0,40$
        MOVAL   30$,ERRPC
        BRW     ERROR

40$:    $HIBER_S
        BLBS    R0,50$
        MOVAL   40$,ERRPC
        BRW     ERROR

50$:    $QIOW_S CHAN=OUTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=WAKEMSG, -
                P2=WAKELEN, -
                P4=#32
        BLBS    R0,60$
        MOVAL   50$,ERRPC
        BRW     ERROR

60$:

; When awakened, begin translating logical names. To translate the
; names, place address of a logical name descriptor in R2 and then
; go to the subroutine that performs the translation. Repeat for
; each logical name to translate.

        MOVAL   ORIONLOG,R2
        JSB     TRANSLATE
        MOVAL   CYGNUSLOG,R2
        JSB     TRANSLATE
        MOVAL   LYRALOG,R2
        JSB     TRANSLATE
        MOVAL   PEGASUSLOG,R2
        JSB     TRANSLATE

; All finished, return

        RET

        .ENABL  LSB
```

```
; Subroutine to translate and print logical names:
; On entry to this subroutine,
; R2 = address of logical name to translate
; It uses: R3 to hold address of final result buffer
;          R4 to hold address of intermediate buffer

TRANSLATE:
        MOVAL   LOGBUFA,R3              ; Get addresses of buffers
        MOVAL   LOGBUFB,R4

; Initial translation places resultant equivalence name in buffer pointed
; to by R3

10$:    MOVL    4(R3),TRNITM+4
        $TRNLNM_S -
                TABNAM=GRPTBL, -
                LOGNAM=(R2), -
                ITMLST=TRNITM
        BLBS    R0,30$
        MOVAL   10$,ERRPC
        BRW     ERROR

; Place length of equivalence name in first word of descriptor and use this
; descriptor as input for next translation. If SS$_NOLOGNAM is returned,
; then there was no nesting of name. If not, update registers to
; provide input and output descriptors for translation and repeat
; translation until SS$_NOLOGNAM is returned.

30$:    MOVZWL  LOGLEN,(R3)             ; Fix length in buffer
        MOVL    4(R4),TRNITM+4
        $TRNLNM_S -
                TABNAM=GRPTBL, -
                LOGNAM=(R3), -
                ITMLST=TRNITM
        BLBS    R0,40$
        CMPL    R0,#SS$_NOLOGNAM
        BEQL    50$
        MOVAL   30$,ERRPC
        BRW     ERROR

40$:    MOVL    R3,SAVER3               ; Switch
        MOVL    R4,R3
        MOVL    SAVER3,R4
        BRB     30$                     ; Try again

; Place addresses of logical name and equivalence names in FAO parameter list
; and call FAO to format output message, then output the message.

50$:    MOVL    R2,TLOGNAM
        MOVL    R3,TEQLNAM
        $FAOL_S CTRSTR=LOGNAMSTR, -
                OUTLEN=FAOLEN, -
                OUTBUF=FAODESC,-
                PRMLST=TLIST
        BLBS    R0,60$
        MOVAL   50$,ERRPC
        BRW     ERROR

60$:    $QIOW_S CHAN=OUTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32
        BLBS    R0,70$
        MOVAL   60$,ERRPC
        BRW     ERROR
70$:    RSB                             ; To main routine
```

```
; Error-handling routine:
; This routine uses the saved PC and R0 to format a message describing
; the conditions under which a call to a system service failed.

