VMS

digital

VMS Version 5.4 New Features Manual

Order Number: AA–LA97C–TE

VMS Version 5.4 New Features Manual

Order Number: AA–LA97C–TE

June 1990

This manual describes the new features of the VMS Version 5.4 operating system. It also describes features that were new for Versions 5.1, 5.2, and 5.3 of the VMS operating system but are not yet documented in other printed manuals.

Revision/Update Information:	This manual supersedes previous versions of the *VMS New Features Manual*.
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

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

ZK5481

Production Note

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

Contents

Preface

Intended Audience

This book is intended for general users, system managers, and programmers who use the VMS operating system.

Structure of This Document

This manual is organized as follows:

- Part 1, Overview of Major New Features, contains a complete summary of the new VMS Version 5.4 software features. Part 1 also includes separate chapters describing the following major, systemwide enhancements to the VMS operating system:
  - Vector processing support
  - DECdtm services

Note: **It is important that you read Part 1 first for a complete overview of the VMS Version 5.4 new features and for a complete description of VMS support for vector processing and DECdtm services. In addition to providing essential information, Part 1 (particularly the vector processing and DECdtm chapters) also directs you to relevant material located elsewhere within this manual and others.**

- Part 2, General User Features, describes new features primarily of interest to general users of the VMS operating system. The chapters within provide information about the following:
  - DCL commands and lexical functions
  - EVE editor
  - System messages
  - DECwindows user and desktop applications
- Part 3, System Management Features, describes new features primarily of interest to system managers. The chapters within provide information about the following components of the VMS operating system:
  - AUTOGEN
  - UETP
  - SYSMAN Utility
  - VAXcluster management
  - System Generation Utility (SYSGEN)
  - Error Log Utility (ERROR LOG)
  - System security

  - Log Manager Control Program Utility (LMCP)
  - Monitor Utility (MONITOR)
  - Network Control Program Utility (NCP)
  - VMS Volume Shadowing Phase II

- Part 4, Programming Features, describes new features primarily of interest to programmers. The chapters within provide information about the following components of the VMS operating system:
  - VMS Debugger
  - Linker Utility (LINK)
  - Mail Utility routines
  - System Services
  - Run-Time Library Routines
  - Record Management Services (RMS)
  - I/O Driver support
  - System Dump Analyzer (SDA)
  - Device Support
  - VAX Text Processing Utility (VAXTPU)
  - VAX RMS Journaling
  - VMSINSTAL
  - Compound Document Architecture (CDA) support
  - VMS DECwindows Display PostScript system
  - XUI Toolkit

The *VMS Version 5.4 New Features Manual* also has three appendixes documenting features that were new to Versions 5.3, 5.2, and 5.1 of the VMS operating system but are not yet documented in other printed manuals.

Associated Documents

Refer to the following documents for more detailed information about the VMS Version 5.4 software features described in this manual. For more information about these documents, see the *Overview of VMS Documentation* or contact your Digital representative.

- *VMS Version 5.4 Release Notes*
- *VMS DCL Dictionary*
- *VMS EVE Reference Manual*
- *VMS System Messages and Recovery Procedures Reference Manual: Part I*

- *VMS System Messages and Recovery Procedures Reference Manual: Part II*
- *VMS SYSMAN Utility Manual*
- *VMS Volume Shadowing Manual*
- *VMS VAXcluster Manual*
- *VMS Debugger Manual*
- *Introduction to VMS System Routines*
- *VMS Utility Routines Manual*
- *VMS RTL Mathematics (MTH$) Manual*
- *VMS RTL Parallel Processing (PPL$) Manual*
- *VAX MACRO and Instruction Set Reference Manual*
- *VAX RMS Journaling Manual*
- *VMS Developer's Guide to VMSINSTAL*
- *VMS I/O User's Reference Manual: Part I*
- *VAX Text Processing Utility Manual*
- *VMS Device Support Manual*
- *CDA Reference Manual*
- *Introduction to the CDA Services*
- *Guide to Creating Compound Documents with the CDA Toolkit*
- *VMS DECwindows Toolkit Routines Reference Manual*
- *VMS DECwindows Display PostScript System Programming Supplement*

The following manuals are published by Adobe Systems Incorporated but are available through Digital. See Section 31.3 of this manual for more information.

- *Display PostScript System Perspective for Software Developers*
- *Display PostScript System pswrap Reference Manual*
- *PostScript Language Extensions for the Display PostScript System*
- *PostScript Language Color Extensions*
- *Display PostScript System Client Library Reference Manual*
- *PostScript Document Structuring Conventions Specification*

Conventions

The following conventions are used in this manual:

mouse	The term *mouse* is used to refer to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. (The buttons can be redefined by the user.)
PB1, PB2, PB3, PB4	PB1, PB2, PB3, and PB4 indicate buttons on the puck.
SB1, SB2	SB1 and SB2 indicate buttons on the stylus.
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.
$\boxed{\text{Return}}$	In examples, a key name is shown enclosed in a box to indicate that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	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.

Part 1: Overview of Major New Features

This part contains the following chapters:

- Chapter 1, Summary of New VMS Version 5.4 Software Features
- Chapter 2, Introduction to Vector Processing
- Chapter 3, Introduction to DECdtm Services

In addition to providing essential information about VMS Version 5.4 software features, the chapters in Part 1 also direct you to relevant material located elsewhere within this manual and others.

1 Summary of New VMS Version 5.4 Software Features

This chapter provides a summary (in Table 1–1) of the new VMS Version 5.4 software features described throughout this manual and in the other new and revised manuals associated with this release (see the Preface for a complete list).

For information about new and enhanced hardware, see the *VMS Version 5.4 Release Notes*.

Table 1–1 Summary of VMS Version 5.4 Software Features

	VMS Version 5.4 Systemwide Features
Vector processing	Systemwide support for vector processing on VAX 9000 series and VAX 6000-400 series computers includes the VAX Vector Instruction Emulation Facility (VVIEF), specific DCL commands and lexical functions, and the Accounting, Error Log, Monitor, SDA, Debugger, Patch, and RTL MTH$ facilities. See Chapter 2 for a complete description of vector processing support.
DECdtm services	Systemwide support for DECdtm services includes the new Log Manager Control Program Utility (LMCP), new MONITOR TRANSACTION command, new and modified system services, new TRANSACTION_ID data type, and enhanced VAX RMS Journaling support. See Chapter 3 for a complete description of DECdtm services.
	VMS Version 5.4 General User Features
DCL commands	New and enhanced DCL commands let you control date compaction on the TA90E tape drive, convert procedures written in PostScript to callable routines, compile fonts for the DECwindows server, use new keywords and qualifiers with SET ACL and SHOW ACL, use expanded SET HOST/DTE functions and subcommands, use symbol scoping, set characteristics for the VT400 family of terminals, and control and monitor specific processors and VAXft 3000 systems.
DCL lexical functions	New and enhanced functions return information about cluster identification numbers, device names, account status, verb scoping state, volume shadowing status, vector processing, active and recognized CPUs in an SMP system, and symbol creation (by other lexical functions).
EVE text editor	Enhancements include box editing, replacement of tab characters with spaces, and new qualifiers that let you edit large files and specify either the character-cell or DECwindows interface.
System Messages	New or modified messages are now available within specific VMS facilities and online Help.

(continued on next page)

Table 1–1 (Cont.) Summary of VMS Version 5.4 Software Features

VMS Version 5.4 General User Features	
DECwindows User	You can now set another session language or change the target screen on the Session Manager, view PostScript files with the CDA Viewer, change to hexadecimal or octal mode in Calculator, use new File, Customize, and Help menus for interacting with Clock, and use DECwindows Mail to display PostScript files.
VMS Version 5.4 System Management Features	
AUTOGEN	This command procedure now includes support for parameter name validation, SYS$SYSTEM:AGEN$PARAMS.REPORT (a new file that replaces AGEN$FEEDBACK.REPORT), reading external parameter files, controlling the size of page and swap files, new feedback parameters, new defined process logical names, a new technique for running AUTOGEN in batch mode, and the ability to use MAIL to send AGEN$PARAMS.REPORT.
UETP	Enhancements to the User Environment Test Package include loading and testing of all installed and enabled vector processors, testing of the VAX Vector Instruction Emulation Facility (VVIEF), and support for the RRD40 compact disc drive, including SCSI disk configurations.
SYSMAN Utility	Enhancements let you run a SYSMAN command procedure, define keys, spawn a subprocess, use DCL verification, and use loadable image commands.
VAXcluster software	Enhancements include CI architecture extensions that allow multiple CI interfaces per CPU and multiple star couplers per VAXcluster system; MSCP server load sharing; and preferred path support for DSA disks (including RA series disks and disks accessed through the MSCP server).
SYSGEN Utility	Enhancements include a new parameter for MicroVAX and VAXstation configurations that include third-party Small Computer System Interface (SCSI) devices, new parameters that support site-specific password policies, and new SHOW commands that display information such as bus identification statistics, device addresses mapped in the I/O space for the VAXBI bus, and device addresses mapped in the I/O space for the XMI bus.
Error Log Utility	Enhancements include support for VAXft 3000 device types, new device-class and entry-type keywords (to support vector processing and VAX 9000 systems) used with the /EXCLUDE and /INCLUDE qualifiers, and support for the new /NODE qualifier, which lets you produce a report of error log entries for specific nodes in a VAXcluster.
System Security	System security enhancements enable you to implement a site-defined password policy by screening new passwords and specifying password algorithms. This support includes enhancements to DCL commands, the SYSGEN Utility, the SYSMAN Utility, and system services. See Chapter 14 for more information.

(continued on next page)

Table 1–1 (Cont.) Summary of VMS Version 5.4 Software Features

VMS Version 5.4 System Management Features	
LMCP Utility	The new Log Manager Control Program Utility (LMCP) lets the system manager create and manage transaction log files in a DECdtm services environment. See Chapter 15 for a complete description of this new utility.
Monitor Utility	Enhancements include support for vector processing with the new MONITOR VECTOR command and VECTOR class, and support for DECdtm services with the new MONITOR TRANSACTION command and TRANSACTION class.
NCP Utility	The Network Control Program Utility now includes support for a new line and circuit name specific to the VAXft 3000 system.
VMS Volume Shadowing	VMS Volume Shadowing phase II includes support for distributed, clusterwide shadowing of all MSCP-compliant DSA disks (with the same number of logical blocks) and shadowing of all DSA devices.
VMS Version 5.4 Programming Features	
VMS Debugger	Enhancements to the debugger's command and DECwindows interfaces let you debug programs containing VAX vector instructions.
Linker Utility	A new command line qualifier, /BPAGE, lets you specify larger page sizes.
Mail Utility routines	New callable mail routines let you create applications that can perform a variety of Mail Utility functions and communicate with users on remote nodes connected to the system with DECnet–VAX.
System Services	New and enhanced system services support DECdtm services, system security enhancements, vector processing, volume shadowing, volume initialization, and the procedure for creating site-specific loadable images.
Run-Time Library	New parallel processing (PPL$) routines let you inform the PPL$ facility when a new caller is forming or joining a parallel application, implement work queues, delete a PPL$ application or object, set and adjust a semaphore maximum, disable event notification, or read a spin lock state.
	New and enhanced mathematics routines (MTH$) let you manipulate and perform operations on vectors.
RMS	Enhancements provide asynchronous support for process-permanent files, an increase in the local buffer maximum, access-mode protection for RMS services and for specific data structures and their associated I/O buffers, and the ability for all applications to selectively suppress updates to the Expiration Date and Time, using XAB$_NORECORD XABITM.
I/O Drivers	Enhancements include support for the Pseudoterminal driver (FTDRIVER) and Shadow Set Virtual Driver (SHDRIVER), modifications to the itemlist read function of the I/O status block (IOSB) and to the itemlist terminal driver read verify operations for the TRIM$_MODIFIERS item code, and the addition of three new ACP-QIO functions.

(continued on next page)

Table 1–1 (Cont.) Summary of VMS Version 5.4 Software Features

VMS Version 5.4 Programming Features	
System Dump Analyzer	New qualifiers to the SHOW PROCESS command let you display statistics about an image (/IMAGE) or about the values of the registers from the process's vector context area (/VECTOR_REGISTERS).
Device Support	Enhanced support includes VAX 9000 and VAX 6000 series systems. Programmers can write and debug driver software for non-Digital-supplied devices attached to a VAX 9000 system.
VAXTPU	Enhancements include work file support, a qualifier you can use to specify either character-cell or DECwindows interface, and new built-in procedures, including GET_INFO, that support journal recovery, pop-up menus, column context values for a buffer, markers within a buffer, and scrolling.
RMS Journaling	Enhancements support DECdtm services as well as existing applications and affect the Recovery Unit Facility (RUF), network support of remote files, RMS record streams, the RMS Detached Recovery server, placement of recovery unit journals, and access of files in a mixed-version cluster.
VMSINSTAL	A new data-file parameter (P4) in the Software Product Kit Building Procedure (SPKITBLD.COM) lets you specify the name of a data file. New callbacks affect messages displayed—and booting procedures required—during product installations, and how you obtain a system-generated or installer-specified password.
DECwindows Programming	Enhancements include new programming examples in the DECW$EXAMPLES directory, new support for the XUI Toolkit color mixing widget (both the Hue Lightness Saturation and Red, Green, Blue color models), support for the Display PostScript system (which provides text and image display capability for bitmapped workstations), and CDA Viewer support for PostScript files, Adobe Font metrics, and DECmath fonts.

2 Introduction to Vector Processing

The VMS Version 5.4 operating system supports vector processing on VAX 9000 series and VAX 6000-400 series computers. This chapter describes how vector processing works, how to manage resources, and how to write programs within a vector processing environment. The following sources in this manual and in other documents also describe aspects of VMS Version 5.4 vector processing support:

- Chapter 9 (of this manual) and the *VMS Version 5.4 Upgrade and Installation Manual* describe modifications to UETP.
- Chapter 4 (of this manual) and the *VMS DCL Dictionary* describe new and modified DCL commands, qualifiers, and lexical functions.
- Chapter 19 (of this manual) and the *VMS Debugger Manual* describe how to debug vectorized programs.
- Chapter 22 describes new and modified system services.
- The *VMS RTL Mathematics (MTH$) Manual* describes new and modified RTL mathematics routines.

2.1 Overview of the Vector Processing Environment

A single data item having one value is known as a **scalar**. A group of related scalar values, or elements, all of the same data type is known as a **vector**.

Traditional scalar computers operate only on scalar values and must process vector elements sequentially. Vector computers, on the other hand, recognize vectors as native data structures and can operate on an entire vector with a single vector instruction.

A vector processor can routinely process a vector four to five times faster than a traditional computer using only scalar instructions can. Vector processors gain this speed advantage over scalar processors by their use of special hardware techniques designed for the fast processing of streams of data. These techniques include data pipelining, chaining, and other forms of hardware parallelism in memory and in arithmetic and logical functional units. Pipelined functional units allow the vector processor to overlap the execution of successive computations with previous computations. Chaining allows the results of one instruction to be forwarded to another before the first instruction has been completely processed.

2.1.1 VAX Vector Processing Systems

An extension to the VAX architecture defines an optional design for integrated vector processing that has been adopted by several VAX processing systems. The VAX vector architecture includes 16 64-bit vector registers (V0 through V15), each containing 64 elements; vector control registers, including the vector count register (VCR), vector length register (VLR), and vector mask register (VMR); vector functional units; and a set of vector instructions. VAX vector instructions transfer data between the vector registers and memory, perform integer and floating-point arithmetic, and execute processor control functions. A more detailed description of the VAX vector architecture, vector registers, and vector instructions appears in the *VAX MACRO and Instruction Set Reference Manual.*

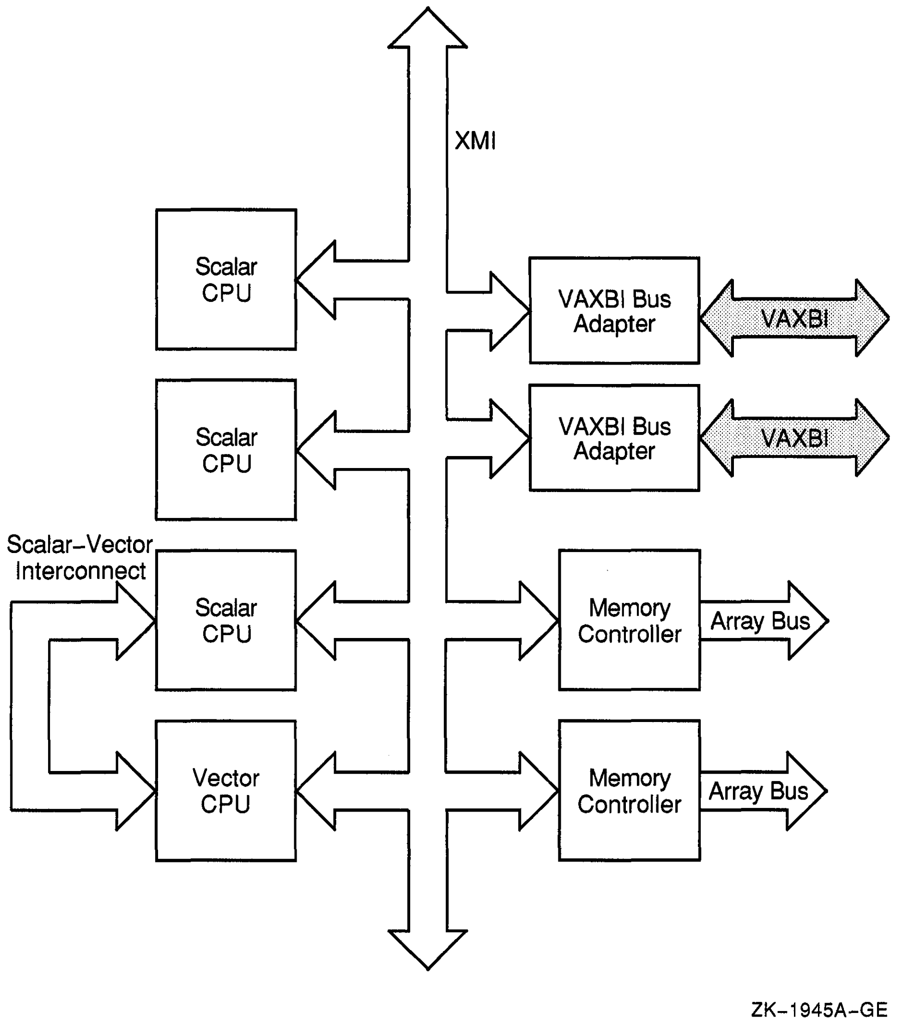
Those VAX systems that comply with the VAX vector architecture are known as **vector-capable** systems.

A VAX vector processing system configuration includes one or more integrated scalar-vector processor pairs, or **vector-present processors**. Such a configuration can either be symmetric, including a vector coprocessor for each scalar, or asymmetric, incorporating additional scalar-only processors. Depending on the model of the VAX vector processing system, the scalar and vector CPUs of vector-present processors can be either a single, integral physical module or separate, physically-independent modules. In either case the scalar and vector CPUs are logically integrated, sharing the same memory and transferring data over a dedicated, high-speed internal path. Because the CPUs are thus tightly-coupled, use of the vector CPU foregoes the expense of I/O operations.

The scalar and vector CPUs operate asynchronously with respect to each other. The scalar CPU fetches and decodes all instructions issued by the current image and executes all scalar instructions. When it encounters a vector instruction, the scalar CPU passes it to the vector CPU. While the vector CPU is executing this instruction, the scalar CPU continues to fetch and decode instructions, executing any scalar instruction it encounters and sending any vector instructions it encounters to the vector CPU. The vector processor maintains a queue of pending instructions in which it places instructions it receives while it is busy. The VMS operating system and its vectorizing compilers help ensure that the activities of both scalar and vector CPUs are synchronized. (Section 2.3.7 describes those situations in which vectorized VAX MACRO programs must enforce scalar and vector CPU synchronization.)

Certain VAX system models offer a vector processing option. In VAX 6000-400 series systems, the vector CPU occupies a slot on the memory interconnect; the scalar-vector interconnect joins it to the scalar CPU, which resides in an adjacent slot (see Figure 2–1). In VAX 9000 series systems, the vector processor is an integral part of the CPU, as shown in Figure 2–2.