ERROR:
        $FAO_S  CTRSTR=ERRSTR, -
                OUTBUF=FAODESC, -
                OUTLEN=FAOLEN,-
                P1=ERRPC, -
                P2=R0
        $QIOW_S CHAN=OUTCHAN, -
                FUNC=#IO$_WRITEVBLK, -
                P1=FAOBUF, -
                P2=FAOLEN, -
                P4=#32
        RET
        .END    LYRA
```

A

User-Written System Services

A **user-written system service** is a shareable image containing one or more routines that nonprivileged users can call to perform privileged functions. The creator of the user-written system service codes, compiles or assembles, links, and installs the routine; other users can then call this routine in their programs using the standard CALL interface, provided they have linked their object module or modules with the user-written system service. User-written system services thus provide a vehicle for you to write and use your own system services.

Because a user-written system service is installed as a protected image, it is allowed to execute in kernel or executive mode. For this reason, 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 the behalf of the user, you must ensure that your image only performs tasks the user could 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. Take care that operating in an inner mode does not give you any special privileges with respect to the operation being performed. Only resume a privileged state 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 *Guide to VMS System Security* for descriptions.)
- As a protected image, your program does not have the entire VMS 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 VMS RMS from executive mode but not from kernel mode. On VMS Version 5.4 or later, any VMS RMS files that were opened with privilege from an inner mode can be left open during user execution, but this is not acceptable on earlier versions of VMS.
- Never make subroutine calls to other shareable images from kernel or executive mode.

If the system service is linked with the /PROTECT command qualifier to the LINK command, the VMS Linker Utility prevents outbound calls. This is not the case when you select the PROTECT option in a linker options file (see Section A.2.1). With only selected image sections protected, the operating system does not prevent outbound calls and the responsibility for evaluating the trustworthiness of calls belongs to the program and its designer. Your program is responsible for determining which outbound calls are suspect and which ones are not.

- 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.

A.1 Coding a User-Written System Service

Any user-written system service has to satisfy the following coding requirements:

- The system service must contain a special change-mode vector identifying a kernel-mode or executive-mode dispatcher, or both.
- Its entry point must be followed by a CHMK or CHME instruction with a negative operand.
- Any kernel-mode or executive-mode dispatcher pointed to in the change-mode vector must validate the CHMK or CHME operand, and must be followed by one or more routines that perform the desired function or functions.
- The user-written system service (or each routine in it) must enable any necessary user privileges and disable them when they are no longer needed. (You use the Set Privileges ($SETPRV) system service to enable and disable user privileges.)

The following sections discuss each of these considerations.

A.1.1 Change-Mode Vector

One of the program sections in a user-written system service must start with a change-mode vector. The purpose of this vector is to point (by means of self-relative offsets) to the start of the kernel-mode or executive-mode dispatch routine within the user-written system service.

The program section containing the change-mode vector must be assigned the VEC attribute. (See the *VAX MACRO and Instruction Set Reference Manual* or the *VMS Linker Utility Manual* for a discussion of program section attributes.)

The change-mode vector must have the following format. The offsets from the base of the vector to specific items are expressed by symbols starting with PLV$L_. The $PLVDEF macro defines these symbols, which are contained in SYS$LIBRARY:STARLET.MLB.

The symbols defined by the $PLVDEF macro are:

Vector Type Code PLV$L_TYPE
(PLV$C_TYP_CMOD)

Kernel Mode Dispatcher Offset PLV$L_KERNEL

Exec Mode Entry Offset PLV$L_EXEC

User Rundown Service Offset PLV$L_USRUNDOWN
Reserved

RMS Dispatcher Offset PLV$L_RMS

Address Check PLV$L_CHECK

The significant offsets in the change-mode vector and their contents are as follows:

- PLV$L_TYPE—Contains the type code PLV$C_TYP_CMOD, identifying this as a change-mode vector.
- PLV$L_KERNEL—Contains a self-relative pointer to the user-supplied kernel-mode dispatcher. (Self-relative means relative to the start of the longword field.) A value of 0 indicates that no kernel-mode dispatcher exists.
- PLV$L_EXEC—Contains a self-relative pointer to the user-supplied executive-mode dispatcher. A value of 0 indicates that no executive-mode dispatcher exists.
- PLV$L_USRUNDWN—Contains a self-relative pointer to the user-supplied rundown routine. This offset is optional. This routine is intended to be used for image-specific cleanup and resource deallocation. When the image linked against the user-written system service is run down by the system, this run-time routine is invoked. Unlike exit handlers, it is always called when a process or image exits. (You call this routine with a JSB instruction; it returns with an RSB instruction in kernel mode, at IPL 0.) For information about exit handlers, see Section 8.6.3.
- PLV$L_RMS—Contains a self-relative pointer to the dispatcher for VMS RMS services. A value of 0 indicates that no user-supplied VMS RMS dispatcher exists. Only one user-written system service should specify the VMS RMS vector, because only the last value will be used. This field is intended for use only by Digital.
- 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.

A.1.2 Entry Point to the User-Written System Service

The entry point to a user-written system service must be an entry mask followed by a CHMK (Change Mode to Kernel) or CHME (Change Mode to Executive) instruction, depending on whether you want control transferred to a kernel-mode or executive-mode dispatcher (specified in the vector). The operand of the CHMK or CHME instruction must be a negative value, because positive values are reserved for calling system services supplied by Digital.

A.1.3 Kernel-Mode or Executive-Mode Dispatcher

The kernel-mode or executive-mode code you write must do the following:

- Validate the CHMK or CHME operand, handling any invalid operands.
- Transfer control to the appropriate coding segment 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.
- Precede the coding segments performing the functions the user-written system service was designed to perform.

A.1.4 Enabling and Disabling User Privileges

A user-written system service must enable any privileges that it needs but 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.

```
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
```

Any code executing in executive or kernel mode is granted an implicit SETPRV privilege, so it can enable any privileges it desires.

A.2 Linking the User-Written System Service

The following conventions apply when you link (create) a user-written system service:

- Use the /SHAREABLE command qualifier to identify the image to be created as shareable.
- Use the /PROTECT command qualifier or the PROTECT= option to identify the entire image or specific clusters, respectively, as protected against user-mode or supervisor-mode write access.
- Define the user-written system service's entry point as a universal symbol, using the UNIVERSAL= option.

A.2.1 Specifying Protection for the Image or Clusters

The VMS Linker allows you to protect all or part of a user-written system service from write access by code executing in user or supervisor mode. The /PROTECT command qualifier causes all image sections to be so protected. The PROTECT= option in a linker options file permits you to specify protection when some clusters require protection and some do not.

When creating a privileged shareable image, you should protect the clusters containing code or data that privileged-mode code must access. You should not protect the clusters that user-mode code must access. Thus, the /PROTECT command qualifier should only be used when the entire shareable image needs to be protected. The PROTECT= option allows user-written system services to contain parts into which the nonprivileged user can write.

Be very careful to segregate user-accessible storage from the privileged parts of your service. If user-accessible storage is used by privileged code, it must be validated as carefully as any other user input.

The linker option takes the form PROTECT=YES or PROTECT=NO and precedes the specifications for clusters that are to be protected or unprotected, respectively. The following example shows the linker options file entries to designate clusters A, B, and D as protected, and cluster C as unprotected.

```
PROTECT=YES
CLUSTER=A,,,MODULE1,MODULE2
CLUSTER=B,,,MODULE3,MODULE4,MODULE5
PROTECT=NO
CLUSTER=C,,,MODULE6,MODULE7
PROTECT=YES
CLUSTER=D,,,MODULE8,MODULE9
```

The *VMS Linker Utility Manual* discusses linker options files and explains each available option.

A.3 Installing the User-Written System Service

To make a user-written system service usable by nonprivileged programs, you must install it as a protected permanent global section. The following procedure is recommended:

1. Move the user-written system service to a protected directory, such as SYS$SHARE.
2. Run the Install Utility, specifying the /PROTECT, /OPEN, and /SHARED qualifiers. You can also specify the /HEADER_RESIDENT qualifier. (See the *VMS Install Utility Manual* for a description.) The following entry could be used to install a user-written system service whose image name is USS.