Figure 2–1 VAX 6000–400 Series Vector-Present Processor Configuration

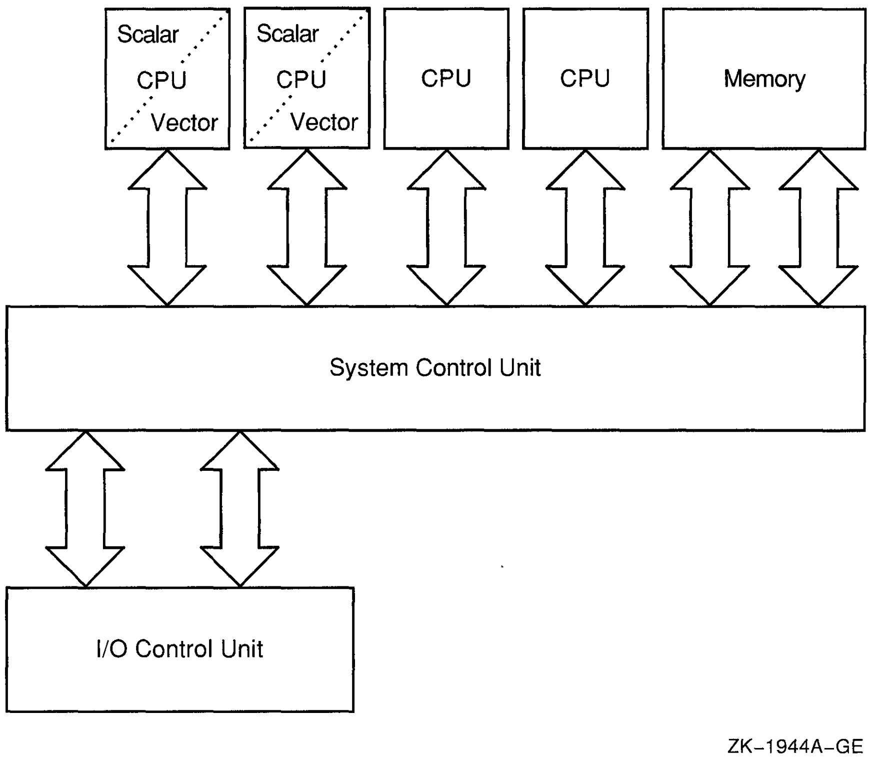


Like VAX scalar processing systems, a VAX vector processing system can participate as a member of a VAXcluster or as a node in a network, or it can be run as a standalone system.

2.1.2 Vectorized Programs

The benefits of vectorization depend, to a large degree, on the specific techniques and algorithms of an application. CPU-intensive applications involving repeated operations on groups of simple elements are well-suited to vectorization. VAX vector processing systems are particularly beneficial in the fields of seismic analysis, weather forecasting, molecular modeling, computational fluid dynamics, signal processing, financial modeling, and finite element analysis.

Figure 2–2 VAX 9000 Series Vector-Present Processor Configuration



There are several methods you can use to produce a vectorized program in a VMS system.

Most applications that benefit from vector processing can be developed as scalar programs in a high-level language and then submitted to a vectorizing compiler for that language. A **vectorizing compiler**, such as the VAX Fortran High Performance Option (HPO), can recognize sections of code within a program, usually inside formal loops, that can be vectorized. It analyzes data dependences, identifies other inhibitors to vector processing, and restructures code sequences to allow the compiler to generate optimized VAX vector instruction sequences.

Additionally, applications can be vectorized by a call to the vectorized routines in the VMS Run-Time Library mathematics facility (RTL MTH$) or to the vectorized routines within the optional DIGITAL Extended Math Library (DXML):

- The vectorized RTL MTH$ routines that can be called by a high-level language application include the Level 1 Basic Linear Algebra Subroutines (BLAS) and First-Order Linear Recurrence (FOLR) routines. In addition, VAX vectorizing compilers (and programs written in VAX MACRO) can generate calls to vectorized versions of the standard scalar RTL MTH$ routines. (The vectorized RTL MTH$ routines are introduced in Section 2.3.1 and fully discussed in the *VMS RTL Mathematics (MTH$) Manual.*)

- The DIGITAL Extended Math Library (DXML) is an optional software product that provides additional vectorized mathematics routines such as BLAS Level 1-extended, 2, and 3, plus signal processing routines.

Finally, those programs that require strict control over the VAX vector hardware can be written in VAX MACRO and use the VAX vector instructions directly.

The terms **vectorized program**, **vectorized application**, and **vectorized image** all refer to programs produced by a vectorizing compiler, programs that call one or more vectorized routines, and programs written in VAX MACRO that issue VAX vector instructions. A vectorized image from any of these categories eventually results in the execution of one or more vector instructions that transform its process into a vector consumer.

See Section 2.3 for an overview of the VMS vector processing programming environment.

2.1.3 VMS Support for Vector Processing

The VMS operating system provides fully-shared, multiprogramming support for VAX vector processing systems. By default, VMS loads vector processing support code when initializing a VAX system that includes vector-present processors but does not load it when initializing vector-absent systems. (A system manager can control this behavior by using the SYSGEN parameter VECTOR_PROC, as described in Section 2.2.1.) The presence of vector support code in a system has little effect on processes running in a scalar-only system or on scalar processes running in a vector-present system. If many processes must simultaneously compete for vector processor resources in a system, the system manager can maintain good performance by adjusting system resources and process quotas as indicated in Section 2.2.3.1.

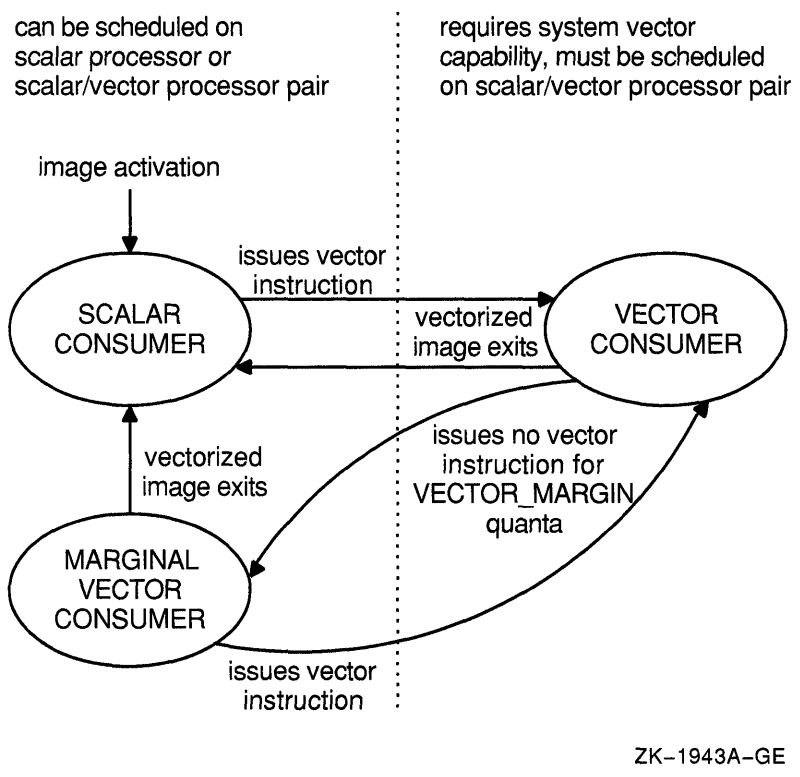
The VMS operating system makes the services of the vector processor available to system users by means of a software abstract known as a **capability**. A system manager can restrict the use of the vector processor to users holding a particular identifier by associating an access control list (ACL) with the vector capability object. (See Section 2.2.4 for additional information.)

2.1.3.1 Life of a Vector Consumer

As shown in Figure 2–3, a process begins execution as a **scalar consumer**, partaking of the resources of a scalar processor or the scalar component of a vector-present processor.

When the image executing within the process's context issues its first vector instruction, VMS marks the process as requiring the system's vector capability. It also allocates sufficient system nonpaged dynamic memory in which to store this process's **vector context**. The vector context of a process consists of the contents of the vector registers V0 through V15, the contents of the vector control registers (VCR, VLR, and VMR), the vector processor status, and the vector exception state.

Figure 2–3 Life of a Vector Consumer


can be scheduled on scalar processor or scalar/vector processor pair
requires system vector capability, must be scheduled on scalar/vector processor pair
image activation
SCALAR CONSUMER
issues vector instruction
VECTOR CONSUMER
vectorized image exits
vectorized image exits
issues no vector instruction for VECTOR_MARGIN quanta
MARGINAL VECTOR CONSUMER
issues vector instruction
ZK–1943A–GE

A process requiring the vector capability and having a vector context is known as a **vector consumer**. VMS *must* schedule a vector consumer on a vector-present processor. As long as it remains a vector consumer, a process is effectively prohibited from executing on any scalar processor in the system.

However, over the course of its execution, a typical vectorized image issues sequences of scalar instructions intermixed with sequences of vector instructions. For those periods in which it performs scalar operations exclusively, a process can relinquish its need for the vector capability and become eligible for execution on any processor in the system. VMS preserves the vector context of any such **marginal vector consumer** in the expectation that it will eventually issue another vector instruction and again become a vector consumer.

In a system in which many vector consumers are competing for the vector processor, the dynamic transition of vector consumers to marginal vector consumers (and back again) allows VMS to more efficiently distribute vector processor resources and enhances the performance of vectorized applications. Note that a system manager can control the transition of a vector consumer to a marginal vector consumer by setting the SYSGEN parameter VECTOR_MARGIN (as discussed in Section 2.2.3.2).

Ultimately, a vector consumer or marginal vector consumer reverts to being a scalar consumer when the vectorized image it is executing exits.

In the course of system activity, another process could preempt the execution of a vector consumer on a vector-present processor. When this occurs, VMS immediately saves the vector consumer's scalar context, as it does for traditional scalar processes. However, VMS allows its vector context to remain intact in the vector CPU. Depending upon the nature of the intervening processes scheduled on that processor, VMS, in most cases, tries to reschedule a vector consumer on the vector-present processor on which it was last scheduled.

Because scalar consumers and marginal vector consumers do not use the vector CPU, they do not disturb the vector context of the latest vector consumer on the vector-present processor on which they are scheduled. If only processes of these types were scheduled on the vector-present processor since the vector consumer last ran, the vector consumer can resume execution on that processor without the overhead associated with a restoration of its vector context from memory. This is known as **"fast" vector context switch**.

Other vector consumers, however, do use the vector CPU. When placing a vector consumer into execution on a vector-present processor, VMS stores in memory the vector context of the processor's latest vector consumer. When it later reschedules this vector consumer, VMS can place it into execution on any vector-present processor in the system, but it must restore its vector context from memory. This is known as **"slow" vector context switch**.

Slow vector processing context switches are most likely when there are more vector consumers than vector-present processors in the systems. A system manager can adjust system parameters (including the VECTOR_ MARGIN parameter) and system resources to help reduce the number of slow vector context switches as described in Section 2.2.

2.1.3.2 VAX Vector Instruction Emulation Facility (VVIEF)

The VAX Vector Instruction Emulation Facility (VVIEF) is a standard feature of the VMS operating system that allows vectorized applications to be written and debugged on a VAX system in which vector processors are not available. VVIEF emulates the VAX vector processing environment, including the nonprivileged VAX vector instructions and the VMS vector system services (described in Sections 2.3.2, 2.3.3, and 2.3.4). Use of VVIEF is restricted to user mode code.

VVIEF is strictly a program development tool and *not* a run-time replacement for vector hardware. There is no performance benefit from vectorizing applications to run under VVIEF; vectorized applications running under VVIEF typically execute five times slower than their scalar counterparts.

VMS supplies the VVIEF bootstrap code as an executive loadable image. The system manager invokes the command procedure SYS$UPDATE:VVIEF$INSTAL.COM to cause VMS to load VVIEF at the *next* system boot and each successive system boot. Note that, in the presence of VMS vector support code, VVIEF remains inactive. Although it is possible to prevent the loading of VMS vector support code in a vector-present system (see Section 2.2.1) and activate VVIEF, there are few benefits. Should the only scalar-vector processor pair in the system fail, the execution of preempted vectorized applications will not be resumed under the emulator.

See Section 2.2.6 for additional information on loading and unloading VVIEF.

2.2 Managing the Vector Processing Environment

Managing a VAX vector processing system includes the following tasks:

- Loading the VMS vector processing support code
- Configuring a vector processing system
- Managing processes requiring the system's vector processing resources
- Obtaining information about the status and use of the system's vector processing resources
- Loading the VAX vector emulation facility (VVIEF) bootstrap code

This section describes the features VMS has introduced or enhanced to facilitate the accomplishment of these tasks. It concludes with a list of messages VMS uses to report information about the condition of the vector processing system.

2.2.1 Loading the VMS Vector Processing Support Code

By default, in a VAX vector processing system, VMS automatically loads the vector processing support code at boot time. You can override the default behavior by setting the static system parameter VECTOR_PROC as described in Table 2–1.

Table 2–1 Settings of VECTOR_PROC System Parameter

Value	Result
0	Do not load the vector processing support code, regardless of the system configuration.
1	Load the vector processing support code if there is at least one vector-present processor. This is the default value.
2	Load the vector processing support code if the system is vector capable. This setting is most useful for a system in which processors have separate power supplies. With this setting, you can reconfigure a vector processor into the system without rebooting the VMS operating system.

2.2.2 Configuring a VMS Vector Processing System

You can add or remove a vector-present processor to or from a VMS multiprocessing configuration at boot time by using the SYSGEN parameter SMP_CPUS or at run time by using the DCL commands START /CPU and STOP/CPU. Note that VMS treats the scalar and vector CPU components of a vector-present processor as a single processor, starting them and stopping them together.

At boot time, the setting of the SYSGEN parameter SMP_CPUS identifies which secondary processors in a VMS multiprocessing system are to be configured, including those processors that are vector present. (VMS always configures the primary processor.) The default value of –1 boots all available processors, scalar and vector-present alike, into the configuration. (See the *VMS System Generation Utility Manual* for additional information on this parameter.) Note that, prior to starting a vector-present processor, you should make sure that the vector processing support code (see Section 2.2.1) is loaded at boot time. Otherwise, processes will only be able to use the scalar CPU component of the vector-present processor.

To bring secondary processors into a running VMS multiprocessing system, you use the DCL command START/CPU. To remove secondary processors from the system, use the STOP/CPU commands. Again, you must make sure that the vector processing support code has been loaded at boot time for the vector CPU component of vector-present processors started in this way to be utilized.

However, note that if you enter a STOP/CPU command that would cause the removal of a vector-present processor that is the sole provider of the vector capability for currently active vector consumers, the command fails and generates a message. In extreme cases, such as the removal of a processor for repair, you can override this behavior by entering the command STOP/CPU/OVERRIDE. This command stops the processor, despite stranding processes.

When a STOP/CPU/OVERRIDE command is entered for a vector-present processor, or when a vector-present processor fails, VMS puts all stranded vector consumers into a CPU-capability-wait (RSN$_CPUCAP) state until a vector-present processor is returned to the configuration. To any other process that subsequently issues a vector instruction (including a marginal vector consumer), VMS returns a "requested CPU not active" message (CPUNOTACT).

See the *VMS DCL Dictionary* for additional information on the START/CPU and STOP/CPU commands.

2.2.3 Managing Vector Processes

As described in Section 2.1.3, VMS scheduling algorithms automatically distribute vector and scalar processing resources among vector consumers, marginal vector consumers, and scalar consumers. However, VAX vector processing configurations vary in two important ways:

- The amount of vector processing activity the configuration must accommodate
- The number of vector-present processors available in the configuration to service vector processing needs

In a configuration in which there are more vector consumers in a system than there are scalar-vector processor pairs to service them, vector consumers share vector-present processors according to process priority. At a given priority, VMS schedules vector consumers on a vector-present processor in a round-robin fashion. Each time VMS must schedule a new vector consumer on a vector-present processor, it must save the vector context of the current vector consumer in memory and restore the vector context of the new vector consumer from memory. When such "slow" vector context switches occur too frequently, a significant portion of the processing time is spent on vector context switches relative to actual computation.

Systems that have heavy vector processing needs should be adequately configured to accommodate those needs. There are, however, some mechanisms a system manager can use to tune the performance of an existing configuration.

2.2.3.1 Adjusting System Resources and Process Quotas

Systems in which several vector consumers are active simultaneously might experience increased paging activity as processes share the available memory. To reduce process paging, you might need to use the Authorize Utility to adjust the working set limits and quotas of the processes running vectorized applications. An increase of the process maximum working set size (SYSGEN parameter WSMAX) might also be necessary. Additionally, a vectorized application can use the Lock Pages in Working Set system service (SYS$LKWSET) to enhance its own performance.

VMS allots to each vector consumer 8 kilobytes of system nonpaged dynamic memory in which VMS stores vector context information. Depending on how many vector consumers are active in the system simultaneously, you might need to adjust the SYSGEN parameter NPAGEDYN. To determine the current usage of nonpaged pool, use the DCL command SHOW MEMORY/POOL/FULL, which displays the current size of nonpaged pool in bytes.

See the *VMS System Generation Utility Manual* and the *VMS Authorize Utility Manual* for additional information on these mechanisms.

To obtain optimal performance of a VAX vector processing system, you should take some care to set up generic batch queues that avoid saturating the system's vector resources. If a queue contains more active vectorized batch jobs than there are vector-present processors in the system, a significant portion of the processing time will be spent on vector context switches.

The recommended means for dispatching vectorized batch jobs to a VAX vector processing system is to set up a separate queue (for instance, VECTOR_BATCH) with a job limit equal to the number of vector-present processors in the system. When submitting vectorized batch jobs, users should be encouraged to submit them to this generic vector processing batch queue.

2.2.3.2 Distributing Scalar and Vector Resources Among Processes

As a vector consumer, a process must be scheduled only on a vector-present processor. If the image the process is executing issues only scalar instructions for a period of time and must share the scalar-vector processor pair with other vector consumers, its inability to run on an available scalar processor could hamper its performance and the overall performance of the system.

By default, VMS assumes that, if a vector consumer has not issued a vector instruction for a certain period of time, it is unlikely that it will issue a vector instruction in the near future. VMS relinquishes this process's need for the vector capability, classifying it as a marginal vector consumer.

In an asymmetric vector processing configuration, detection of marginal vector consumers achieves the following desirable effects:

- Because a marginal vector consumer is eligible to run on a larger set of processors, its response time will improve.
- The scheduling of marginal vector consumers on scalar processors reduces the contention for vector-present processors.
- Because vector consumers issuing vector instructions are more likely the be scheduled on vector-present processors, the vector CPU is more efficiently used.

A system manager uses the SYSGEN parameter VECTOR_MARGIN to establish the interval of time at which VMS checks the status of all vector consumers. The VECTOR_MARGIN parameter accepts an integer value between 1 and –1 (FFFFFFFF_{16}). This value represents a number of consecutive process quanta (as determined by the SYSGEN parameter QUANTUM). If the process has not issued any vector instructions in the specified number of quanta, VMS declares it a marginal vector consumer. A value of –1 disables the checking mechanism.

The default value of the VECTOR_MARGIN parameter is 100_{10}.

2.2.4 Restricting Access to the Vector Processor by Using ACLs

Using the SET ACL and SHOW ACL commands, a system manager can restrict the use of the vector processor to users holding a particular identifier. By associating an access control list (ACL) with the vector capability, a university might limit use of the vector processor to faculty and students in an image processing course, or a service bureau might charge users for access to the vector capability, time spent on the vector processor, or both.

When using either the SET ACL or SHOW ACL command with Version 5.4 of the VMS operating system, the system manager can specify a new object type, CAPABILITY, as the argument to the /OBJECT_TYPE qualifier. This object type is a system capability, such as the ability to process VAX vector instructions. Currently, the only defined object name for the CAPABILITY object type is VECTOR. Therefore, when using the SHOW ACL or SET ACL command, the system manager must supply the capability name (VECTOR) as the argument to the object type, as in the following examples. (For additional information on the SET ACL and SHOW ACL commands, see the *VMS DCL Dictionary*.)

The following DCL command establishes one or more access control entries (ACEs) on the vector capability.

```
$  SET ACL/OBJECT=CAPABILITY VECTOR/ACL[=(ace[,...])]
```

Note that you must be in the SYSTEM user category (as described in *VMS DCL Concepts Manual*) to set an ACL on the vector capability.