```
$ INSTALL
ADD SYS$SHARE:USS/PROTECT/OPEN/SHARED/HEADER_RES
```

Installing the system service as /WRITABLE creates a situation where all users calling the services can write to a common storage area (provided you define the program section's attributes as global and shareable). While a common data area might be desirable in special situations, it is more typical for each process to have its own writable storage area.

A.4 Using the User-Written System Service

To the nonprivileged user of a user-written system service, there is no difference between using it and using an ordinary shareable image. To use a user-written system service, you must do the following:

1. Call the user-written system service.
2. Link the user-written system service into the executable image being created.

A.5 Program Listings

Refer to SYS$EXAMPLES:USSDISP.MAR for listings of modules in a user-written system service and of a module that calls the user-written system service.

B

Using Shared Memory

The MA780 is a multiport memory unit that can be attached to VAX–11/780 processors. Each VAX–11/780 processor can support up to two MA780s. Each MA780 has four ports, thereby allowing up to four VAX–11/780 processors to be attached to it.

Using one or more multiport memory units, an application can consist of multiple processes running on different VAX–11/780 processors. Regardless of the processor on which they are running, these processes can communicate the completion of an event, send messages, and share common data and code by means of the shared memory.

B.1 Preparing Multiport Memory for Use

Before an application using multiport memory can execute under the VMS operating system, the system manager must activate VMS in the processors connected to the multiport memory unit and initialize that memory. See your operations guide for an explanation of the system management responsibilities associated with a multiport memory unit; this section summarizes the system management functions for the benefit of the application programmer.

First, the system manager activates the VMS operating system in a VAX–11/780 processor and initializes the multiport memory unit. These actions cause the following to occur:

- The uninitialized shared memory is connected to the VMS operating system running in the processor.
- A name is defined that all processes running in all processors can use to refer to the shared memory (see Section B.3).
- Limits are set for the following resources in this multiport memory unit:
  - Common event flag clusters: the total number that can be created, and the number that can be created by processes running on this processor
  - Mailboxes: the total number that can be created, and the number that can be created by processes running on this processor
  - Global sections: the total number that can be created, and the number that can be created by processes running on this processor

The system manager activates the VMS operating system in the other processors connected to the multiport memory unit. The system manager then connects the initialized shared memory to the VMS operating system running in each of these processors and sets limits for the number of common event flag clusters, global sections, and mailboxes that processes on each processor can create in the multiport memory.

The system manager can also install global sections in shared memory just as they are installed in local memory. The Install Utility can be used to create shared memory global sections for known files. After the global sections are installed, a process running in any processor connected to the multiport memory can map to the section, if the process has the appropriate privilege. The process can gain access to the global section either by using a logical name defined by the system manager or by using the section name specified when the global section was created. In the latter case, the section name must be unique on the processor running the process attempting to access the global section.

B.2 Privileges Required for Shared Memory Use

To use facilities in memory shared by multiple processors, you must have all of the user privileges required to use the equivalent facility in local memory. For example, to create a permanent global section you must have the PRMGBL privilege, and to create a temporary or permanent mailbox you must have the TMPMBX or PRMMBX privilege, respectively.

In addition to any other required privileges, you must have the SHMEM privilege to create or delete a common event flag cluster, mailbox, or global section in memory shared by multiple processors. However, you do not need the SHMEM privilege to use an existing cluster, mailbox, or global section in multiport memory.

B.3 Naming Facilities in Shared Memory

To allow access to facilities in memory shared by multiple processors, the system manager and application programmers define names that application programs use to refer to individual shared memory units. During system installation, the system manager defines the name that processes on that particular processor use to refer to the shared memory itself. Application programs define the names that they use to refer to common event flag clusters, global sections, and mailboxes located in the shared memory.

By convention, facilities in shared memory have a name string in the following format:

memory-name:facility-name

where:

memory-name	Name assigned by the system manager during system installation to the shared memory containing the facility. The VMS operating system requires the memory name when you specify a common event flag cluster or mailbox. The colon is recognized as a delimiter separating the two parts of the name string. The name must contain 43 or fewer characters, and can consist only of alphabetic characters, numeric characters, the dollar sign ($), and the underscore (_).
facility-name	Logical name assigned to the event flag cluster, global section, or mailbox. The name must contain 43 or fewer characters, and can consist only of alphabetic characters, numeric characters, the dollar sign ($), and the underscore (_).

Following are examples of facility names.

SHRMEM:GS_DATA	Identifies the global section GS_DATA in the shared memory named SHRMEM
SHRMEM:MAILBX	Identifies the mailbox MAILBX in the same shared memory

B.4 Assigning Logical Names and Logical Name Translation

You can define a logical name for a shared memory facility with the DEFINE or ASSIGN command or with the Create Logical Name ($CRELNM) system service. Application programs can then refer to the facility using the logical name; for example, a process can invoke the Create Mailbox and Assign Channel ($CREMBX) system service specifying the logical name for an existing mailbox to which a channel is to be assigned.

When translating a logical name for a shared memory facility, the VMS operating system uses a slightly different approach from that used for other logical names. The purpose of this approach is to allow programmers to specify either the complete name (memory name and facility name) or a logical name that the system will translate to the complete name. If you define logical names properly, a program that uses a given facility in local memory can be run without change to use the facility in shared memory.

Whenever the VMS operating system encounters the name of a common event flag cluster, mailbox, or global section, it performs the following special logical name translation sequence:

1. Inserts one of the following prefixes to the name (or to the part of the name before the colon if a colon is present):

   CEF$ for common event flag clusters
   MBX$ for mailboxes
   GBL$ for global sections

2. Subjects the resultant string to logical name translation. If translation does not succeed (that is, the original name did not use a logical name), passes the original name string to the system service. If translation does succeed, goes to step 3.

3. Appends the part of the original string after the colon (if any) to the translated name.

4. Repeats steps 1 to 3 (up to a number of times determined by the system, if necessary) until logical name translation fails. When translation fails, passes the string to the system service.

For example, assume that you have made the following logical name assignment:

```
$ DEFINE MBX$CHKPNT SHRMEM$1:CHKPNT
```

Assume also that your program refers to the mailbox name as CHKPNT in a system service argument. The following logical name translation takes place:

1. MBX$ is prefixed to CHKPNT.

2. MBX$CHKPNT is translated to SHRMEM$1:CHKPNT.

3. No further translation is successful; therefore, the string SHRMEM$1:CHKPNT is passed to the system service.

The logical name definition in the preceding example allows a program that used a mailbox named CHKPNT in local memory to run using the mailbox in shared memory, without being recompiled or relinked.

Note that if a process creates one or more subprocesses and they use a mailbox or common event flag cluster in shared memory, the creator should place the logical name in the job or group logical name table (for example, specify the /JOB or /GROUP qualifier with the DEFINE command). If the name is defined in the process logical name table (the default), the subprocesses do not receive the correct equivalence name, because each subprocess has its own process logical name table.

There are two exceptions to the logical name translation method discussed in this section:

- If the facility name starts with an underscore (_), the VMS operating system strips the underscore and considers the resultant string the actual name (that is, no further translation is performed).
- If the facility is a global section with a name in the format *name_nnn*, VMS 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 with known images and shared files installed by the system manager.

B.5 How VMS Finds Facilities in Shared Memory

After the VMS operating system performs the logical name translation described in Section B.4, the final equivalence name must be the name of a facility in the processor's local memory or in shared memory. If the equivalence name specifies the name of a shared memory (that is, the name is in the format *memory-name:facility-name*), VMS searches for the facility in the appropriate database of the specified shared memory unit.

If the equivalence name specifies a common event flag cluster or mailbox and does not specify a memory name, VMS searches through the local memory common event flag cluster database or mailbox database until it locates the specified cluster or mailbox. Absence of a memory name as part of a common event flag cluster name or mailbox name indicates that the facility is located in local memory.

If the equivalence name specifies a global section and does not specify a memory name, the VMS operating system looks for the section in the following order:

1. It searches the global section tables for sections in the processor's local memory.
2. It searches the global section tables for each initialized shared memory connected to the processor in the order in which they were connected and recognized by the processor.

The result of searching in this order is that global sections in the processor's local memory take precedence over those in shared memories. Thus, absence of a memory name as part of a global section name is not used as an indication of where the global section is located.

B.6 Using Common Event Flags in Shared Memory

Under the VMS operating system, any process can associate with up to two common event flag clusters (event flag numbers 64 through 95 and 96 through 127). These clusters can be located in shared memory or in local memory. To create and associate with a common event flag cluster in shared memory and manipulate flags in the cluster, you use the same steps as you would to associate with a common event flag cluster in local memory:

1. Issue the Associate Common Event Flag Cluster ($ASCEFC) system service to create and name the cluster or to associate with an existing named cluster.
2. Issue any of the services that set, clear, and wait for designated event flags, as appropriate.

Creation of a shared memory common event flag cluster requires CEF PORT quota. You can set up this quota by using the SYSGEN command SHARE. This quota is restored when the common event flag cluster is deleted.

As with local memory clusters, the first process among cooperating processes to issue the Associate Common Event Flag Cluster ($ASCEFC) system service causes the cluster to be created and named. Any other process calling this service and specifying the same cluster name associates with that existing cluster. The VMS operating system implicitly qualifies cluster names with the group number of the creator's UIC; therefore, other cooperating processes must belong to the same group. All of the event flag system services, with the exception of Associate Common Event Flag Cluster and Disassociate Common Event Flag Cluster, function identically whether they are used with local or shared memory clusters. The only difference with the associate and disassociate system services is that, to specify a cluster in shared memory, you must provide the memory name as well as the cluster name. That is, after VMS performs logical name translation of the name argument, the cluster name must have the following format:

```
memory-name:cluster-name
```

Section B.3 describes the name format and Section B.4 explains the logical name translation performed by the system. Chapter 4 describes all of the event flag services in detail.

B.7 Using Mailboxes in Shared Memory

The creation of a mailbox in shared memory requires MAILBOX PORT quota. This quota is acquired by means of the SYSGEN command SHARE and is returned when the mailbox is deleted.

The first process on each processor must use the Create Mailbox and Assign Channel ($CREMBX) system service to create a shared memory mailbox and assign a channel to it. Any $CREMBX system service call referring to a shared memory mailbox must specify a mailbox name that has or translates to the following format (Section B.4 explains the logical name translation procedure):

```
memory-name:mailbox-name
```

When the mailbox is created, the $CREMBX system service also creates the mailbox-name portion of the name string as a logical name with an equivalence name in the format MB*n*. For example, if the complete name string is SHMEM:MAILBOX, the system service creates MAILBOX as a logical name with an equivalence name of, for example, MBB005.

The Assign I/O Channel ($ASSIGN) and Deassign I/O Channel ($DASSGN) system services require that you specify only the mailbox-name portion of a shared memory mailbox name string. Likewise, any high-level language program statements that open, close, read from, or write to a shared memory mailbox must specify only the mailbox-name portion.

The following code example shows two VAX FORTRAN programs using a shared-memory mailbox. The memory name in this example is SHMEM. The programs are running in processes on separate processors.

```
        PROGRAM   ONE
        INTEGER*4 SYS$CREMBX,STATUS,CHAN

        STATUS = SYS$CREMBX(,CHAN,,,,,'SHMEM:MAILBOX')
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

C-- Open the mailbox using the mailbox-name; write a message.

        OPEN (UNIT=1,NAME='MAILBOX',STATUS='NEW')
        WRITE (1,*)  MESSAGE
           .
           .
           .
        END

        PROGRAM   TWO
        INTEGER*4 SYS$CREMBX,STATUS,CHAN

        STATUS = SYS$CREMBX(,CHAN,,,,,'SHMEM:MAILBOX')
        IF (.NOT. STATUS) CALL LIB$STOP(%VAL(STATUS))

C-- Open the mailbox using the mailbox-name; read the message.

        OPEN (UNIT=1,NAME='MAILBOX',STATUS='OLD')
        READ (1,*)  MESSAGE
           .
           .
           .
        END
```

You cannot use a mailbox in shared memory as a process termination mailbox. Note that because the processes run on different processors, each must issue a $CREMBX system service request.

A mailbox located in memory shared by multiple processors is deleted when all of the following occur:

- A processor is rebooted.
- The multiport memory is not reinitialized.
- No other processor has any processes with channels assigned to the mailbox.

Section 7.20 discusses mailboxes and related system services in detail.

B.8 Using Global Sections in Shared Memory

You need GLOBAL SECTION PORT quota to create a global section in memory shared by more than one processor. You acquire this quota using the SYSGEN command SHARE; it is returned when the global section is deleted or when you reissue the SYSGEN command SHARE after the processor on that port is rebooted.

Under the VMS operating system, processes can map global sections located in local memory or in shared memory. A global section in shared memory can be mapped to an image file or a data file, just like a global section in local memory. To create a global section in shared memory, you perform the same steps as you would to create a global section in local memory:

1. Using VMS RMS, open the file to be mapped.
2. Issue the Create and Map Section ($CRMPSC) system service.

The file to be mapped must reside on a disk device attached to the local processor. After the section is created, however, processes on all processors attached to the shared memory can map the section. To map to an existing global section in shared memory, you issue a Map Global Section ($MGBLSC) system service specifying the name of the section. After the section is mapped, processes gain access to shared memory global sections in the same manner as they do to local memory sections. VMS thus makes use of the shared memory unit transparent to the process.

The VMS operating system treats the pages of a global section in shared memory differently from pages in local memory. When a process creates a shared-memory global section, VMS brings all of the pages of the mapped image or data file into memory. Any process mapped to that global section can gain access to those pages without incurring a page fault because the pages are already in physical memory. Unlike process pages in local memory, global section pages in shared memory are not included in the working sets of the processes that map the section.

Because no paging occurs, VMS never writes the contents of shared memory global section pages back to their disk file. For read/write global sections in which you want to maintain an updated file while the application executes, you must issue an Update Section File on Disk ($UPDSEC) system service. The process issuing the update request must execute on the same processor as the process that created the global section. You can update the disk file periodically during execution of the application as a checkpoint precaution. The disk file is automatically updated when the section is deleted.

Each process that has mapped a global section in shared memory can unmap the section in either of the following ways:

- Issue a Delete Virtual Address Space ($DELTVA) system service to delete the process's virtual address space that maps the section.
- Terminate the current image, thereby causing VMS to unmap the process from the section automatically.

Deleting a global section in shared memory requires an explicit deletion request, because all global sections in shared memory must be permanent sections. The deletion request can be either a Delete Global Section ($DGBLSC) system service issued by the application or a deletion request issued by the system manager using the Install Utility. In either case, the VMS operating system does not perform the actual deletion until all processes that have mapped the section unmap it.

When a process requests deletion of a shared memory global section page, VMS waits until no direct I/O is outstanding for the process before deleting the page. This is because no reference count is maintained for shared memory global section pages. (For example, VMS cannot determine whether outstanding direct I/O is for the shared memory global section page or not.) Applications using

devices that have direct I/O perpetually outstanding, such as the DR32, must not delete shared memory global section pages because this causes the process to hang in the MWAIT state (unless the applications cancel the outstanding I/O request first).

B.8.1 Removing Shared Memory Global Sections

A shared memory global section can be deleted only by the creating processor.

If you rebootstrap a processor and reconnect it to an MA780 without reinitializing the MA780, the System Generation Utility (SYSGEN) does cleanup for the processor. This cleanup causes all global sections created by processes running on this processor to be marked as having no creating processor. (The data structures that allow the data in the global section pages to be written back into the disk file no longer exist.)

Without a creating processor, you must do the following before you attempt to delete shared memory global sections:

1. Reboot all processors.
2. Reinitialize the MA780.

Section 12.7 provides information on the use of the VMS system services used with global sections, that is, memory management system services. Section B.8.2 provides information specifically related to creating and mapping a global section in shared memory. The $CRMPSC, $MGBLSC, $DGBLSC, and $UPDSEC system services are the only memory management system services for which the shared memory has any direct implications.

B.8.2 Create and Map Section System Service

The Create and Map Section system service has the following general formats when issued to create or map (or both) a global section in shared memory:

VAX MACRO Format