The following DCL command displays the ACL on the vector capability.

```
$  SHOW ACL/OBJECT=CAPABILITY VECTOR
```

Note that the ACL is on the vector capability, not on the use of any or all vector-present processors in the system. For this reason, VMS can still schedule processes without permission to use the vector capability on a vector-present processor. However, these processes can use only the scalar CPU component of the processor and cannot execute vector instructions. Likewise, because the ACL is on the vector capability and not on a vector-present processor, you cannot establish an ACL to force long-running jobs to a specific processor.

The Change ACL ($CHANGE_ACL) and Check Access ($CHECK_ACCESS) system services provide means for setting and removing ACLs on the VECTOR capability and for checking a process's ability to use vector processing resources. See Section 22.5 for additional information.

2.2.5 Obtaining Information About a Vector Processing System

You can obtain information about the status of the vector processing system and the use of the system by individual processes through various means, including:

- The DCL lexical functions F$GETJPI and F$GETSYI
- The DCL command SHOW CPU
- The DCL commands SHOW PROCESS and LOGOUT/FULL
- The Accounting Utility (ACCOUNTING)
- The Error Log Utility (ERROR LOG)
- The Monitor Utility (MONITOR)

2.2.5.1 DCL Lexical Functions F$GETJPI and F$GETSYI

The DCL lexical function F$GETJPI accepts the following items and returns the corresponding information regarding the vector status of a specified process:

Item	Return Type	Information Returned
FAST_VP_SWITCH	Integer	Number of times this process has issued a vector instruction that resulted in an inactive vector processor being enabled without the expense of a vector context switch
SLOW_VP_SWITCH	Integer	Number of times this process has issued a vector instruction that resulted in an inactive vector processor being enabled with a full vector context switch
VP_CONSUMER	Boolean	Flag indicating whether the process is a vector consumer
VP_CPUTIM	Integer	Total amount of time the process has accumulated as a vector consumer

The DCL lexical function F$GETSYI accepts the following items and returns the corresponding information regarding the status of the vector processing system:

Item	Return Type	Information Returned
VP_NUMBER	Integer	Number of vector processors in the system
VP_MASK	Integer	Mask indicating which processors in the system have vector coprocessors
VECTOR_EMULATOR	Integer	Flag indicating the presence of the VAX vector instruction emulator facility (VVIEF) in the system

See the *VMS DCL Dictionary* for additional information about the DCL lexicals F$GETJPI and F$GETSYI.

2.2.5.2 SHOW CPU Command

The SHOW CPU/FULL command lists the capabilities of the specified CPU. The manager of a VAX vector processing system can issue this command to determine the presence of the vector capability in the system prior to executing a STOP/CPU command.

See the *VMS DCL Dictionary* for additional information about the SHOW CPU command.

2.2.5.3 SHOW PROCESS and LOGOUT/FULL Commands

If the target process has accrued any time as a vector consumer scheduled on a vector-present processor, the DCL commands SHOW PROCESS and LOGOUT/FULL display the elapsed vector CPU time and the charged vector CPU time, respectively.

To accumulate vector CPU time, a process must be a vector consumer (that is, require the system vector capability) and be scheduled on a vector-present processor. VMS still charges the vector consumer vector CPU time, even if, when scheduled on the vector-present processor, it does not actually use the vector CPU. Note that, because scalar consumers and marginal vector consumers do not use the vector CPU, they do not accrue vector CPU time, even when scheduled on a vector-present processor.

See the *VMS DCL Dictionary* for additional information about the SHOW PROCESS and LOGOUT commands.

2.2.5.4 Vector Processing Support Within the VMS Accounting Utility (ACCOUNTING)

In its full listing format, the VMS Accounting Utility displays the vector CPU time accumulated by a process or an image during its life span.

A process accumulates vector CPU time while it is a vector consumer (that is, requiring the system vector capability) and it is scheduled on a vector-present processor. VMS still charges the vector consumer vector CPU time, even if, when scheduled on the vector-present processor, it does not actually use the vector CPU. Note that, because scalar consumers and marginal vector consumers do not use the vector CPU, they do not accrue vector CPU time, even when scheduled on a vector-present processor.

An image accrues vector CPU time while it is executing within the context of a vector consumer on a vector-present processor. Because VMS marks all processes, including vector consumers, as scalar consumers at image rundown, it is impossible for an image that issues only scalar instructions to accumulate vector CPU time.

The /SORT qualifier to the ACCOUNTING command accepts the VECTOR_PROCESSOR keyword and sorts the accounting records according to ascending or descending vector CPU time. The /REPORT qualifier also accepts the VECTOR_PROCESSOR keyword and produces a summary report of vector CPU usage.

See Section 2.3.8 for a description of the vector CPU time field in the ACCOUNTING resource packet. The *VMS Accounting Utility Manual* provides a complete description of the VMS Accounting Utility.

2.2.5.5 Vector Support Within the Error Log Utility (ERROR LOG)

With Version 5.4 of the Error Log Utility, the /INCLUDE qualifier to the ANALYZE/ERROR_LOG command accepts the device-class keyword VECTOR, which produces an error log report that includes vector processing errors. (Specifying the VECTOR keyword with the /EXCLUDE qualifier excludes vector processing errors from the error log report.)

2.2.5.6 Vector Support Within the VMS Monitor Utility (MONITOR)

With Version 5.4 of the VMS Monitor Utility, the new MONITOR VECTOR command initiates monitoring of the VECTOR class and displays the number of 10-millisecond clock ticks per second in which one or more vector consumers have been scheduled on each currently configured vector processor.

See Section 16.3 for a complete description of the MONITOR VECTOR command and the VECTOR class. See Section 2.3.9 and Section 16.4 for related information about the VECTOR class record and format. Refer to the *VMS Monitor Utility Manual* if you need additional information about the VMS Monitor Utility.

2.2.6 Loading the VAX Vector Instruction Emulation Facility (VVIEF)

The VAX Vector Instruction Emulation Facility (VVIEF) is a standard feature of the VMS operating system that allows vectorized applications to be written and debugged on a VAX system in which vector processors are not available. VVIEF is intended strictly as a program development tool and *not* as a run-time replacement for vector hardware. There is no performance benefit from vectorizing applications to run under VVIEF; vectorized applications running under VVIEF typically execute five times slower than their scalar counterparts.

VMS supplies the VVIEF bootstrap code as an executive loadable image. To cause VMS to load VVIEF at the *next* system boot and at each subsequent system boot, invoke the command procedure SYS$UPDATE:VVIEF$INSTAL.COM. To unload VVIEF, invoke the command procedure SYS$UPDATE:VVIEF$DEINSTAL.COM and reboot the system. You can determine the presence or absence of VVIEF on a system by issuing the following DCL commands:

```
$  X = F$GETSYI("VECTOR_EMULATOR")
$  SHOW SYMBOL X
X = 1    Hex = 00000001  Octal = 0000000001
```

A return value of 1 indicates the presence of VVIEF; a value of 0 indicates its absence.

Note that, although VVIEF might be loaded into the system, in the presence of VMS vector support code, it remains inactive. Although it is possible to prevent the loading of VMS vector processing support code in a vector-present system (see Section 2.2.1) and activate VVIEF, there are few benefits. Should the only vector-present processor in the system fail, the execution of preempted vectorized applications will not resume under the emulator.

2.2.7 System Messages Related to Vector Processing Activities

Table 2–2 lists the system messages that might result from vector activity in a VAX vector processing system. It describes the conditions that might have resulted in the message and suggests how you can repair the condition causing the error.

For information on how VMS reports exception conditions to condition handlers, see Section 2.3.6.

Table 2–2 System Messages Relating to Vector Processing

Message	Message Text	Description and Recovery
ACCVIO	access violation, reason mask = xx, virtual address = location, PC = location, PSL = xxxxxxx	See the *VMS System Messages and Recovery Procedures Reference Manual* for a description of the ACCVIO message. The lowest three bits of the reason mask indicate that an instruction has caused a length violation (bit 0), referenced the process page table (bit 1), and attempted a read or modify operation (bit 2). VMS defines two additional bits to reflect vector processing memory management exceptions: a vector operation on an improperly-aligned vector element in memory (bit 3) and vector instruction reference to an I/O space address (bit 4).
BADCONTEXT	invalid or corrupted context encountered	The vector state of a mainline routine as saved in process P1 space has been corrupted and cannot be restored. A call to the Restore Vector State system service (SYS$RESTORE_VP_STATE) can result in this error, if some coding error has overwritten the saved vector state. (See Chapter 22 for more on this system service.)
CPUNOTACT	requested CPU not active	The current process requires system capabilities that are not available or no longer available among the active processors in the system. If this message is associated with a vector disabled (VECDIS) status code, a vector-present processor within the system is not available, has failed, or has been removed from the system. See Section 2.2.2.
EXQUOTA	exceeded quota	If this message is associated with a vector disabled (VECDIS) status code, the process's paging file quota prohibits the allocation of sufficient process memory for storing its mainline vector state. (See Section 2.2.3.1.)
ILLVECOP	illegal vector opcode fault, opcode='xx', PC='location', PSL='xxxxxxxx'	An operation code designated as an illegal vector opcode by the VAX architecture has been encountered during the execution of an image. See Section 2.3.6 and the *VAX MACRO and Instruction Set Reference Manual* for additional information about this exception.
IMGVEXC	image exiting with pending vector exceptions	An exception has resulted due to the execution of a vector instruction issued by an image, but the image has exited before the exception could be delivered. See Section 2.3.7.4.
INSFMEM	insufficient dynamic memory	If this message is associated with a vector disabled (VECDIS) status code, the current process has issued a vector instruction, but insufficient system nonpaged dynamic memory exists to establish the process as a vector consumer. (See Section 2.2.3.1.)

(continued on next page)

Table 2–2 (Cont.) System Messages Relating to Vector Processing

Message	Message Text	Description and Recovery
INSFWSL	insufficient working set limit	If this message is associated with a vector disabled (VECDIS) status code, the process's current working set list limit does not allow its mainline vector state to be resident in memory. (See Section 2.2.3.1.)
NOPRIV	no privilege for attempted operation	If this message is associated with a vector disabled (VECDIS) status code, an ACL on the system's vector capability has prevented the process from executing vector instructions. (See Section 2.2.4.)
NOSAVPEXC	no saved vector exception for the exception-id	A call was made to the Restore Vector Processing State system service (SYS$RESTORE_VP_EXCEPTION) that specified a value for an exception ID that does not correspond to that of any saved vector exception state. (See Chapter 22 for more on this system service.)
VARITH	vector arithmetic fault, summary=xx, mask=xx, PC=location, PSL=xxxxxxxx	A vector operate instruction, executing within the current context, has resulted in a vector arithmetic trap. (See Section 2.3.6 for assistance in interpreting the exception summary mask, vector register mask, PC, and PSL.) Because arithmetic operations are performed in a substantially different manner on vectors than on scalars, the resolution of vector arithmetic exceptions requires some special techniques. (See Section 2.3.6 for information on the mechanisms by which exceptions are reported and identified.) One or a combination of several debugging strategies can help you determine which calculations resulted in the reported error or errors: • Recompile the source with the /DEBUG, /NOVECTOR, /CHECK=BOUNDS qualifiers; relink using the /DEBUG and /MAP qualifiers; and run the resulting scalar image with the /DEBUG qualifier. A scalar arithmetic exception should occur at the calculation that caused the original vector arithmetic exception. • Recompile the source using the /DEBUG, /LIST, and /VECTOR qualifiers; relink using the /DEBUG and /MAP qualifiers; and run the resulting image with the /DEBUG qualifier. (If the /ASSUME=NOACCURACY_SENSITIVE qualifier was used in the original compilation, specify /ASSUME=ACCURACY_SENSITIVE.) Use the SET VECTOR_MODE SYNCHRONIZED or the SYNCHRONIZE VECTOR_MODE debugger command to guarantee that all exceptions resulting from vector operations be delivered before the execution of the next scalar instruction. Step through the program, inspecting the contents of those vector registers that are involved in each vector operation.

(continued on next page)

Table 2–2 (Cont.) System Messages Relating to Vector Processing

Message	Message Text	Description and Recovery
		When a vector operate instruction causes an floating-point exception in a vector element, the exception result is encoded into the corresponding element of the destination register. When the destination vector register is the target of an EXAMINE/FLOAT debugger command, the debugger displays the decoded exception message in the associated vector element. When a vector operate instruction causes an integer overflow in a vector element, the corresponding element of the destination register contains a value that is larger than 32 bits, but of a different sign than the instruction's operands. When the destination vector register is the target of an EXAMINE debugger command, you must inspect each element for such results.
VASFUL	virtual address space is full	If this message is associated with a vector disabled (VECDIS) status code, insufficient process virtual address space exists to allow the current process's mainline vector state to be saved. (See Section 2.2.3.1.)
VECALIGN	access violation, reason mask = xx, virtual address = location, PC = location, PSL = xxxxxxx	The current process has issued a VAX vector memory access instruction that has attempted an operation on an improperly-aligned vector element. The VAX architecture requires that vector operands to vector memory access instructions be naturally aligned in memory. Longwords must be aligned on longword boundaries; quadwords must be aligned on quadword boundaries. See Section 2.3.6. and the *VAX MACRO and Instruction Set Reference Manual* for additional information.

(continued on next page)

Table 2–2 (Cont.) System Messages Relating to Vector Processing

Message	Message Text	Description and Recovery
VECDIS	vector disabled fault, code=xx, PC = location, PSL = xxxxxxx	The current process has issued a vector instruction that requires that a vector processor become active. Under normal circumstances, this event is not reported to a system user. However, if the vector processor was unavailable due to some previously unreported condition, the VECDIS message is issued in association with one of the following messages. • BADCONTEXT • CPUNOTACT • EXQUOTA • INSFMEM • INSFWSL • MCHECK • NOPRIV • VASFUL See the description of the associated message in this table and the *VMS System Messages and Recovery Procedures Reference Manual* for additional information on any specific error.

2.3 Programming in a Vector Processing Environment

Most applications that benefit from vector processing can be developed as scalar programs in a high-level language and then submitted to a vectorizing compiler for that language.

Additionally, applications can be vectorized by a call to the vectorized routines in the VMS Run-Time Library mathematics facility (RTL MTH$) or to the vectorized routines within the optional DIGITAL Extended Math Library (DXML):

- The vectorized RTL MTH$ routines that can be called by a high-level language application include the Level 1 Basic Linear Algebra Subroutines (BLAS) and First-Order Linear Recurrence (FOLR) routines. In addition, VAX vectorizing compilers (and programs written in VAX MACRO) can generate calls to vectorized versions of the standard scalar RTL MTH$ routines. (The vectorized RTL MTH$ routines are introduced in Section 2.3.1 and fully discussed in the *VMS RTL Mathematics (MTH$) Manual.*)
- The DIGITAL Extended Math Library (DXML) is an optional software product that provides additional vectorized mathematics routines such as BLAS Level 1-extended, 2, and 3, plus signal processing routines.

Finally, those programs that require strict control over the VAX vector hardware can be written in VAX MACRO and use the VAX vector instructions directly.

Use of high-level interfaces to VAX vector processing systems, such as the VAX Fortran High Performance Option (HPO) vectorizing compiler and the vectorized RTL MTH$ routines, provide a mechanism for quickly developing a vectorized program that conforms to the requirements of the VAX Procedure Calling and Condition Handling Standard and the VAX vector architecture. For instance, VAX vectorizing compilers and vectorized library routines automatically handle the complexities of properly handling scalar-vector synchronization, vector memory alignment, and the preservation of vector state across procedure calls. Additionally, the VAX Fortran HPO vectorizing compiler can recognize sections of code within a program, usually inside formal loops, that can be vectorized. It analyzes data dependences, identifies inhibitors to vector processing, and restructures code sequences to allow the compiler to generate optimized VAX vector instruction sequences.

By contrast, VAX MACRO programmers must themselves ensure that vector code conforms to the rules stated in the *VAX MACRO and Instruction Set Reference Manual* and Section 2.3.7.

If you must write a vectorized program in VAX MACRO, you should be aware of the following:

- You must specifically enable the processing of vector instructions by the VAX MACRO assembler by assembling with the /ENABLE or /NODISABLE qualifier to the MACRO command and supplying the keyword VECTOR. You can also explicitly enable the assembly of vector instructions by using the .ENABLE VECTOR directive.
- The VAX MACRO assembler parses the assembler notation form of vector instructions and produces binary code in the instruction stream form prescribed by the VAX vector architecture. The *VAX MACRO and Instruction Set Reference Manual* describes both vector instruction forms and presents the assembler notation form in its instruction pages.
- VAX MACRO programs must synchronize the vector CPU's memory references across procedure calls, as well as guarantee that pending vector exceptions are raised before crossing procedure boundaries. VAX MACRO programs must also ensure that the vector CPU's memory references are synchronized with the scalar CPU's memory references. Section 2.3.7 and the *VAX MACRO and Instruction Set Reference Manual* describes the mechanisms by which VAX MACRO code can comply with these requirements.
- The *VAX MACRO and Instruction Set Reference Manual* lists several additional restrictions, including the following:
  - VAX MACRO programs must naturally align vector operands to vector memory access instructions. Longwords must be aligned on longword boundaries; quadwords must be aligned on quadword boundaries.
  - VAX MACRO instructions cannot reference addresses in I/O space.

— Vector instructions cannot be issued at elevated interrupt priority levels (IPLs), specifically at or above IPL$_RESCHED. The vector disabled handler will force a system crash with the VPIPLHIGH bugcheck code ("IPL too high to use the Vector Facility") when user vector instruction is issued at or above IPL$_RESCHED.

The remainder of this section discusses the following topics:

- Vector routines in the MTH$ Run-Time Library
- Obtaining information about a vector processing system
- Releasing the vector processor
- Preserving and restoring a routine's vector state
- Issuing vector instructions at high IPLs
- Debugging a vector application
- Servicing vector processing exceptions
- Utilizing vector information contained within the informational packets generated by the VMS Accounting Utility and VMS Monitor Utility

2.3.1 Vector Routines in the MTH$ Run-Time Library

The RTL MTH$ facility provides three sets of routines that allow manipulation of vectors and perform operations on vectors:

- The Basic Linear Algebra Subroutines (BLAS) Level 1 copy vectors, swap the elements of two vectors, scale vector elements, perform reduction operations on vectors, and effect a Givens plane rotation. Scalar and vector versions of the BLAS Level 1 are provided in the new BLAS1$ and VBLAS1$ facilities, respectively. BLAS Level 1 forms an integral part of many standard libraries such as LINPACK and EISPACK. The version of the subroutines in the RTL VBLAS1$ facility have been tuned to the VAX architecture to take advantage of vectorization.
- The First Order Linear Recurrence (FOLR) routines provide a vectorized algorithm for the linear recurrence relation. (The traditional algorithm generally inhibits vectorization by using the result of a previous pass through a loop as an operand in subsequent passes through the loop.)

  The FOLR routines in the RTL MTH$ facility perform addition, multiplication, or both addition and multiplication on recursion elements, saving the result of each iteration in an array or saving only the last result in a variable. The RTL MTH$ facility supplies these routines in four groups, each accepting any of four data types: longword integer, F-floating, D-floating, or G-floating.
- Certain key MTH$ routines have been vectorized to support Digital's vectorizing compilers, such as the VAX FORTRAN High Performance Option (HPO). Vectorized versions of key F-floating, D-floating, and G-floating scalar routines employ vector hardware to the fullest,

while maintaining results that are identical to those of their scalar counterparts.

Vectorized MTH$ routines are never called directly from a high-level language program. At a call to a scalar version of one of these routines, a vectorizing compiler automatically determines whether an operation should be performed by the vector or scalar version of a routine. VAX MACRO programs, however, can call the vectorized MTH$ routines directly.

See the *VMS RTL Mathematics (MTH$) Manual* for complete information about these routines.

Note that the VAX FORTRAN HPO detects usage of the vectorizable constructs within source code and automatically issues a call to the appropriate RTL MTH$ routines. See the description of the /BLAS qualifier in the compiler documentation.