```
$CRMPSC [inadr], [retadr], [acmode], [flags], gsdnam
        ,[ident], [relpag], [chan], [pagcnt], [vbn], [prot]
```

High-Level Language Format

```
SYS$CRMPSC [inadr], [retadr], [acmode], [flags], gsdnam
           ,[ident], [relpag], [chan], [pagcnt], [vbn], [prot]
```

With the exception of the **flags**, **gsdnam**, and **pfc** arguments, the arguments of this service are not affected by MA780 considerations.

flags
Mask defining the section type and characteristics. Of the flags defined, you must set the following two.

Flag	Meaning
SEC$M_GBL	Global section
SEC$M_PERM	Permanent section

That is, sections in shared memory must be permanent global sections.

If appropriate, you can also set the following flags.

Flag	Meaning	Default
SEC$M_DZRO	Pages are demand-zero pages	Pages are not zeroed when copied
SEC$M_WRT	Read/write section	Read-only
SEC$M_SYSGBL	System global section	Group global section
SEC$M_EXPREG	Map section into the first free range of virtual addresses large enough to hold the section	Map section according to the **inadr** argument

When using the Create and Map Section system service to create global sections in shared memory, you cannot set either SEC$M_CRF (copy-on-reference) or SEC$M_PFNMAP (page frame number mapping). If you set SEC$M_CRF, VMS places the global section in local memory.

gsdnam

Address of a character string descriptor pointing to the text name string for the global section. This argument is required for creating sections in shared memory.

The string can be either the name of a global section or the logical name of a global section. The VMS operating system performs logical name translation as described in Section B.4.

VMS implicitly qualifies global section names with an identification. For group global sections, the section name is also implicitly qualified by the group number of the process creating the global section.

pfc

Page fault cluster size for local memory sections. This argument is ignored for global sections in shared memory, because VMS reads the file into memory when it creates the section and does not allow paging for sections in shared memory.

C

Implementing Site-Specific Security Policies

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 VMS 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.

C.1 Creating Loadable Security Services

This section describes how to create a system service image and how to update the file SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA, 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 three system services with services specific to your site:

- $ERAPAT—Generates a security erase pattern
- $MTACCESS—Controls magnetic tape access
- $HASH_PASSWORD—Applies a hash algorithm to an ASCII password

When creating the system service, you code the source module and define the vector offsets, the entry point, and the program sections for the system service. At this point, 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 VMS system 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 in your service.

The following sections describe how to create and load the $ERAPAT system service. An example of the $ERAPAT system service can be found in SYS$EXAMPLES:DOD_ERAPAT.MAR on the VMS operating system. What is described here also applies to the system services $HASH_PASSWORD and $MTACCESS. An example of how to prepare and load the $HASH_PASSWORD service can be found in SYS$EXAMPLES:HASH_PASSWORD.MAR.

C.1.1 Preparing and Loading a System Service

Use the following procedure to prepare and load a system service, in this case $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 $ERAPAT system service.)

   c. Use the following macros to declare the desired program sections (PSECT).

   ```
   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 $HASH_PASSWORD example.) To link the $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$ERAPAT.EXE image produced by the link command into the directory SYS$COMMON:[SYS$LDR]. 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 *VMS SYSMAN Utility Manual* for a description.) 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	Not enough nonpaged pool	Check SYSGEN parameters
MULTIPLE_ISDS	More than one image section of a given type	Check link options
BAD_GSD	An inconsistency was detected	Verify that the image was properly linked
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. 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 $ERAPAT system service use the normal VMS 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 SYSGEN parameter LOAD_SYS_ IMAGES to 0.