2.3.2 Obtaining Information About a Vector Processing System

The Get Job/Process Information system service (SYS$GETJPI) accepts the following item codes and returns the indicated information about the vector status of one or more processes in the system:

Item Code	Return Value
JPI$_FAST_VP_SWITCH	Unsigned longword containing the number of times this process has issued a vector instruction that resulted in an inactive vector processor being enabled without the expense of a vector context switch. This count reflects those instances in which the process has reenabled a vector processor on which the process's vector context has remained intact.
JPI$_SLOW_VP_SWITCH	Unsigned longword containing the number of times this process has issued a vector instruction that resulted in an inactive vector processor being enabled with a full vector context switch. This vector context switch involves the saving of the vector context of the process that last used the vector processor and the restoration of the vector context of the current process.
JPI$_VP_CONSUMER	Byte, the low-order bit of which, when set, indicates that the process is a vector consumer.
JPI$_VP_CPUTIM	Unsigned longword that contains the total amount of time the process has accumulated as a vector consumer.

The Get Systemwide Information system service (SYS$GETJPI) accepts the following item codes and returns the indicated information about the vector status of the system:

Item Code	Return Value
SYI$_VP_NUMBER	Unsigned longword containing the number of vector processors in the system.
SYI$_VP_MASK	Longword mask, the bits of which, when set, indicate which processors in the system have vector coprocessors.
SYI$_VECTOR_EMULATOR	Byte, the low-order bit of which, when set, indicates the presence of the VAX vector instruction emulator facility (VVIEF) in the system.

See Section 22.5 (of this manual) and the *VMS System Services Reference Manual* for additional information on the $GETJPI and $GETSYI system services.

2.3.3 Releasing the Vector Processor

The Release Vector Processor system service (SYS$RELEASE_VP) terminates the current process's status as a vector consumer. Because $RELEASE_VP declares that the process no longer needs the system's vector capability, VMS is no longer restricted to scheduling it on a vector-present processor. As a result, the process can be placed into execution on other CPUs in the system.

See Chapter 22 for a full description of the invocation format and functions of this service.

2.3.4 Preserving and Restoring a Routine's Vector State

The vector context of a process consists of the contents of the vector registers V0 through V15, the contents of the vector control registers (VLR, VCR, and VMR), the vector processor status, and the vector exception state. When a vectorized application involves calls among two or more routines, each of which issues vector instructions, two components of a process's vector context must be considered:

- The vector registers that are shared across procedure calls
- The vector exception state that exists just prior to a procedure call or return

The VAX Procedure Calling and Condition Handling Standard (see Section 2.3.7.1) requires that calling and called procedures agree as to the conventions by which they preserve and manipulate vector registers. For languages such as VAX MACRO, which allows direct access of registers, either the calling procedure or called procedure can save or restore vector registers shared between routines.

The standard also requires that, if a procedure executes a vector instruction that might possibly raise an exception, the procedure must ensure that this exception is reported before it calls another procedure, returns to its caller, or exits. If a vector exception were pending at the time a procedure transferred control, it would be reported in the context of a procedure that did not incur the exception. VAX vectorizing compilers ensure that compiled code properly follows this requirement; calls to vector routines in the RTL MTH$ facility (as described in Section 2.3.1) also comply with this prescription. However, vectorized code written in VAX MACRO must adhere to the rules discussed in Section 2.3.7.4.

For those routines that can run asynchronously with respect to the mainline routine—such as asynchronous system trap (AST) routines, condition handlers, and exit handlers—VMS automatically handles the saving and restoring of vector context. VMS supports vector usage in these asynchronous routines by providing each routine that is active asynchronously within a process with its own **vector state**.

The vector state of a routine reflects the vector context of the process at the time of the routine's execution or preemption, as the case may be, when an AST is delivered to the process or a condition handler is triggered. A process can have several vector states; for instance, one for its mainline routine and one for an AST routine that has interrupted the mainline. However, a process has only a single vector context, reflecting its current vector state.

VMS automatically preserves the vector state of a routine as follows:

- When a user mode AST routine issues a vector instruction, VMS saves the vector state of the mainline routine. It restores the mainline vector state when the AST routine exits.
- When a user mode condition handler issues a vector instruction, VMS saves the vector state of the mainline routine. It restores the mainline vector state on continuing from the exception and on stack unwind.
- When calling an exit handler, VMS clears the vector exception state.

By default, when an asynchronous routine interrupts the execution of a mainline routine, VMS creates a new vector state when the routine issues its first vector instruction. At this point, the vector state of the mainline routine is inaccessible to the asynchronous routine.

In certain cases, however, an AST routine or condition handler might need to read or modify the saved exception state of the mainline routine. To do so, the routine must call the Restore Vector State system service (SYS$RESTORE_VP_STATE). $RESTORE_VP_STATE restores the vector state of the process's mainline routine.

In very rare cases, a procedure might need to preserve and restore the current vector exception state across individual contexts that it creates and maintains. For instance, a task manager could set up several discrete tasks, each of which has its own vector state. To implement such a system, the routine saves the contents of the appropriate vector registers and calls the Save Vector Exception State (SYS$SAVE_VP_EXCEPTION) and Restore Vector Exception State (SYS$RESTORE_VP_EXCEPTION) system services.

The Save Vector Exception State service saves in memory any pending vector exception state and clears the vector processor's current exception state. The Restore Vector Exception State service restores from memory the vector state saved by a prior call to $SAVE_VP_EXCEPTION. After a routine invokes this service, the next vector instruction issued within the process causes the restored vector exception to be reported.

See Chapter 22 for a full description of the syntax and use of the $SAVE_VP_EXCEPTION, $RESTORE_VP_EXCEPTION, and $RESTORE_VP_STATE system services.

2.3.5 Debugging a Vectorized Program

The Version 5.4 of the VMS operating system supports the debugging of vector applications by adding new capabilities to the VMS Debugger, the VMS System Dump Analyzer (SDA), the debuggers of the VMS Delta /XDelta Utility (DELTA/XDELTA), and the Patch Utility. Additionally, the VMS exception detecting and reporting mechanism collects information regarding the nature and context of vector processing errors. Section 2.3.6 describes the information VMS provides when reporting a vector processing exception.

2.3.5.1 Vector Processing Support Within the VMS Debugger

Through enhancements and additions to its existing command set, the VMS Debugger allows you to correct and tune vectorized applications. VMS Debugger commands enable you to perform the following tasks:

- Control and monitor the execution of vector instructions with breakpoints, watchpoints, and other mechanisms
- Examine and deposit into the vector control registers (VCR, VLR, and VMR) and the vector registers (V0 through V15)
- Examine and deposit vector instructions
- Perform masked operations on vector registers to display only certain register elements or override the masking associated with a vector instruction
- Control synchronization between the scalar and vector processors
- Save and restore the current vector state when using the CALL command to execute a routine that might affect the vector state
- Display vector register data using a screen-mode display
- Display the decoded results of vector arithmetic exceptions

See Chapter 19 (of this manual) and the *VMS Debugger Manual* for complete information about these and other functions of the VMS Debugger.

2.3.5.2 Vector Processing Support Within the VMS System Dump Analyzer (SDA)

The System Dump Analyzer (SDA) provides several mechanisms for examining vector instructions and vector context from a system dump file or in a running system. They include the following:

- You can decode and display vector instructions using the EXAMINE /INSTRUCTION command. This command displays the vector opcodes, switches, and operands in the form and order defined by the VAX MACRO assembler notation. Note that, when you use SDA to display the contents of memory locations, vector instructions appear in the instruction stream format defined by the VAX architecture: that is, an opcode followed by the vector control word in immediate addressing mode. (See the *VAX MACRO and Instruction Set Reference Manual* for descriptions of the assembler notation and instruction stream formats of vector instructions.)

- You can examine the values of a process's vector registers and vector control registers by entering the SHOW PROCESS/VECTOR_REGISTERS command. This command obtains the values of the registers from the process's vector context area. Note that the names of these registers (V0 through V15, VCR, VLR, and VMR) are not defined in the SDA symbol table. You cannot display the current contents of any of these registers using the EXAMINE or EVALUATE command.
- You can format the contents of a memory location as a process's vector context block. The symbol table SYS$SYSTEM:SYSDEF.STB contains a definition of this structure. You must use the READ command to load the symbols defined within this file into the SDA symbol table.
- You can determine the presence and location of the VMS vector processing support code (VECTOR_PROCESSING.EXE) and the VAX Vector Instruction Emulation Facility (VVIEF) bootstrap code (VVIEF$BOOTSTRAP.EXE) by entering the SDA command SHOW EXECUTIVE. Both are executive loadable images. You can also use the SDA command READ/EXECUTIVE to load definitions of locations within these images into the SDA symbol table.

2.3.5.3 Vector Processing Support Within the VMS Delta/XDelta Utility

The VMS Delta/XDelta Utility (DELTA/XDELTA) provides mechanisms for stepping through vector code, examining and decoding vector instructions, and setting breakpoints at vector instructions. You can use the following commands to debug a vectorized application:

- The Open Location and Display Instruction in Instruction Mode command (!) displays the vector opcodes, switches, and operands in the form and order defined by the VAX MACRO assembler notation. Note that, when you use DELTA/XDELTA to display the contents of memory locations, vector instructions appear in the instruction stream format defined by the VAX architecture: that is, an opcode followed by the vector control word in immediate addressing mode. (See the *VAX MACRO and Instruction Set Reference Manual* for descriptions of the assembler notation and instruction stream formats of vector instructions.)
- The Step Instruction command (S) enables you to single step through vector instructions.
- The List Names and Locations of Loaded Executive Images command (;L) enables you to determine the presence and location of the VMS vector processing support code (VECTOR_PROCESSING.EXE) and the VAX Vector Instruction Emulation Facility (VVIEF) bootstrap code (VVIEF$BOOTSTRAP.EXE).

- The Breakpoint (;B) and Proceed from Breakpoint (;P) commands allow you to set and proceed from breakpoints at a vector instruction.

Note that, because the names of the vector registers (V0 through V15) and vector control registers (VCR, VLR, and VMR) are not defined in the DELTA/XDELTA symbol table, you cannot display their values using DELTA/XDELTA.

2.3.5.4 Vector Processing Support Within the VMS Patch Utility

Enhancements to the VMS Patch Utility allow it to interpret and display vector instructions that are replaced or deposited in a VAX MACRO program image file.

When issuing a REPLACE/INSTRUCTION instruction, you must supply the vector opcode, switches, and operands in the form and order defined by the VAX MACRO assembler notation. When displaying the contents of an image in instruction format, the Patch Utility produces vector instructions in this format. However, its hexadecimal listings present vector instructions in the instruction stream format defined by the VAX architecture: that is, an opcode followed by the vector control word in immediate addressing mode. (See the *VAX MACRO and Instruction Set Reference Manual* for descriptions of the assembler notation and instruction stream formats of vector instructions.)

2.3.6 Servicing Vector Exceptions

During the execution of an image, the image can incur a fatal error known as an exception condition. If the image has not declared a condition handler, the system forces the image to exit and displays a message indicating the reason for the exception. If the image has declared a condition handler, VMS transfers control to the handler to manage the exception. (See *Introduction to VMS System Services* for a description of how to write and declare a condition handler.)

There are two major classes of vector processing exceptions:

- Memory management exceptions, including access violations, vector alignment faults, and vector instruction references to I/O space
- Vector arithmetic exceptions

VMS reports exceptions in the first category (memory management exceptions) as forms of access violation, using the signals SS$_ACCVIO and SS$_VECALIGN (see Table 2–3). The exception argument list VMS supplies when signaling vector memory management exceptions is identical to the one it supplies with scalar access violations, except that VMS defines two additional bits in the reason mask to indicate the nature of the vector exception: a vector operation on an improperly-aligned vector element in memory (bit 3) and vector instruction reference to an I/O space address (bit 4).

VMS reports exceptions in the second category (vector arithmetic exceptions) using the signal SS$_VARITH (see Table 2–3). As defined by the VAX vector architecture (see the *VAX MACRO and Instruction Set Reference Manual*), vector operate instructions always execute to completion. If an exception occurs, the default result is written as follows:

- The low-order 32 bits of the true result for integer overflow.
- Zero for floating underflow if exceptions are disabled.
- An encoded reserved operand for floating divide by zero, floating overflow, reserved operand, and enabled floating underflow. For vector convert instructions that convert floating-point data to integer data, where the source element is a reserved operand, the value written to the destination element is UNPREDICTABLE.

Table 2–3 provides a summary of the means by which VMS signals vector processing exceptions and the arguments it provides for condition handlers. For information on how these exception conditions are reported by the VMS message facility, see Section 2.2.7.

Table 2–3 Summary of Exception Conditions

Exception	Type	Description	Arguments	
SS$_ACCVIO	Fault	Access violation	Two, as follows:	
			1 Reason for access violation. This is a mask with the following format:	
			Bit	**Description**
			0	Type of access violation: Clear if page table entry protection code did not permit intended access Set if P0LR, P1LR, or SLR length violation
			1	Page table entry reference: Clear if specified virtual address is not accessible Set if associated page table entry is not accessible
			2	Intended access: Clear if read Set if modify
			3	Vector alignment exception: Set if vector element is not properly aligned in memory[1]
			4	Vector instruction reference of I/O space Set if vector instruction referred to an I/O space address
			2 Virtual address to which access was attempted or, on some processors, virtual address within the page to which access was attempted. For access violations that occur due to a vector alignment exception or a vector instruction reference to I/O space, this virtual address is *always* an address within the page to which access was attempted.	
SS$_ILLVECOP	Fault	Illegal vector opcode.[2]	Four, as follows: 1 Signal name, SS$_ILLVECOP 2 Illegal opcode that caused the exception 3 Program counter (PC) of the vector instruction that caused the exception to be reported. (Note that this instruction is not always that that caused the exception.) 4 Processor status longword (PSL) at the time the exception is reported.	

[1] Note that the VMS operating system reports this exception with an SS$_VECALIGN fault.

[2] Note that some processors report illegal vector opcodes with the SS$_OPCDEC exception.

(continued on next page)

Table 2–3 (Cont.) Summary of Exception Conditions

Exception	Type	Description	Arguments
SS$_VARITH	Trap	Vector arithmetic trap	Five, as follows: 1 Signal name, SS$_VARITH. 2 Exception summary. This is a mask, the bits of which, when set, signify the following: **Bit** — **Meaning** 0 — Floating underflow 1 — Floating divide by zero 2 — Floating reserved operand 3 — Floating overflow 5 — Integer overflow 3 Vector register mask, the bits of which (0 through 15) correspond to the VAX vector registers (V0 through V15). When set, a bit indicates that an element of the associated vector register was involved in an operation that caused one or more of the vector arithmetic exceptions reported in the exception summary argument. 4 Program counter (PC) of the vector instruction that caused the exception to be reported. (Note that this instruction is not always the one that caused the *exception*.) 5 Processor status longword (PSL) at the time the exception is reported.
SS$_VECALIGN	Fault	Vector alignment exception	Identical to the argument list for SS$_ACCVIO

(continued on next page)

Table 2–3 (Cont.) Summary of Exception Conditions

Exception	Type	Description	Arguments
SS$_VECDIS	Fault	Vector processor disabled	Three, as follows: 1 Reason for vector disabled exception. The reason argument can have any of the following values: SS$_NOPRIV—An ACL on the vector capability has denied a user mode program access to the vector processor. SS$_MCHECK—The vector processor has been disabled due to the detection of a hardware error. SS$_INSFMEM—Insufficient nonpaged dynamic memory exists to turn the current process into a vector consumer. SS$_CPUNOTACT—The VAX system contains no vector-present processor on which to schedule the current process. SS$_BADCONTEXT—The vector state of the mainline routine is corrupt and cannot be restored. SS$_EXQUOTA—The VMS operating system cannot allocate sufficient space to save the vector state of the mainline routine because the process in which the routine is executing has exceeded process paging file quota. SS$_INSFWSL—The VMS operating system cannot allocate sufficient space to save the vector state of the mainline routine because the working set limit of the process in which the routine is executing is too low. SS$_VASFUL—The VMS operating system cannot allocate sufficient space to save the vector state of the mainline routine because the address space (P0 space) of the process in which the routine is executing is full. 2 Program counter (PC) of the vector instruction that caused the exception to be reported. (Note that this instruction is not always that that caused the exception.) 3 Processor status longword (PSL) at the time the exception is reported.

2.3.7 Requirements of the VAX Procedure Calling and Condition Handling Standard for Vector Processing

This section contains excerpts from the VAX Procedure Calling Standard that describe the requirements that procedures must follow when using the system's vector processing resources.

Code generated by VAX vectorizing compilers adheres to the rules described in this section. VAX MACRO code containing vector instructions must be written to comply with these requirements.

2.3.7.1 Vector Register Usage

The VAX Calling Standard specifies no conventions for preserved vector registers, vector argument registers, or vector function value return registers. All such conventions are by agreement between the calling and called procedures. In the absence of such an agreement, all vector registers, including V0 through V15, VLR, VCR, and VMR are scratch registers. Among cooperating procedures, a procedure that does preserve or otherwise manipulate the vector registers by agreement with its callers must provide an exception handler to restore them during an unwind.

2.3.7.2 Vector and Scalar Processor Synchronization

There are two kinds of synchronization between a scalar and vector processor pair: memory synchronization and exception synchronization.

2.3.7.3 Memory Synchronization

Every procedure is responsible for synchronization of memory operations with the calling procedure and with procedures it calls. If a procedure executes vector loads or stores, the following must occur:

- An MSYNC instruction (a form of the MFVP instruction) must be executed before the first vector load/store to synchronize with memory operations issued by the caller. While an MSYNC instruction might typically occur in the entry code sequence of a procedure, exact placement can also depend on a variety of optimization considerations.
- An MSYNC instruction must be executed after the last vector load/store to synchronize with memory operations issued after return. While an MSYNC instruction might typically occur in the return code sequence of a procedure, exact placement can also depend on a variety of optimization considerations.
- An MSYNC must be executed between each vector load/store and each standard call to other procedures to synchronize with memory operations issued by those procedures.

That is, any procedure that executes vector loads or stores is responsible for synchronizing with potentially conflicting memory operations in any other procedure. However, execution of an MSYNC to ensure scalar/vector memory synchronization can be omitted when it can be determined for the current procedure that all possibly incomplete vector load/stores operate only on memory that is not accessed by other procedures.

2.3.7.4 Exception Synchronization

Every procedure is responsible for ensuring that no exception can be raised after the current frame is changed (as a result of either a CALL or RET). If a procedure executes any vector instruction that might possibly raise an exception, then a SYNC instruction (a form of the MFVP instruction) must be executed prior to any subsequent CALL or RET.

However, if it can be determined that the only possible exceptions that can occur are ensured to be reported by an MSYNC instruction that is otherwise needed for memory synchronization, then the SYNC is redundant and can be omitted as an optimization.

Moreover, if it can be determined that the only possible exceptions that can occur are ensured to be reported by one or more MFVP instructions that read the vector control registers, then the SYNC is redundant and can be omitted as an optimization.

2.3.7.5 Synchronization Summary

Memory synchronization with the caller of a procedure that uses the vector processor is required because there might be scalar machine writes (to main memory) still pending at the time of entry to the called procedure. The various forms of write-cache strategies allowed by the VAX architecture combined with the possibly independent scalar and vector memory access paths imply that a scalar store followed by a CALL followed by a vector load is not safe without an intervening MSYNC.

Within a procedure that uses the vector processor, proper memory and exception synchronization might require use of an MSYNC instruction or a SYNC instruction, or both, prior to calling another procedure or upon being called by another procedure. Further, for calls to other procedures, the requirements may vary from call to call depending on details of actual vector usage.

An MSYNC instruction (without SYNC) at procedure entry, procedure exit, and prior to a call, should provide proper synchronization in most cases. A SYNC instruction (without an MSYNC prior to a CALL or RET) will sometimes be appropriate. The remaining two cases, where both or neither MSYNC and SYNC are needed, are probably rare.

Refer to the *VAX MACRO and Instruction Set Reference Manual* in the VAX Vector Architecture section for the specific rules on what exceptions are ensured to be reported by MSYNC and other MFVP instructions.

2.3.7.6 Condition Handler Parameters and Invocation

If the VAX vector hardware or emulator option is in use, then, for hardware detected exceptions, the vector state is implicitly saved before any condition handler is entered and restored after the condition handler returns. (No save/restore is required for exceptions that are initiated by calls to LIB$SIGNAL or LIB$STOP because there can be no useful vector state at the time of such calls in accordance with the rules for Vector Register Usage in Section 2.3.7.1.) A condition handler can thus make use of the system vector facilities in the same manner as mainline code.

The saved vector state is not directly available to a condition handler. A condition handler that needs to manipulate the vector state to carry out agreements with its callers can call the $RESTORE_VP_STATE service. This service restores the saved state to the vector registers (whether hardware or emulated) and cancels any subsequent restore. The vector state can then be manipulated directly in the normal manner by means of vector instructions. (This service is normally of interest only during processing for an unwind condition.)

2.3.8 VMS Accounting Utility (ACCOUNTING) Resource Packet Format

The VMS Accounting Utility uses the longword field ACR$L_VP_CPUTIME in the resource packet (ACR$K_RESOURCE) to record the vector CPU time (measured in 10-millisecond clock ticks) accrued by a process or image.

See the *VMS Accounting Utility Manual* for a complete description of the format and contents of ACCOUNTING records.

2.3.9 VMS Monitor Utility (MONITOR) VECTOR Class Record

As discussed in *VMS Monitor Utility Manual,* the VMS Monitor Utility (MONITOR) writes binary performance data to a VAX RMS sequential file known as the MONITOR recording file. Once per recording interval, MONITOR writes to this file a record containing data pertinent to each currently selected class. Version 5.4 of the VMS Monitor Utility includes the VECTOR class record, which contains data describing the time during which vector consumers have been scheduled on a vector-present processor.

See Section 16.3 for a complete description of the MONITOR VECTOR command and the VECTOR class. See Section 16.4 for specific information about the VECTOR class record and format.

3 Introduction to DECdtm Services

The VMS Version 5.4 operating system includes DECdtm services, which provide system services that demarcate and coordinate distributed transactions. By using the two-phase commit protocol, these services ensure consistent execution of distributed transaction on the VMS operating system. In turn, these system services make use of underlying logging and communication primitives necessary to enable distributed transaction commitment.

This chapter describes how the DECdtm services coordinate distributed transaction processing. The following sources in this manual also describe aspects of VMS Version 5.4 support for DECdtm services:

- Chapter 15 (Log Manager Control Program Utility (LMCP))
- Section 16.1 (MONITOR TRANSACTION Command and TRANSACTION class)
- Chapter 22 (new and modified system services)
- Chapter 29 (of this manual) and the *VAX RMS Journaling Manual* (RMS Journaling support)
- *VMS Version 5.4 Release Notes*

Note: **By default, processes for DECdtm services are started when a full VMS boot is executed. Before any transactions can be started, however, you must first use the Log Manager Control Program Utility (LMCP) to create a transaction log file (as described in Chapter 15).**

If you do *not* want to run DECdtm software, you can prevent the startup of DECdtm processes by defining the systemwide logical name SYS$DECDTM_INHIBIT in the SYS$MANAGER:SYLOGICALS.COM command procedure. You can define SYS$DECDTM_INHIBIT to be any string. For example:

```
$ DEFINE/SYSTEM/EXEC SYS$DECDTM_INHIBIT "yes"
```

See the *Guide to Setting Up a VMS System* for more information about the SYLOGICALS.COM command procedure.

3.1 Characteristics of Distributed Transactions

In business terminology, a transaction is a discrete unit of work. One example of a transaction is the purchasing of tickets from an airline reservation system. Another example is the transferring of funds between customer accounts using an automated teller machine (ATM). In both examples, the processing of the transaction involves interaction with databases.

Characteristically, transaction processing incorporates large, corporate-level applications that support many users for critical business functions. In transaction processing applications, there are usually many users simultaneously performing predefined functions (query and update) to a collection of shared data, generally a database. Results are usually expected immediately.

Another characteristic of transaction processing is that it is usually distributed. Transaction execution typically involves communication between a client program and one or more databases that can be locally or remotely located. This communication between client and server might typically take place through a network of systems distributed at various geographic locations; hence, the operation can be called distributed transaction processing. In the example of funds transfers at an ATM, the central system—or database—acts as a server, providing services to the customer—or client—at the ATM.

A single transaction represents the execution of a set of procedures. A client and the server must communicate using read and write operations to enable the client program to perform the desired task, for example, to perform a debit/credit operation to transfer funds in customer accounts.

Figure 3–1 shows the execution flow of a simple debit/credit application. A user at the ATM requests a financial operation, such as a transfer of funds from one account to another. A client program on Node A receives this request from the ATM. The client program forwards the request to a debit/credit program on Node B, and the debit/credit program updates the customer accounts database. The transaction shown in this figure is distributed because the cooperating programs are located on different computer systems.

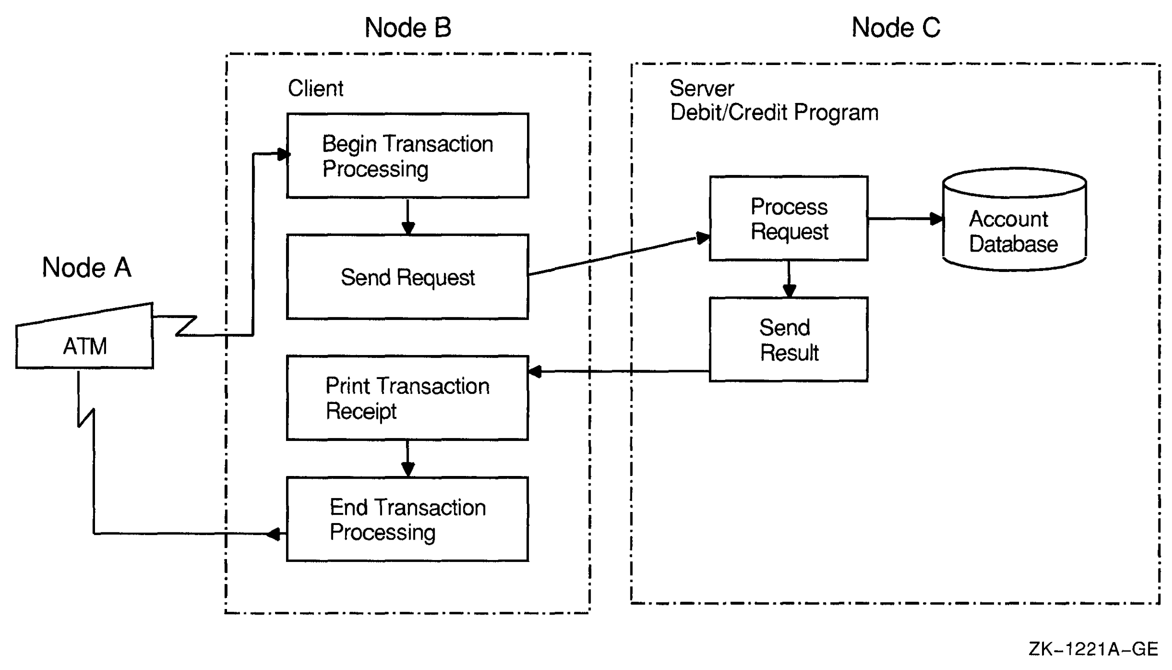
For transaction processing to be reliable, every required operation involved in the execution of the transaction must be completed before the transaction is made permanent; otherwise none of the operations are completed. A transaction that has this characteristic, known as atomicity, is considered an **atomic** transaction.

An atomic transaction must execute in its entirety or must have no effect at all. A transaction that executes in its entirety is called committed. One that terminates prematurely (and therefore has no effect) is called aborted.

The DECdtm services implement a commit protocol to guarantee atomic transaction processing. This protocol, known as the two-phase commit protocol, ensures atomicity by sequencing the commit process in such a way as to ensure that all resources (for example, databases) will be committed.

In the funds transfer example, it is vital that each of the customer's accounts is properly debited or credited and the account files updated only after it has been acknowledged that the transfer has occurred. If a system failure occurs while the transaction is processing, all of the previous operations of the transaction must be nullified. This arrangement keeps the database consistent; no operation is ever partially applied to the database.

Figure 3–1 Sample Debit/Credit Transaction Execution



3.2 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 define the task to be accomplished, such as query, update, and debit/credit. Application programs also specify 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 the three main transaction processing components:

- Resource managers
- Transaction managers
- Log managers

3.2.1 Resource Manager

A resource manager controls shared access to a set of recoverable resources on behalf of applications programs. A resource is usually a database. The term recoverable means that all updates to the resources on behalf of the transaction can be made permanent or can be undone.

A resource manager participates in the two-phase commit protocol to commit or abort a transaction.

Resource managers provide recovery mechanisms that work together with the DECdtm services and perform any necessary logging and recovery operations. The most common type of resource manager is a database system. Several Digital products can act as resource managers, including VAX RMS Journaling, Rdb/VMS, and VAX DBMS.

The execution of a transaction can span several nodes. The root application program can use the services of one or more resource managers on its home node. An application can also communicate with applications on other nodes, and these remote applications can also use other resource managers.

3.2.2 Transaction Manager

A transaction manager supports the services issued from application programs to start, end, and abort transactions. A transaction manager coordinates the action of a distributed transaction by sending instructions to resource managers about how to complete the transaction.

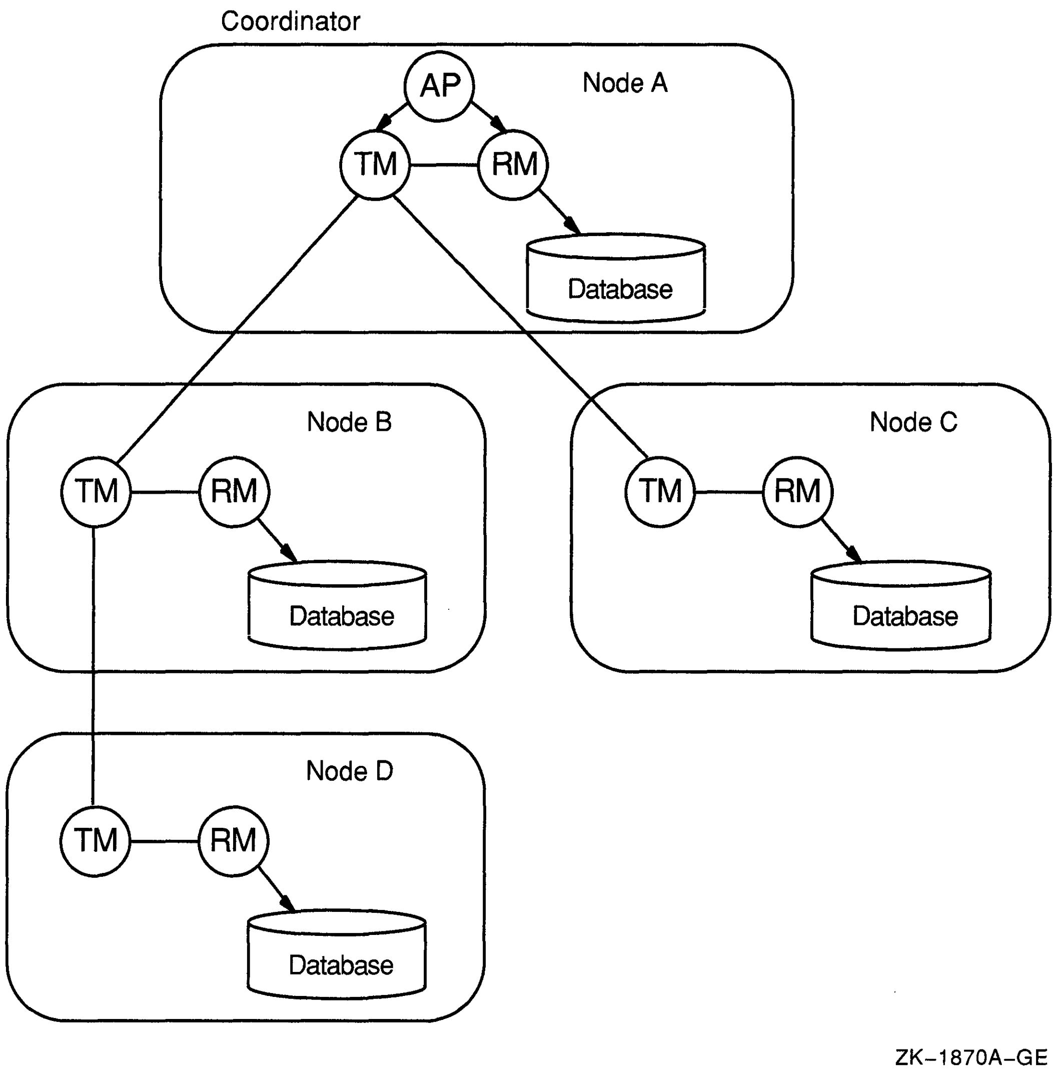
In a distributed network of transaction processing systems, each VMS node normally contains one DECdtm object. This object contains the transaction manager for transactions initiated from that node. The transaction manager maintains a list of participants in a transaction. In the execution of a transaction, participants may include:

- Resource managers on a local node, spanning one more or processes
- Transaction managers on other nodes within a network, which may also have associated resource manager and transaction manager participants

In this way, a hierarchy, or "tree," of resource managers and transaction managers can be established within the execution of a single transaction. The node on which a transaction is created is the "root" of the transaction. This is the coordinating or home node. Nodes containing the participating transaction managers and resource managers branch off from the root node. On each node, a transaction manager communicates only with its local resource managers, the transaction managers that are its immediate subordinates and the transaction manager that is its superior. A subordinate node is also referred to as a child node. A superior transaction manager is also referred to as a parent transaction manager.

In Figure 3–2, Node A is the coordinating node. It contains the parent transaction manager (TM) and the local resource manager (RM). The parent transaction manager coordinates the transaction started by the application program (AP) Node A with participating transaction managers and resource managers on other nodes. Nodes B, C, and D are all subordinates of Node A.

Figure 3–2 Participants in a Distributed Transaction Example



3.2.3 Log Manager

A log manager provides the mechanism for storing a permanent record of the execution of distributed transactions in log files. Each recoverable resource manager implements its own log manager component, which consists of a set of logging services. Logging services are also provided by the DECdtm services. During normal operation, resource managers and transaction managers write log files containing records of transaction state information. After recovering from a failure, a resource manager or transaction manager can read the log file to determine the state of a transaction at the time of failure.

3.3 Overview of Two-Phase Commit Protocol

Specific transaction management system services called from application programs mark the start and end of a transaction. The DECdtm system services include:

- 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)

The transaction manager component of the DECdtm services coordinates the execution of these system services. See Chapter 22 for more detailed descriptions of the DECdtm system services new for Version 5.4 of the VMS operating system.

The processing of a distributed transaction begins when an application calls the $START_TRANS or $START_TRANSW service. In response, the transaction manager 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.

Each resource manager is responsible for providing recovery capabilities for its own resources by performing transaction logging. The transaction manager is responsible for notifying all resource managers involved in a transaction of all relevant transaction state transitions. The transaction manager keeps track of the state of each transaction in case a system or process fails before the transaction completes.

The transaction manager maintains a list of resource managers and transaction managers that participate in a transaction's execution. 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.

For a complete description of transaction log files, see Chapter 15.

Each participating resource manager supports atomic transactions on its resources. To do this, the resource manager notifies the transaction manager as soon as that resource manager is first accessed by the application. A resource manager logs enough information to allow it to undo or redo operations it performed on behalf of a transaction. Similar to a transaction manager, a resource manager logs transaction state changes to a log file.

The processing of a transaction completes when one of the following calls is made:

- Commit—Using $END_TRANS or $END_TRANSW
- Planned abort—Using $ABORT_TRANS or $ABORT_TRANSW

(See Chapter 22 for more detailed descriptions of the DECdtm system services introduced in Version 5.4 of the VMS operating system.)

Upon receiving an End Transaction call, the DECdtm services implement the two-phase commit protocol to inform all participants how to proceed with the execution of the transaction.

The first phase of the two-phase commit protocol is the prepare phase. During this phase, the transaction manager uses a polling mechanism to determine if the participants can complete all the steps involved in a given transaction and can therefore commit the transaction. A participant that has successfully prepared casts a "yes" vote. If an error occurs during the polling that prevents a participant from responding—for example, if a resource manager fails or if a network link goes down—a "no" vote is assumed.

A "yes" vote indicates that the participating resource manager can either commit or abort the operations performed within this transaction, even if a failure occurs.

If all of the participants declare that they can commit by voting "yes," the transaction manager makes a decision to commit and proceeds to the second phase, known as the commit phase.

The transaction manager now orders the participants to commit the transaction. At this point all participants complete their transaction operations.

If any of the participants fails to prepare successfully, the transaction is aborted. The transaction manager orders all remaining participants to abort the transaction and roll back their transaction processing work. Thus, none of the actions of the distributed transaction are made permanent.

3.4 Managing DECdtm Services Using VMS Utilities

The VMS operating system provides the following utilities to manage the information provided by the DECdtm services:

- The Log Manager Control Program Utility (LMCP) is used to create and manage log files that are used by transaction managers. See Chapter 15 for a complete description.
- The VMS Monitor Utility can be used to monitor the status of transactions executing on the system. See Chapter 16 for more information.

3.5 New TRANSACTION_ID Data Type for Programming Routines

To support DECdtm programming routines, there is a new VMS data type, or structure, for low- and high-level languages. The **transaction_id** data type is an octaword that stores a unique transaction identifier.

Part 2: General User Features

This part contains the following chapters:

4 DCL Commands and Lexical Functions

This chapter includes the following information:

- Table 4–1 contains a summary of DCL commands that are new or enhanced in the VMS Version 5.4 operating system.
- Table 4–2 contains a summary of the lexical functions that are new or enhanced in the VMS Version 5.4 operating system.

Refer to the revised *VMS DCL Dictionary* for complete descriptions of all new and enhanced VMS Version 5.4 DCL commands and lexical functions.

Table 4–1 Summary of New and Enhanced DCL Commands

Command	Enhancements	
BACKUP	Now includes new /MEDIA_FORMAT qualifier, which controls data compaction on a TA90E tape drive.	
FONT	New command that compiles fonts for use by the DECwindows server and converts an ASCII bitmap distribution format (BDF) into binary server natural form (SNF).	
INITIALIZE	Now includes new /MEDIA_FORMAT qualifier, which controls data compaction on a TA90E tape drive.	
MOUNT	Now includes new /MEDIA_FORMAT qualifier, which controls data compaction on a TA90E tape drive.	
PSWRAP	New command that invokes the PSWRAP translator, which converts procedures written in PostScript to callable routines.	
SET ACL	Now includes the new CAPABILITY keyword for the /OBJECT_TYPE qualifier, which lets you specify a system capability such as the ability to process vector instructions.	
	Also includes the following new qualifiers:	
	/BACKUP	Modifies the time value from /SINCE or /BEFORE to select files according to their most recent BACKUP
	/EXPIRED	Modifies the time value from /SINCE or /BEFORE to select files according to their expiration date
	/MODIFIED	Modifies the time value from /SINCE or /BEFORE to select files according to their last modification date

(continued on next page)

Table 4–1 (Cont.) Summary of New and Enhanced DCL Commands

Command	Enhancements
SET HOST/DTE	Now includes new qualifiers and subcommands that provide a greater ability to control and customize the SET HOST/DTE operation. Specific enhancements include the following: • Using DTEPAD, you can now control and customize the configuration of a connection to a remote system through a terminal line. • New qualifiers let you select all configurational characteristics, such as XON /XOFF flow control, the maximum number of buffers, read, delay, and parity. • A new interactive command mode lets you configure the SET HOST/DTE session while the session is in progress. • The following new subcommands help you effectively control the SET HOST /DTE operation: CLEAR—Disconnects your local system from DTEPAD EXIT—Returns the session to DCL emulation mode QUIT—Disconnects your local system from DTEPAD SAVE—Saves the current configuration settings SEND BREAK—Sends a break to the remote system SET DTE—Modifies characteristics of DTEPAD SHOW DTE—Displays the configurable characteristics of DTEPAD SPAWN—Creates a subprocess of your local process See the *VMS DCL Dictionary* for complete information about the SET HOST/DTE command.
SET MAGTAPE	Now includes new /MEDIA_FORMAT qualifier, which controls data compaction on a TA90E tape drive.
SET SYMBOL	Now includes the following new qualifiers to control symbol scoping: /ALL, /GENERAL, and /VERB.
SET TERMINAL	Now includes a new value for the /DEC_CRT qualifier that sets characteristics for the VT400 family of terminals.
SHOW ACL	Now includes the new CAPABILITY keyword for the /OBJECT_TYPE qualifier, which lets you display a system capability such as the ability to process vector instructions.
SHOW CPU	New command that displays the current state of the processors in a VMS multiprocessing system.
SHOW ZONE	New command that displays the current state of a VAXft 3000 system.
START/CPU	New command that starts a secondary processor in a VMS multiprocessing system.
START/ZONE	New command that adds a zone to a running VAXft 3000 system.
STOP/CPU	New command that stops a secondary processor in a VMS multiprocessing system.
STOP/ZONE	New command that removes a zone from a running VAXft 3000 system.
VIEW	Now accepts new PS input format, which lets you use the CDA Viewer to view PostScript files (which use the file extension .PS). See Section 31.4.1 for additional information.