C.1.2 Removing an Executive Loaded Image

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. 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 $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 SYSGEN parameter LOAD_SYS_IMAGES to 0.

C.2 Installing Site-Specific Password Policy Filters

A site security administrator can screen new passwords to make sure they comply with a site-specific password policy. (See the *Guide to VMS System Security* for more information.) This section describes how a security administrator would 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. In addition, if INSTALL and SYSPRV file access auditing are enabled, multiple security alarms are generated when the shareable image is installed and the change to the SYSGEN parameter is noted on the operator console. The shareable image contains two global routines, which are called by the VMS Set Password Utility whenever a user changes a password with the SET PASSWORD command.

Caution

The two global routines allow a site security administrator to obtain both the proposed plaintext password and its equivalent quadword hash value. Therefore, unauthorized use of the global routines by a malicious privileged user compromises your system's security.

Digital recommends that you place an alarm access control list entry (ACE) on the following shareable image and on its parent directory.

```
$ SET ACL/ACL=(ALARM=SECURITY,ACCESS=WRITE+CONTROL+DELETE+SUCCESS+FAILURE) -
_$ SYS$LIBRARY:VMS$PASSWORD_POLICY.EXE
$ SET ACL/ACL=(ALARM=SECURITY,ACCESS=WRITE+CONTROL+SUCCESS+FAILURE) -
_$ SYS$COMMON:[000000]SYSLIB.DIR
```

You must also enable access control list (ACL) alarms using the following command:

```
$ SET AUDIT/ALARM/ENABLE=ACL
```

Once in place, these alarms catch all attempts to replace or to modify the VMS$PASSWORD_POLICY image.

C.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 directions 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 using the command procedure VMS$PASSWORD_POLICY_LNK.COM, located in SYS$EXAMPLES.

C.2.2 Installing a Shareable Image

To install a shareable image, perform the following steps:

1. Copy the resulting 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 SYSGEN parameter LOAD_PWD_POLICY to 1.

```
$ 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 file SYS$SYSTEM:SYSTARTUP_V5.COM and modify the system parameter file, MODPARAMS.DAT, so the parameter LOAD_PWD_POLICY is set to 1.

4. Run AUTOGEN as follows to ensure that the SYSGEN parameters are set correctly on subsequent system startups:

```
$ @SYS$UPDATE:AUTOGEN SAVPARAMS SETPARAMS
```

Index

A

B

C

D

E

F

G

H

I

J

L

M

Q

R

S

T

U

V

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

Introduction to VMS System Services

AA–LA68B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Information Products
ZK01–3/J35
110 SPIT BROOK RD
NASHUA, NH 03062–9987

-- Do Not Tear - Fold Here ----------

Reader's Comments

Introduction to VMS System Services

AA–LA68B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

--- **Do Not Tear - Fold Here and Tape** ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Information Products
ZK01–3/J35
110 SPIT BROOK RD
NASHUA, NH 03062–9987

--- **Do Not Tear - Fold Here** ----------