Table 4–2 Summary of New and Enhanced Lexical Functions

Lexical Function	Enhancements	
F$CSID	New function that returns the cluster identification numbers for nodes in a cluster.	
F$DEVICE	New function that returns the device names of devices on the system.	
F$ENVIRONMENT	Now includes the following new item codes:	
	DISIMAGE	Reports whether you are logged into an account that allows the RUN, MCR, or foreign commands.
	RESTRICTED	Reports whether you are logged into a restricted account.
	VERB_SCOPE	Reports the current verb scoping state.
F$GETDVI	Now includes the following new item codes:	
	Volume shadowing item codes	Provide the volume-shadowing status of a device.
	TT_DECCRT4	Reports whether a terminal is a Digital CRT4 terminal.
F$GETJPI	Includes the following new item codes:	
	Vector item codes	Provide information on the process's use of vector processing.
	Process rights item codes	Provide information about the process's rights.
F$GETSYI	Now includes the new parameter *cluster-id*, which specifies the cluster node for which information is to be returned.	
	Also includes the following new item codes:	
	ACTIVECPU_CNT	Returns the number of CPUs active in an SMP system.
	AVAILCPU_CNT	Returns the number of CPUs recognized by an SMP system.
	SYSTEM_RIGHTS	Contents of the system rights list.
	Vector item codes	New item codes provide information about the vector processors in the system.
F$TYPE	Now includes the following new return values:	
	PROCESS_CONTEXT	Indicates whether a symbol was created by the F$PID lexical function.
	CLUSTER_SYSTEM_ CONTEXT	Indicates whether a symbol was created by the F$CSID lexical function.

5 EVE Editor

This chapter describes the EVE Version 2.6 new features that are included in Version 5.4 of the VMS operating system. See the revised *VMS EVE Reference Manual* for more detailed information.

5.1 Box Editing

The new box editing feature lets you edit text using rectangular areas, or **boxes**, as well as standard, linear ranges. For example, you can select a box containing a list or columns in a table, and then cut and paste or perform some other editing operation on the box. Table 5–1 lists the new commands for box editing.

Table 5–1 EVE Box Editing Commands

Command	Usage
BOX COPY	Copies a box of text, without removing it, so you can paste it elsewhere.
BOX CUT	Cuts a box of text so you can paste it elsewhere, usually padding the area with spaces to keep the column alignment of text to the right of the box.
BOX CUT INSERT	Cuts a box, making text to the right of the box "collapse" to the left, closing the gap.
BOX CUT OVERSTRIKE	Cuts a box, padding the area with spaces to keep the column alignment of text to the right of the box.
BOX PASTE	Pastes a box of text you copied or cut, usually, overwriting existing text.
BOX PASTE INSERT	Pastes a box, pushing existing text to the right.
BOX PASTE OVERSTRIKE	Pastes a box, overwriting existing text.
BOX SELECT	Selects a box of text. Typically, you start at the upper left corner of the box and move the cursor to where you want the lower right corner.
RESTORE BOX SELECTION	Puts back (undeletes) a box erased with pending delete, usually overwriting existing text.
SET BOX NOPAD	Disables padding and overstriking for box editing unless the mode of the buffer is overstrike.
SET BOX NOSELECT	(Default.) Disables box selection, cutting, and pasting. Commands such as SELECT, COPY, REMOVE, and so on, use standard, linear ranges. To edit boxes, use BOX commands.

(continued on next page)

Table 5–1 (Cont.) EVE Box Editing Commands

Command	Usage
SET BOX PAD	(Default.) Enables automatic padding and overstriking for box editing, regardless of the mode of the buffer.
SET BOX SELECT	Enables box selection, making commands such as SELECT, REMOVE, and INSERT HERE the same as the corresponding BOX commands, without having to redefine keys.

5.2 New Command: CONVERT TABS

This command replaces tab characters with the appropriate number of spaces. This is useful if your file will be printed or displayed on devices with tab stops different from your settings in EVE.

5.3 New Qualifiers: /WORK and /INTERFACE

The /WORK qualifier (used with the EDIT/TPU command) determines the work file that is used to swap memory, allowing you to edit very large files.

The /INTERFACE qualifier, which you use with the EDIT/TPU command to specify either the character-cell or DECwindows interface, has been added for compatibility with other DECwindows applications. It is virtually the same as the /DISPLAY qualifier.

5.4 Additional Sources of New EVE Information

In addition to reading the revised *VMS EVE Reference Manual* to learn more about the new and changed features, you should also read the section about EDIT/TPU in the *VMS DCL Dictionary* (particularly the descriptions of the /JOURNAL and /RECOVER qualifiers) and review the following online help topics within EVE:

Attributes	Names For Keys
Defaults	New Features
Journal Files	Pending Delete
List Of Topics	Ranges And Boxes

6 System Messages

This chapter lists the VMS facilities that have new or modified system messages in Version 5.4 of the VMS operating system. There is also information about installing and accessing an online Help version of the *VMS System Messages and Recovery Procedures Reference Manual*.

6.1 VMS Facilities with New or Modified System Messages

The following VMS facilities have new or modified system messages in Version 5.4 of the VMS operating system. Refer to the *VMS System Messages and Recovery Procedures Reference Manual* for additional information.

- ANALDISK, Analyze/Disk_Structure Utility
- BACKUP, Backup Utility
- BUGCHECK, System Bugcheck
- CDA, Compound Document Architecture
- CLI, Command Language Interpreter (DCL)
- DISMOUNT, DISMOUNT Command
- FDL, Create/FDL Utility
- FDL, Edit/FDL Utility
- JBC, Job Controller
- LMCP, Log Manager Control Program
- MOUNT, Mount Utility
- MTH, Mathematics Facility
- NCP, Network Control Program
- PTD, Pseudoterminal
- REM, Set Host Facility
- RMS, VMS Record Management Services
- SDA, System Dump Analyzer
- SET, SET Facility
- SET PASSWORD Facility
- SYSBOOT, System Bootstrap Facility
- SYSGEN, System Generation Facility
- SYSTEM, VMS System Services
- UETP, User Environment Test Package

- VAXTPU, VAX Text Processing Utility
- Volume Processing Facility

6.2 System Messages Available from Online Help

With Version 5.4 of the VMS operating system, you can now install and access an optional online Help version of the *VMS System Messages and Recovery Procedures Reference Manual*. Because this is a large file, it is *not* included as part of the default root library, SYS$HELP:HELPLIB.HLB. You can access the file, named SYS$HELP:SYSMSGHELP.HLB, as follows:

- Use the /LIBRARY qualifier with the HELP command. For example:

```
$ HELP/LIBRARY=SYS$HELP:SYSMSGHELP.HLB ERRORS ACCVIO
```

- Define a logical name that instructs the Help Facility to search the new help library when it it does not find the specified topic in the VMS root help library. For example:

```
$ DEFINE HLP$LIBRARY DISK$2:[QUAIL]SYSMSGHELP
$ HELP ERRORS DISMAL
```

  In this example, the DEFINE statement creates a logical name for the help library that the Help Facility is to search after it has searched the root library, SYS$HELP:HELPLIB.HLB.

  The Help Facility first searches the root library for ERRORS. When it does not find error,[1] it then searches the library defined by HLP$LIBRARY until it finds ERRORS and displays the appropriate information. For information on defining logical names and search patterns for the Help facility, see the HELP COMMAND in the *VMS DCL Dictionary*.

- Using the VMS Librarian Utility, you can extract the ERRORS module from SYSMSGHELP.HLB and insert it into the default root help library HELPLIB.HLB. This allows direct access without using extra HELP qualifiers or logical names. For more information, see the *VMS Librarian Utility Manual*.

The system messages help library is in compressed format. Decompressing the library gives you faster access to it but requires an additional 1600 blocks of disk space. To decompress the library, enter a command similar to the following:

```
$ LIBRARY/DATA=EXPAND/OUTPUT=device:[directory]SYSMSGHELP.HLB -
_$ device:[directory]SYSMSGHELP.HLB
```

In this example, *device* is the name of the device where the file is located, and *directory* is the name of the directory.

Note: The system messages help library is not decompressed when you execute the LIBDECOMP.COM procedure described in the *VMS Version 5.4 Upgrade and Installation Manual*.

[1] Previous versions of HELPLIB.HLB provided information about system messages format under the name ERROR. This information is now named FORMAT_OF_ERROR.

You can use the VMS tailoring utility (VMSTAILOR) to add or delete the system messages help library. Deleting this library does not affect the other help libraries.

7 DECwindows User and Desktop Applications

This chapter describes new features of interest to DECwindows users. These features include enhancements to the Session Manager, the CDA Viewer, Calculator, Clock, and Mail.

7.1 Session Manager

Enhancements to the Session Manager include the addition of new languages to the Customize language dialog box and the ability to change your target screen, as described in Section 7.1.1 and Section 7.1.2 respectively.

7.1.1 Setting Another Session Language

The following languages have been added to the Customize Language dialog box in the Session Manager:

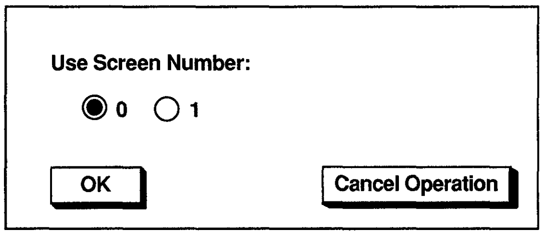
- Australian
- Austrian
- Belgian Dutch
- Belgian French
- Danish
- Fiji
- Finnish
- Hebrew
- New Zealand
- Papua New Guinea
- Portuguese

For more information about setting another session language, see the Version 5.3 edition of the *VMS DECwindows User's Guide*.

7.1.2 Changing Your Target Screen

When you run an application or choose Print Screen on a workstation that supports more than one screen display, by default DECwindows displays a dialog box asking you which screen you want to use (see Figure 7–1).

Figure 7–1 DECwindows Screen Number Dialog Box



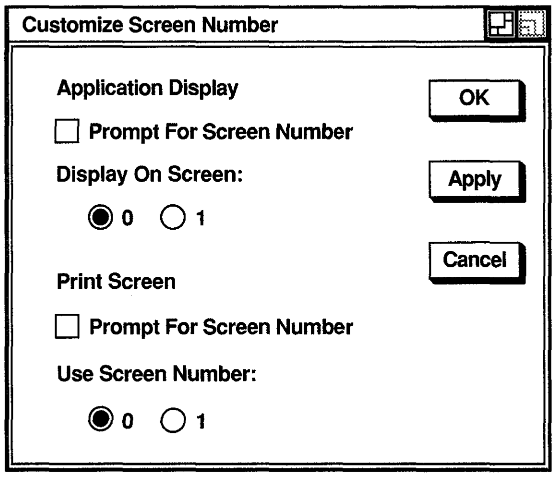
Use Screen Number:

(●) 0 ( ) 1

ZK-1959A-GE

If you want to use the same screen every time you run an application or use PrintScreen, you can disable the screen number prompt and choose your target screen. To disable the screen number prompt or change your target screen, choose Screen Number... from the Session Manager's Customize menu. The Session Manager displays the Customize Screen Number dialog box (see Figure 7–2).

Figure 7–2 DECwindows Screen Number Dialog Box



Customize Screen Number

Application Display

☐ Prompt For Screen Number

Display On Screen:

(●) 0 ( ) 1

Print Screen

☐ Prompt For Screen Number

Use Screen Number:

(●) 0 ( ) 1

ZK-1958A-GE

When you choose your target screen in the Customize Screen Number dialog box, DECwindows will run applications (or PrintScreen) on the screen you designated. If you click on the Prompt for Screen buttons, DECwindows will not display the screen number dialog box.

7.2 CDA Viewer

The DECwindows CDA Viewer now lets you view PostScript files. Section 7.2.1 describes how to view a PostScript file and Section 7.2.2 describes the new processing options available.

7.2.1 Viewing a PostScript File

To view a PostScript file, select the CDA Viewer menu item from the FileView Applications menu. In the Open window, click on PS in the File Format box and then select the PostScript file you want to view.

From a DCL window, enter the VIEW command in the following format to open a PostScript document for viewing:

```
VIEW filename.PS /FORMAT=PS /INTERFACE=DECWINDOWS
```

When you invoke the CDA Viewer from the DCL prompt, you do not need to specify processing options for the PostScript files.

PostScript file viewing is supported only in the DECwindows CDA Viewer and only when running to displays with servers containing the Display PostScript Extension. The CDA Viewer does not provide support for PostScript files on character cell terminals.

When viewing a PostScript file, after you select or turn to a particular page, you can click on the CDA Viewer Cancel button if you decide not to view the page while it is being processed. The CDA Viewer immediately stops processing that page.

7.2.2 New Processing Options for Viewing PostScript Files

In addition to the Default Paper Size option, new processing options specific to viewing PostScript files are available in the Paper Size dialog box. The additional PostScript options are highlighted, unless you already chose PS as the file format to display.

These options are valid only for viewing PostScript files and are ignored for all other file formats:

- Orientation radio box

  The Orientation radio box lets you select the orientation for displaying PostScript files. By default, the CDA Viewer displays files in the same portrait or landscape mode in which they were created. You can use the Orientation radio box to select different orientations to view files in reverse landscape mode or upside down.

- Scale Factor option

  The Scale Factor option lets you scale the page display size of your PostScript file. The number you select indicates whether the CDA Viewer will shrink or enlarge the page display. If the scale factor is less than 1.0, the page display will shrink. If the number is greater than 1.0, the page display will expand. You can specify a scale factor in the range of 0.1 to 4.0 times the size of the original page display. By default, a typical page display has a scale factor of 1.0.

- Use Comments toggle button

  The Use Comments option specifies that the CDA Viewer should interpret File structure comments that often appear in PostScript files. This enables the CDA Viewer to detect the location of page breaks in a PostScript file, for example.

The Use Comments option is enabled by default. This is indicated by the highlighted Use Comments toggle button.

You can disable the Use Comments option by clicking on it before opening your PostScript file. This is recommended in instances where the PostScript file contains comments that are not correct, causing the CDA Viewer to either display the PostScript file incorrectly or generate an error message. In most cases, disabling the Use Comments option and reopening the file corrects the problem.

- Use Bitmap Widths toggle button

  The Use Bitmap Widths option adjusts the display of your PostScript file for improved viewing on the screen. By default, a printed PostScript file has a finer resolution, or more dots per inch, than a PostScript file displayed on a screen. If you try to view the printed format of a PostScript file on line, the page layout will be the same, but the text may be dense and difficult to read.

  To clarify your PostScript file for on line viewing, you can specify the Use Bitmap Widths option so that the CDA Viewer will use spacing formulas designed for bitmaps (screen images) instead of those designed for print.

  The Use Bitmap Widths option is disabled by default. If you select the Use Bitmap Widths option, the next time you open a PostScript file, the CDA Viewer will use bitmap widths to display your file. Text characters will appear well spaced and easy to read. However, the file may look slightly different on screen than it would when printed. Columns may not be aligned precisely or a paragraph formatted for right justification may appear instead with a ragged right margin.

- Use Fake Trays toggle button

  The Use Fake Trays option lets you view a PostScript file that contains tray size directives. Tray size directives are instructions that tell the printer what paper tray size to use. These directives, however, are specific to certain printers (such as the LPS40) and are not part of the Display PostScript language.

  By default, the CDA Viewer ignores tray size directives if you try to display a PostScript file that contains them. To override that default behavior and view tray size directives in a PostScript file (to identify occurrences of nonstandard PostScript, for example), click on the Use Fake Trays option and reopen the file.

- Watch Progress toggle button

  The Watch Progress option lets you view a PostScript file while it is being processed for display in the CDA Viewer window. You can view a page as it is being processed, rather than waiting to view the entire page after it has been processed.

7.3 Calculator

Calculator now has two additional modes: hexadecimal and octal. When you first start the Calculator, it is in decimal mode. A new Mode menu contains Hexadecimal and Octal menu entries for changing modes. The keyboard display and functions change according to the mode.

7.4 Clock

Clock now has a menu bar with File, Customize, and Help menus for interacting with Clock. The menu bar provides an alternative to the previous method of pressing MB2 while pointing to the Clock display.

The only menu item under File is Quit. Choose Quit to exit from Clock.

The Customize menu lets you change the Clock display. The Customize Menu has three menu items. The menu items correspond to the Settings..., Save Settings, and Use System Settings previously available on a pop-up menu. Choosing the Settings... menu item displays the Clock Settings dialog box. The only change to the dialog box is the addition of a toggle button for Menu Bar. By default, the Menu Bar button is shaded and the menu bar is displayed. If you do not want the menu bar displayed, click on the Menu Bar button.

Help is now available directly as a menu on the menu bar, rather than from a pop-up dialog box.

7.5 Mail: Displaying PostScript Files

Mail can now display PostScript files, provided the files you send or receive contain *only* PostScript language. A PostScript file always begins with a percent sign and an exclamation point (%!). If any other text precedes the %!, Mail cannot display the file. For example, when mail is forwarded, additional text (in the form of extra mail headers) is often inserted at the beginning of the file. Because this additional text precedes the %!, Mail cannot display the PostScript file correctly. To avoid this problem, use an editor to remove all headers before you forward a mail message in PostScript format. Similarly, if you receive a PostScript file that does not display properly, use an editor to remove all headers (or any other text that precedes the %!), and forward the file to yourself. The file should then display properly.

Part 3: System Management Features

This part contains the following chapters:

8 AUTOGEN Command Procedure

This chapter describes changes to the AUTOGEN command procedure in Version 5.4 of the VMS operating system.

8.1 Parameter Name Validation

When AUTOGEN reads a parameter file such as MODPARAMS.DAT, it now checks to determine if the parameter names specified in the file are valid. If a parameter name is invalid, a warning message is written to AGEN$PARAMS.REPORT (a new file described further in Section 8.2). The following is an example of this warning message:

```
** WARNING ** - Invalid parameter name: LPRCOUNT
        The following record is suspect:
           LPRCOUNT = 34
```

AUTOGEN checks only the parameter name. It does not check the validity of the value specified for the parameter.

If a parameter name is invalid, the line is *not* ignored. AUTOGEN attempts to use the specified value.

A parameter name is not checked if it is specified in a line that contains a DCL expression other than the symbol assignment (=). For example, AUTOGEN does not check the validity of a parameter name specified in a line with a DCL IF statement. Instead, AUTOGEN writes a warning message to AGEN$PARAMS.REPORT. The following is an example of this message:

```
** WARNING ** - DCL command detected
        Parameter validation turned off for:
           IF WINDOW_SYSTEM = 1 THEN NPAGEDYN = 250000
```

8.2 AGEN$FEEDBACK.REPORT Replaced by New File

The file SYS$SYSTEM:AGEN$FEEDBACK.REPORT has been replaced by a new file called SYS$SYSTEM:AGEN$PARAMS.REPORT. This new file includes all of the information previously contained in AGEN$FEEDBACK.REPORT, as well as information about the non-feedback parameters and additional messages. Many of the warning and informational messages that AUTOGEN previously displayed on the screen are now written to AGEN$PARAMS.REPORT.

For example, when AUTOGEN finds multiple MIN_, MAX_, or ADD_ values for a single parameter, AUTOGEN writes a warning message to AGEN$PARAMS.REPORT. The warning message includes the parameter name, the value being used for the MIN_, MAX_, or ADD_ value, and the value being superseded. The following are examples of this type of message:

```
** WARNING ** - Multiple ADD records for ADD_LRPCOUNT found.
        VMS value (300) combining with MODPARAMS value (400)
        Value used is 700

** WARNING ** - Multiple MIN values found for MIN_LRPCOUNTV.
        Using VMS value (1000) which is superseding MODPARAMS value (800)

** WARNING ** - Multiple MAX values found for MAX_SWAPFILE2_SIZE.
        Using MODPARAMS value (1000) which is superseding VMS value (1200)
```

When AUTOGEN uses feedback information to calculate the value for a new parameter, this information is written to AGEN$PARAMS.REPORT. The following is an example of this type of message:

```
MAXPROCESSCNT parameter information:
        Feedback information.
           Old value was 41. New value is 50
           Maximum Observed Processes: 35
```

When an AUTOGEN calculation is overridden by a value specified in a parameter file, AUTOGEN writes a message to AGEN$PARAMS.REPORT. This message includes the new parameter value and the reason why the parameter was overridden. AUTOGEN will write this message for any parameter value that overrides AUTOGEN's calculations, whether the value is supplied by the system manager or by Digital. The following is an example of this type of message:

```
LONGWAIT parameter information:
        Override Information - parameter calculation has been overridden.
           The calculated value was 30.  The new value is 10.
           LONGWAIT has been disabled by a hard-coded value of 10.
```

8.3 MODPARAMS.DAT Includes External Parameter Files

To aid in cluster management, AUTOGEN can now read external parameter files specified within MODPARAMS.DAT. This feature allows system managers to maintain both clusterwide and system-specific versions of AUTOGEN parameters.

To include a parameter file, place the following command in MODPARAMS.DAT or in any subsequent parameter file:

```
AGEN$INCLUDE_PARAMS full-directory-specification:filename
```

Note: **If an include statement is the first line in MODPARAMS.DAT, AUTOGEN attempts to resolve all subsequent parameter settings. For example, if AUTOGEN finds two MIN_ statements for the same parameter, it uses the higher value. If the statements cannot be resolved, AUTOGEN uses the parameter setting specified after the include file.**

The following is an example of a MODPARAMS.DAT that includes an external parameter file:

```
! include system wide parameter settings
!
AGEN$INCLUDE_PARAMS SYS$COMMON:[SYSMGR]COMMON_CI_NODE_MODPARAMS.DAT

MIN_LRPCOUNT = 45
DUMPSTYLE = 0
.
.
.
```

This example reads the parameter file named SYS$COMMON:[SYSMGR]COMMON_CI_NODE_MODPARAMS.DAT before reading the parameters specified after the include statement in MODPARAMS.DAT. If the included file in this example specified the parameter setting DUMPSTYLE = 1, AUTOGEN would override this setting with the statement DUMPSTYLE = 0, which is specified after the include statement in MODPARAMS.DAT.

The format of all included parameter files should be the same as MODPARAMS.DAT. For information on MODPARAMS.DAT, see the description of AUTOGEN in *Guide to Setting Up a VMS System.*

8.4 MIN_, MAX_, and ADD_ Values Allowed for Page and Swap Files

You can now control the size of page and swap files by specifying MIN_, MAX_, and ADD_ values in a parameter file. The syntax for specifying MIN_, MAX_, and ADD_ values is identical to that used with other parameters.

For example, you can control the size of general page and swap files by including one or more of the following lines in a parameter file:

```
PAGEFILE = 20000
ADD_PAGEFILE = 5000
MIN_SWAPFILE = 1500
MAX_SWAPFILE = 4000
```

You can also specify the sizes of individual page and swap files (including secondary files) by including one or more of the following lines in a parameter file:

```
SWAPFILE1_SIZE = 2000
ADD_PAGEFILE1_SIZE = 2000
MIN_PAGEFILE2_SIZE = 3000
MAX_SWAPFILE3_SIZE = 3000
```

Note: You cannot specify a MIN_, MAX_, or ADD_ value for both a general page or swap file and a specific page or swap file.

8.5 New Feedback Parameters

The existing parameters LRPCOUNT and LNMSHASHTBL are now feedback parameters. This means that AUTOGEN can set these parameters using data collected in AUTOGEN feedback mode. You should remove any values for LRPCOUNT and LNMSHASHTBL that are specified in MODPARAMS.DAT, including MIN_, MAX_ and ADD_ values, so that AUTOGEN can set these parameters using feedback information.

8.6 Logical Names Defined by AUTOGEN

To aid in system management, AUTOGEN defines three process logical names to indicate how AUTOGEN was last run. These logical names are assigned a character string value each time AUTOGEN is run on a system. The following table lists and describes the logical names:

Logical Name	Description
AGEN$P1	The starting phase of AUTOGEN, for example, SAVPARAMS.
AGEN$P2	The end phase of AUTOGEN, for example, TESTFILES. If an error occurred which caused AUTOGEN to abort, then "_E" is appended to the phase name, for example, GENPARAMS_E.
AGEN$P3	The mode of execution, that is, either FEEDBACK or NOFEEDBACK.

8.7 New Technique for Running AUTOGEN in Batch Mode

As of Version 5.2-1 of the VMS operating system, Digital recommends a new technique for running AUTOGEN. This technique automates AUTOGEN feedback, allowing the system manager to receive reports from multiple systems on a regular basis. To use this technique, create a batch-oriented procedure which runs AUTOGEN in two stages. A sample command procedure is shown in Example 8–1.

The first stage of the command procedure runs AUTOGEN at peak times to collect data on realistic system loads. The following command accomplishes this task:

```
$ @SYS$UPDATE:AUTOGEN SAVPARAMS SAVPARAMS FEEDBACK
```

Executing this command does not affect the performance of the system.

The second stage of the command procedure runs AUTOGEN again during off-peak hours to interpret the data collected in the first stage. The following command accomplishes this task:

```
$ @SYS$UPDATE:AUTOGEN GETDATA TESTFILES FEEDBACK
```

The procedure sends the resulting report, contained in the file AGEN$PARAMS.REPORT, to the SYSTEM account using the following MAIL command:

```
$ MAIL/SUBJECT="AUTOGEN FEEDBACK REPORT FOR system-name" -
   SYS$SYSTEM:AGEN$PARAMS.REPORT SYSTEM
```

Review this report on a regular basis to see whether the load on a system has changed. If AUTOGEN's calculations are different from the current values, correct the tuning by executing AUTOGEN with one of two commands:

- If the system can be shut down and rebooted immediately, execute the following command:

  ```
  $  @SYS$UPDATE:AUTOGEN GETDATA REBOOT FEEDBACK
  ```

- If the system cannot be shut down and rebooted immediately, execute the following command to reset the system parameters:

  ```
  $  @SYS$UPDATE:AUTOGEN GETDATA SETPARAMS FEEDBACK
  ```

  The new parameters will take effect the next time the system boots.

The sample command procedure shown in Example 8–1 will run AUTOGEN in the new technique described. Use this procedure only as an example; create a similar command procedure as necessary to meet your requirements.

Example 8–1 Sample AUTOGEN Command Procedure

```
$ BEGIN$:   ! ++++++++++ AGEN_BATCH.COM ++++++++++
$  on warning then goto error$
$  on error then goto error$
$  on severe_error then goto error$
$  on control_y then goto error$
$!
$! Setup process
$!
$! Set process information
$  set process/priv=all/name="AUTOGEN Batch"
$! Keep log files to a reasonable amount
$  purge/keep=5 AGEN_Batch.log
$  time = f$time()     ! Fetch current time
$  hour = f$integer(f$cvtime(time,,"hour")) ! Get hour
$  today = f$cvtime(time,,"WEEKDAY")  ! Get Day of the week
$  if f$integer(f$cvtime(time,,"minute")) .ge. 30 then hour = hour + 1
$!
$! Start of working day...
$!
$ 1AM$:
$  if hour .le. 2
$     then
$     next_time = "today+0-14"
$     gosub submit$    ! Resubmit yourself
$     set noon
```

(continued on next page)

Example 8–1 (Cont.) Sample AUTOGEN Command Procedure

```
$!
$!      Run AUTOGEN to setparams using the parameter values collected earlier
$!      in the day (i.e., yesterday at 2:00pm)
$     if today .eqs. "Tuesday" .OR. today .eqs. "Thursday" .OR. -
  today .eqs. "Saturday"
$        then
$    @sys$update:autogen getdata testfiles feedback
$    mail/sub="Autogen Feedback Report for system-name" -
  sys$system:agen$params.report system
$! Clean up
$          purge/keep=7 sys$system:agen$feedback.report
$          purge/keep=7 sys$system:agen$feedback.dat
$          purge/keep=7 sys$system:params.dat
$          purge/keep=7 sys$system:autogen.par
$          purge/keep=7 sys$system:setparams.dat
$          purge/keep=7 sys$system:agen$addhistory.tmp
$          purge/keep=7 sys$system:agen$addhistory.dat
$        endif
$     goto end$
$     endif
$!
$ 2PM$:
$  if hour .le. 15
$     then
$     next_time = "today+0-17"
$     gosub submit$
$     if today .eqs. "Monday" .OR. today .eqs. "Wednesday" .OR. -
  today .eqs. "Friday"
$        then
$          @sys$update:autogen savparams savparams feedback
$        endif
$     goto end$
$     endif
$!
$ 5PM$:
$  if hour .le. 18
$     then
$     next_time = "tomorrow+0-1"
$     gosub submit$
$     endif
$!
$! End of working day...
$!
$ END$:     ! ---------- BATCH.COM ----------
$  exit
$!++
$! Subroutines
$!--
```

(continued on next page)

Example 8–1 (Cont.) Sample AUTOGEN Command Procedure

```
$!
$ SUBMIT$:
$  submit/name="AGEN_Batch"/restart/noprint -
   /log=AGEN_batch.log -
   /queue=sys$batch/after="''next_time'" sys$system:AGEN_batch.com
$  return
$!++
$! Error handler
$!--
$ ERROR$:
$  mail/sub="AGEN_BATCH.COM - Procedure failed." _nl: system
$  goto end$
```

8.8 Using MAIL to Send AGEN$PARAMS.REPORT

After closing the AGEN$PARAMS.REPORT file, AUTOGEN now checks for the existence of a file named SYS$UPDATE:AGEN$MAIL.COM. If this file exists, it is executed from within AUTOGEN. (Note, however, that AUTOGEN does not execute AGEN$MAIL.COM during VMS upgrades or installations or after minimum system boots.)

You can use AGEN$MAIL.COM alone or with the batch-oriented procedure described in Section 8.7 to send AGEN$PARAMS.REPORT to the SYSTEM account or to an account of your choice. To do so, create a command procedure named SYS$UPDATE:AGEN$MAIL.COM that includes the following command:

```
$ MAIL/SUBJECT="AUTOGEN FEEDBACK REPORT FOR system-name" -
   SYS$SYSTEM:AGEN$PARAMS.REPORT SYSTEM
```

If you use the AGEN$MAIL.COM procedure along with the batch-oriented procedure described in Section 8.7, AGEN$MAIL.COM replaces the MAIL command line in the batch-oriented command procedure.

9 User Environment Test Package (UETP)

This chapter describes enhancements to the User Environment Test Package (UETP) that are new for Version 5.4 of the VMS operating system. For additional information about UETP, see the *VMS Version 5.4 Upgrade and Installation Manual.*

9.1 RRD40 Compact Disc Drive Support

With Version 5.4 of the VMS operating system, the User Environment Test Package (UETP) now supports the following:

- The RRD40 compact disc drive, including multiple RRD40 units
- SCSI disk configurations that allow both read-only compact discs and standard read/write disks to use the same device controller name

9.2 Vector Processing Support

With Version 5.4 of the VMS operating system, the User Environment Test Package (UETP) now automatically loads and tests all installed and enabled vector processors. It also lets you test the VAX Vector Instruction Emulation Facility (VVIEF).

The vector processor device test, UETVECTOR.EXE, performs simple vector-scalar and vector-vector arithmetic operations and compares the results with expected values. The test also uses vector-related system service extensions and forces the system to generate arithmetic and memory management exceptions.

For information on using UETP to test vector processors and VVIEF, see the *VMS Version 5.4 Upgrade and Installation Manual.* For complete information about vector processing support, see Chapter 2.

10 SYSMAN Utility

This chapter briefly describes enhancements to the VMS System Management Utility (SYSMAN) that are new for Version 5.4 of the VMS operating system. For complete information about these new features, see the revised *VMS SYSMAN Utility Manual.*

10.1 Running a SYSMAN Command Procedure

The SYSMAN command @ now executes the specified SYSMAN command procedure on each node in the environment.

10.2 Defining Keys with the DEFINE command

SYSMAN lets you define keys to execute SYSMAN commands. By defining keys, you can avoid typing lengthy SYSMAN commands. You can also put your key definitions in a SYSMAN initialization file, which executes each time you invoke SYSMAN.

10.3 Spawning a Subprocess from Within SYSMAN

If you are in SYSMAN but want to leave temporarily to perform other functions (such as displaying a directory listing or printing a file) and then return to SYSMAN, you can use the SPAWN and ATTACH commands. These commands function in much the same way as the DCL commands SPAWN and ATTACH.

10.4 Using DCL Verification

Using the SYSMAN command SET PROFILE/VERIFY, you can set DCL verification (both procedure and image) for future DO commands. This lets you view the lines of DCL command procedures as they execute.

10.5 Using Loadable Image Commands

Caution: SYS_LOADABLE commands are not intended for general use. Only advanced system programmers should use these commands.

The SYS_LOADABLE command set adds and removes executive loaded images from the list of images that are loaded by the SYSINIT process during the bootstrap. This new feature is intended primarily for system programmers to use when writing routines that implement site-specific policies or special algorithms. These routines can either replace or augment the built-in VMS policies.

In addition to consulting the *VMS SYSMAN Utility Manual* for a more detailed description of the SYS_LOADABLE command set, see Chapter 14 and Section 22.6 in this manual for related information about implementing site-specific policies.

11 VAXcluster Management

This chapter describes enhancements to the following VAXcluster components:

- Computer interconnect (CI) architecture extensions
- Mass Storage Control Protocol (MSCP) server load sharing
- Preferred path support for DIGITAL Storage Architecture (DSA) disks

See the revised *VMS VAXcluster Manual* for more information.

11.1 CI Architecture Extensions

Extensions to the computer interconnect (CI) architecture allow the application of multiple CI interfaces per CPU and multiple star couplers per VAXcluster system. These extensions make possible VAXcluster systems with many times the data throughput capacity of current VAXcluster systems with a single star coupler.

11.2 MSCP Server Load Sharing

Beginning with Version 5.4 of the VMS operating system, Mass Storage Control Protocol (MSCP) servers monitor their I/O traffic and periodically calculate a Load Available rating to indicate available capacity for I/O requests.

Load Available is calculated by counting the read and write requests sent to the server and periodically converting this to requests per second and subtracting this calculated value from the server's Load Capacity (also specified in requests per second).

This information is communicated to the VMS Version 5.4 MSCP class driver (DUDRIVER and DSDRIVER). When a disk is mounted or a failover occurs, the class driver selects the server with the highest Load Available rating to access the disk.

Load Balancing is enabled and controlled by the SYSGEN parameters MSCP_LOAD and MSCP_SERVE_ALL. In most cases, the values established by CLUSTER_CONFIG.COM are appropriate.

MSCP_SERVE_ALL determines whether the server participates in load balancing. If it is set to 2 (serve only local disks), the server does not monitor its I/O traffic and does not participate in load balancing. Other valid settings for MSCP_SERVE_ALL (0, 1) result in the server monitoring I/O traffic and communicating Load Available information to the class drivers.

MSCP_LOAD is used to communicate Load Capacity to the server, in addition to its existing function of controlling the loading of the MSCP server. If it is set to 1, the MSCP server is loaded and its Load Capacity is set to a default value based upon CPU type. If MSCP_LOAD is set to a value greater than one, the server is loaded and its Load Capacity set to that value.

As before, setting MSCP_LOAD to zero disables loading of the MSCP server.

11.3 Preferred Path Support for DSA disks

The VMS Version 5.4 operating system lets you specify a preferred path for DIGITAL Storage Architecture (DSA) disks. This includes RA series disks and disks accessed through the MSCP server.

If a preferred path is specified for a disk, the MSCP disk class drivers (DUDRIVER and DSDRIVER) uses the path as their first attempt to locate the disk and bring it online as a result of a DCL MOUNT command or failover of an already mounted disk.

In addition, it is possible to initiate failover of a mounted disk to force the disk to the preferred path or to use load balancing information for disks accessed via MSCP servers.

The preferred path is specified by a $QIO function IO$_SETPRFPTH, with the P1 parameter containing the address of a counted ASCII string (.ASCIC). This string is the node name of the HSC or VMS system that is to be the preferred path. The node name must match an existing node known to the local node and, if it is a VMS system, it must be running the MSCP server. This function does not move the disk to the preferred path. For more information on the IO$_SETPRFPTH function, refer to the *VMS I/O User's Reference Manual: Part I.*

12 System Generation Utility (SYSGEN)

This chapter describes enhancements to the VMS System Generation Utility (SYSGEN) that are new for Version 5.4 of the VMS operating system.

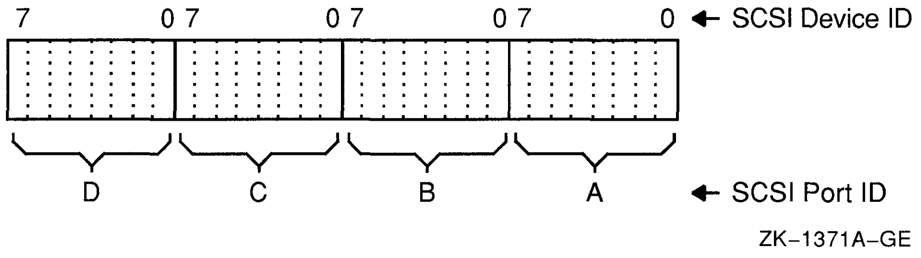
12.1 SCSI_NOAUTO Parameter

The VMS Version 5.4 operating system defines the special SYSGEN parameter SCSI_NOAUTO for use with MicroVAX or VAXstation configurations that include third-party Small Computer System Interface (SCSI) devices. (See *VMS Device Support Manual* for more about SCSI devices.) The SYSGEN parameter SCSI_NOAUTO replaces the SYSGEN parameter VMSD1.

SYSGEN's autoconfiguration facility automatically loads the VMS SCSI disk or tape class driver for a device on the SCSI bus that identifies itself as either a random-access or sequential-access device. If this SCSI device is to be supported instead by the VMS generic SCSI class driver or a third-party SCSI class driver, the automatic loading of a VMS SCSI class driver for the device must be disabled.

The SCSI_NOAUTO parameter, as shown in Figure 12–1, allows a configuration including a SCSI third-party device to prevent the loading of a VMS disk or tape SCSI class driver for any given device ID.

Figure 12–1 SCSI_NOAUTO System Parameter



The SCSI_NOAUTO system parameter stores a bit mask of 32 bits, where the low-order byte corresponds to the first SCSI bus (PKA0), the second byte corresponds to the second SCSI bus (PKB0), and so on. For each SCSI bus, setting the low-order bit inhibits automatic configuration of the device with SCSI device ID 0; setting the second low-order bit inhibits automatic configuration of the device with SCSI device ID 1, and so forth. For instance, the value 00002000_{16} would prevent the device with SCSI ID 5 on the bus identified by SCSI port ID *B* from being configured. By default, all of the bits in the mask are cleared, allowing all devices to be configured.

12.2 LOAD_PWD_POLICY Parameter

The SYSGEN parameter LOAD_PWD_POLICY works in conjunction with the SET PASSWORD Utility and with LOGINOUT (if you are forced to change your password at login). This parameter controls whether or not the SET PASSWORD Utility or LOGINOUT attempts to use site-specific password policy routines, which are contained in the shareable image SYS$LIBRARY:VMS$PASSWORD_POLICY.EXE. The default is 0.

Installing and enabling a site-specific password policy image requires both SYSPRV and CMKRNL privileges. To set the LOAD_PWD_POLICY parameter, enter the following commands:

```
$   RUN SYS$SYSTEM:SYSGEN
SYSGEN>   USE ACTIVE
SYSGEN>   SET LOAD_PWD_POLICY 1
SYSGEN>   WRITE ACTIVE
SYSGEN>   WRITE CURRENT
```

To make the changes permanent, modify the system parameter file, MODPARAMS.DAT so the parameter LOAD_PWD_POLICY is set to 1.

For descriptions of site-defined password filters for the VMS Version 5.4 operating system, see Chapter 14 and Section 22.6.

12.3 LOAD_SYS_IMAGES Parameter

The LOAD_SYS_IMAGES parameter controls the loading of system images described in the system image data file, VMS$SYSTEM_IMAGES.DATA. 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

Section 22.6 describes how to create a system service image and how to copy the image into the SYS$LOADABLE_IMAGES directory and add an entry for it in the VMS system images file using the SYSMAN Utility. After generating a new system image data file, you reboot the system to load in your service.

If you have difficulty booting with the site-specific system services and therefore do not want the site-specific system services loaded, you can set the parameter of LOAD_SYS_IMAGES to 0 during SYSBOOT. The default is 1.

12.4 Supported Device Names for VAXft 3000 Systems

With Version 5.4 of the VMS operating system, the System Generation Utility (SYSGEN) supports the following device types in VAXft 3000 systems:

Code Name	Device Type
CM	Environmental control monitor
GD	DMA driver
EF	Logical Ethernet driver
EP	Physical Ethernet driver
PW	DSSI disk driver
SF	Logical DSF driver
SM	Physical DSF driver

12.5 New SYSGEN Commands

This section describes the following new SYSGEN commands:

- SHOW/BI=BIindex
- SHOW/BUS=busId
- SHOW/XMI=BIindex

SHOW/BI=BIindex

The SHOW/BI=BIindex command displays device addresses that are currently mapped in the I/O space for the VAXBI bus. It also displays node and nexus numbers and generic names of UNIBUS and MASSBUS adapters, VAXBI adapters, memory controllers, and interconnection devices such as the DR32 and CI.

Use of the SHOW/BI=BIindex command requires the CMEXEC privilege.

FORMAT

SHOW/BI=*BIindex*

EXAMPLE

```
SYSGEN>   SHOW/BI




(CPU Type:  VAX 8800                     Cpu Connection: NMI

      ** Bus map for BI 00 on 28-FEB-1990 14:13:02.95 **
 Address 20000000 (node 00) responds with value 0108 CI
 Address 20004000 (node 02) responds with value 0106 BI - NMI Adapter (NBIB)
 Address 2000E000 (node 07) responds with value 0109 BI Combo Board (DMB32)
      ** Bus map for BI 01 on 28-FEB-1990 14:13:03.00 **
 Address 22000000 (node 00) responds with value 0102 UB
 Address 22004000 (node 02) responds with value 0106 BI - NMI Adapter (NBIB)
 Address 2200E000 (node 07) responds with value 410F BI - NI Adapter (DEBNA))
```

The command in this example displays device addresses that are currently mapped in the I/O space for the BI bus and additional information about the BI bus adapters.

SHOW/BUS=busId

The SHOW/BUS=busId command displays the buses and any subsequent attached buses and all attached device node numbers, generic names of processors, memory modules, adapters, VAXBI adapters, memory controllers, and interconnection devices such as the NI.

Use of the SHOW/BUS command requires the CMEXEC privilege.

FORMAT

SHOW/BUS=*busId*

EXAMPLE

```
SYSGEN>  SHOW/BUS
```

```
Cpu Type: VAX 8800                            Cpu Connection: NMI
Bus      Node    Generic Name                 Nexus(hex)  Connection Address
BI 00     00     CI                             0000
BI 00     02     BI - NMI Adapter (NBIB)        0002
BI 0      07     BI Combo Board (DMB32)         0007
BI 01     00     UB                             0010
BI 01     02     BI - NMI Adapter (NBIB)        0012
BI 01     07     BI - NI Adapter (DEBNA)        0017
```

The command in this example displays information about all the adapters on the system buses.

SHOW/XMI=BIindex

The SHOW/XMI=BIindex command displays device addresses that are currently mapped in the I/O space for the XMI bus. It also displays node and nexus numbers and generic names of processors, adapters, VAXBI adapters, memory controllers, and interconnection devices such as the NI.

Use of the SHOW/XMI=BIindex command requires the CMEXEC privilege.

FORMAT

SHOW/XMI=*BIindex*

EXAMPLE

```
SYSGEN>   SHOW/XMI



      ** Bus map for XMI 00 on 28-FEB-1990 14:14:50.48 **
 Address 21880000 (node 01) responds with value 8082 XMI - 6000-400 processor
 Address 21900000 (node 02) responds with value 8082 XMI - 6000-400 processor
 Address 21980000 (node 03) responds with value 8082 XMI - 6000-400 processor
 Address 21A00000 (node 04) responds with value 8082 XMI - 6000-400 processor
 Address 21A80000 (node 05) responds with value 8082 XMI - 6000-400 processor
 Address 21B00000 (node 06) responds with value 4001 XMI - memory module
 Address 21B80000 (node 07) responds with value 4001 XMI - memory module
 Address 21C00000 (node 08) responds with value 4001 XMI - memory module
 Address 21C80000 (node 09) responds with value 4001 XMI - memory module
 Address 21D00000 (node 0A) responds with value 4001 XMI - memory module
 Address 21D80000 (node 0B) responds with value 4001 XMI - memory module
 Address 21E00000 (node 0C) responds with value 0C03 XMI - NI adapter (DEMNA)
 Address 21E80000 (node 0D) responds with value 2001 XMI - BI Adapter (DWMBA/A)
 Address 21F00000 (node 0E) responds with value 2001 XMI - BI Adapter (DWMBA/A)
```

The command in this example displays device addresses that are currently mapped in the I/O space for the XMI bus and additional information about the XMI bus adapters.

13 Error Log Utility (ERROR LOG)

This chapter describes enhancements to the VMS Error Log Utility (ERROR LOG) that are new for Version 5.4 of the VMS operating system.

13.1 Supported Device Types for VAXft 3000 Systems

With Version 5.4 of the VMS operating system, the Error Log Utility supports the following device types in VAXft 3000 systems:

Code Name	Device Type
CM	Environmental control monitor
DSF32	Synchronous communications adapter
GD	DMA driver
EF	Logical Ethernet driver
EP	Physical Ethernet driver
PW	DSSI disk driver
RF31	DSSI fixed hard disk
SF	Logical DSF driver
SM	Physical DSF driver
TF70	DSSI magnetic tape drive

13.2 New Keywords for /EXCLUDE and /INCLUDE Qualifiers

The /EXCLUDE and /INCLUDE qualifiers accept new device-class and entry-type keywords, described in the following table:

Device-Class Keyword	Function
ADAPTER	Includes or excludes entries for adapter errors
CACHE	Includes or excludes entries for memory caching errors
INFORMATIONAL	Includes or excludes error log entries such as media quality reports from magnetic tape devices
VECTOR	Includes or excludes entries for vector processing errors

Entry-Type Keyword	Function
CONFIGURATION	Includes or excludes entries that describe system configuration

Entry-Type Keyword	Function
SYNDROME	Includes or excludes VAX 9000 console-generated entries that provide encoded syndrome values used by Customer Services

13.3 New Qualifier: /NODE

The Error Log Utility now accepts the /NODE qualifier. This qualifier enables you to generate a report consisting of error log entries for specific nodes in a VAXcluster.

FORMAT

/NODE *=(node-name[,...])*

Parameter

node-name
Specifies the names of one or more VAXcluster members. Names cannot exceed six characters. If more than one node name is entered, you must specify a comma-separated list of node names enclosed in parentheses.

Example

```
$ ANALYZE/ERROR_LOG/NODE=(ORANGE,NASSAU) ERRLOG.OLD;72
```

In this example, a VAXcluster includes members ORANGE, PUTNAM, and NASSAU. The output consists of only those entries that were logged for VAXcluster members ORANGE and NASSAU.

14 System Security

This chapter describes new features of the VMS Version 5.4 operating system that system managers can use to enhance the security of their systems. The following sources contain related information as well:

- Chapter 4 (of this manual) and the *VMS DCL Dictionary* (SET ACL and SHOW ACL)
- Section 10.5 (using loadable image commands within SYSMAN)
- Section 12.2 and Section 12.3 (relevant SYSGEN parameters)
- Chapter 22 (new and modified system services; implementing site-specific security policies)

14.1 Site-Defined Password Policy

Starting with the VMS Version 5.4 operating system, passwords selected by users can be screened for acceptability. The VMS system automatically compares new passwords against a system dictionary to ensure that a password is not a native language word. It also maintains a history list of a user's passwords and compares each new password against this list to guarantee that an old password is not reused. Sites can screen passwords further by developing and installing an image that filters passwords for words that are particularly sensitive to the installation.

In addition, a site with contractual obligations to use special algorithms for encrypting passwords will be able to use them.

This chapter describes these security enhancements. For descriptions of security-related system services new for the VMS Version 5.4 operating system, see Chapter 22.

14.1.1 Screening New Passwords

Sites that choose to let users select their own passwords rather than use the password generator can now screen user-selected passwords. As of Version 5.4, the VMS system automatically compares new passwords against a system dictionary, which is stored in SYS$LIBRARY, to ensure that a password is not a native language word. The VMS system also maintains a list of all the passwords a user has had during the year and compares each new password against this history list to guarantee that an old password is not reused.

Both the dictionary and the history search can be disabled through the Authorize Utility. You disable the dictionary search with the DISPWDDIC option to the /FLAGS qualifier; you disable the history search with the DISPWDHIS option to the /FLAGS qualifier.

14.1.1.1 Password History List

VMS keeps a year's worth of data in the password history list. If the password limit is exceeded, the system forces a user to accept generated passwords. By default, the list stores 60 passwords. A security administrator can change the defaults for the length of time passwords are retained and the maximum number of passwords per user.

System Logical Name	Default	Min	Max	Units
SYS$PASSWORD_HISTORY_LIFETIME	365	1	28000	days
SYS$PASSWORD_HISTORY_LIMIT	60	1	2000	absolute count

Using the DCL command DEFINE, you can change the defaults for the capacity and lifetime of the password history list. For example, to increase the capacity of the history list from 60 passwords to 100, you would add the following line to the command procedure SYLOGICALS.COM, which is located in SYS$MANAGER:

```
$ DEFINE/SYSTEM/EXEC SYS$PASSWORD_HISTORY_LIMIT 100
```

There is a correspondence between the lifetime of a password history list and the number of passwords allowed on the list. For example, if you increase the password history lifetime to four years and your passwords expire every two weeks, you would need to increase the password history limit to at least 104 (4 years times 26 passwords a year). The password history lifetime and limit can be changed dynamically, but they should be consistent across all nodes on the cluster.

Sites using secondary passwords might need to double the password limit to account for the secondary password storage.

The password history list is located in SYS$SYSTEM. The list can be redirected off the system disk using the logical name VMS$PASSWORD_HISTORY. This logical name should also be defined /SYSTEM/EXEC and placed in SYS$MANAGER:SYLOGICALS.COM.

14.1.1.2 Site-Specific Filter

Security administrators can develop a site-specific password filter to ensure that passwords are not words readily associated with their site, for example, product names or personnel names. A filter can also check for particular character variations.

To create a list of site-specific words, you write the source code, create a shareable image, install the image, and, finally, enable the policy by setting a SYSGEN parameter. See Section 22.6 for step-by-step instructions. Installing and enabling a site-specific password filter requires both SYSPRV and CMKRNL privileges. In addition, if INSTALL and SYSPRV file access auditing are enabled, multiple security alarms are generated when the password filter image is installed and the required change to the SYSGEN parameter is noted on the operator console.

The shareable image contains two global routines that are called by the VMS Set Password Utility whenever a user changes a password.

Warning: The two global routines allow a security administrator to obtain both the proposed plaintext password and its equivalent quadword hash value. All security administrators should be aware of this feature as its subversion by a malicious privileged user will compromise your system's security.

Digital recommends that you place security alarm ACEs on the password filter image and its parent directory. See Section 22.6 for instructions.

14.1.2 Specifying a Password Algorithm

The VMS operating system protects passwords from disclosure through encryption. VMS algorithms transform passwords from plaintext strings into cipher text, which is then stored in the user authorization file (UAF). Whenever a password check is done, the check is based on the encrypted password, not the plaintext password. The system password is always encrypted with an algorithm known to the VMS system.

The /ALGORITHM qualifier in the Authorize Utility allows you to define which algorithm the VMS system should use to encrypt a user's password, both primary and secondary. Your choices are the current VMS algorithm or a site-specific algorithm. The syntax is as follows:

```
/ALGORITHM=keyword=type [=value]
```

Table 14–1 lists all the keywords and types you can specify with the /ALGORITHM qualifier.

To assign the VMS password encryption algorithm for a user, you would enter the following command.

```
UAF>   MODIFY HOBBIT/ALGORITHM=PRIMARY=VMS
```

If a site-specific algorithm is selected, you must give a value to identify the algorithm.

```
UAF>   MODIFY HOBBIT/ALGORITHM=CURRENT=CUSTOMER=128
```

Section 22.6.1 provides directions for using a customer algorithm. You must create a site-specific $HASH_PASSWORD in which you define an algorithm number. This number has to correspond with the number used in the AUTHORIZE command MODIFY/ALGORITHM.

Whenever a user is assigned a site-specific algorithm, the Authorize Utility reports this information in the display provided by the SHOW command.

Table 14–1 Arguments to the /ALGORITHM Qualifier

Keyword	Function
BOTH	Set the algorithm for primary and secondary passwords.
CURRENT	Set the algorithm for the primary, secondary, both, or no passwords depending on account status. Current is the default value.
PRIMARY	Set the algorithm for the primary password only.
SECONDARY	Set the algorithm for the secondary password only.

Type	Definition
VMS	The algorithm used in the version of VMS that is running on your system.
CUSTOMER	A numeric value in the range 128–255 identifies a customer algorithm.

15 Log Manager Control Program Utility (LMCP)

The Log Manager Control Program Utility (LMCP) is a component of DECdtm services residing within the VMS Version 5.4 operating system. The log manager ensures that, as each transaction is processed, a record of each transaction state is recorded in a log file on disk.

The DECdtm transaction manager invokes the log manager to write these transaction records as necessary, ensuring that a consistent transaction outcome is achieved even in the event of a system failure. Writing log records is necessary for the consistent recovery of the transaction-specific data.

This chapter describes how a system manager can use the Log Manager Control Program Utility (LMCP) to create and manage transaction log files, and it provides a complete description of all the LMCP commands.

See Chapter 3 for a complete overview of DECdtm services.

15.1 Managing Transaction Log Files

To optimize the execution of distributed transactions on your system, you need to consider a number of factors relating to transaction log files. This section discusses these factors, providing recommendations and guidelines in the following areas:

- Using the SYS$JOURNAL logical name
- Where to place a transaction log file
- How VAXcluster failover works
- Determining the initial size required for a transaction log file
- Creating a transaction log file
- Resizing a transaction log file

Note: **To use LMCP commands, you must have SYSPRV privilege. To use the LMCP command CONVERT, you must have CMKRNL privilege. It is assumed throughout this section that system managers or other individuals who have these privileges will be implementing the procedures described herein.**

15.1.1 Defining SYS$JOURNAL

The logical name SYS$JOURNAL defines the directory location where DECdtm services expect to find log files. SYS$JOURNAL is a system-table, executive-mode logical name, normally defined in the SYS$STARTUP:SYLOGICALS.COM command procedure.

If SYS$JOURNAL is not defined in SYS$STARTUP:SYLOGICALS.COM, then a default logical name value is defined as SYS$COMMON:[SYSEXE].

You can define SYS$JOURNAL using the following command format:

DEFINE/SYSTEM/EXEC SYS$JOURNAL device:[directory]

The logical name SYS$JOURNAL can be defined as a search list. For example, the following command defines a search list consisting of two directories.

```
$ DEFINE/SYSTEM/EXEC SYS$JOURNAL DISK1:[LOGFILES], DISK2:[LOGFILES]
```

This example shows DISK1:[LOGFILES] to be the primary, or local, directory that DECdtm services always search first. DISK2:[LOGFILES] is the secondary directory, DECdtm services search this directory after the directory DISK1:[LOGFILES] is searched. If you create a transaction log file using the LMCP CREATE command, then the log file is placed in the first directory DISK1:[LOGFILES].

If a transaction log file is created on a different node using DISK2:[LOGFILES] as the primary—or local—directory and DISK1:[LOGFILES] as the secondary directory, then the search list should specify the local log file directory first. Thus, the following command defines a search list consisting of two directories, where DISK2:[LOGFILES] is the local directory and the first to be searched by DECdtm services:

```
$ DEFINE/SYSTEM/EXEC SYS$JOURNAL DISK2:[LOGFILES], DISK1:[LOGFILES]
```

If you create a transaction log file using the LMCP CREATE command, then the log file is placed in the first directory DISK2:[LOGFILES].

15.1.2 Placing a Transaction Log File

Transactions cannot be started until you have created a transaction log file, using the LMCP CREATE command. But before you create a transaction log file, you should consider where to locate it for best performance on your system.

A log file can be placed on any file-structured device that is available to the processor. The following list includes possible alternate locations for log files, in the recommended order:

1 Shadowed nonsystem disk
2 Nonsystem disk
3 Shadowed system disk
4 System disk

For increased performance, follow the general guidelines for installing a secondary page/swap file. Use a high-performance, HSC-based disk that has little activity.

You should also take into account the following considerations when locating a log file:

- Shadowed versus nonshadowed disk

Because a transaction log file is almost exclusively write-only during normal processing, a shadowed disk may be slower than a nonshadowed disk. However, a shadowed disk provides increased data availability in the event of media failure.

- Local versus cluster disk

  Although a disk on a local node can provide higher performance, particularly in an NI-based VAXcluster, if that VAXcluster member node fails, other nodes in the VAXcluster will not be able to access the failed nodes disk. (See Section 15.1.3.) Therefore, it is better if disks are mounted VAXcluster-wide and correctly defined using the logical name SYS$JOURNAL. That way, if a node fails, other nodes can still access the failed nodes disk.

In a VAXcluster, log files should be placed on disks accessible to all members of the VAXcluster. This practice facilitates VAXcluster failover by making the log files on each VAXcluster member node available to other VAXcluster members.

15.1.3 VAXcluster Failover

VAXcluster failover is a mechanism that DECdtm services provide to enable VAXcluster nodes to perform recovery for a member node that has failed.

To make VAXcluster failover work, you need to correctly define SYS$JOURNAL (as described in Section 15.1.1) so that DECdtm services can locate all transaction log files in use in the VAXcluster.

VAXcluster failover only occurs within a VAXcluster environment and is completely automatic and transparent to applications and resource managers using DECdtm services. VAXcluster failover starts when a VAXcluster member node fails and holds information that surviving VAXcluster member nodes need to process their transactions.

When VAXcluster failover is initiated, recovery proceeds while the failed node is rebooting. This allows other nodes that need information from the failed node to resolve transactions. It also allows resource managers to release locks on database records without waiting for the failed node to reboot.

Normally, each VAXcluster member node is primarily responsible for accessing its own transaction log file. Any node that requires information from a log file it does not have open must send a request for that information to the VAXcluster node member that currently has the log file open—the node normally responsible for that log file.

During VAXcluster failover, the first requesting node that requires information from a failed node opens the failed node's transaction log file to perform recovery. This action lets recovery on the failed node's log file begin while the failed node is rebooting. Normally, transaction recovery on the log file completes before the failed node has rebooted. Therefore, nodes that had their transactions blocked by the failure of the VAXcluster node have their transactions resolved before the failed node reboots. The surviving VAXcluster members proceed as if the failed node had already rebooted.

Once a VAXcluster member node has opened the log file of a failed node, all further requests from other VAXcluster member nodes are directed to the node that has opened the log file. Only one VAXcluster member node can access a failed node's log file at any one time. When the failed node has rebooted, it reacquires access to its log file and requests are passed to that rebooted VAXcluster node member once again.

15.1.4 Determining Transaction Log File Size

Use the LMCP CREATE command to create transaction log files. The /SIZE qualifier of this command specifies the size of the log file in blocks. By default, the file size is 4000 blocks. However, since performance of transaction processing applications depend on transaction logging, Digital recommends that you plan ahead when creating log files.

A number of factors must be considered when estimating transaction log file requirements. These factors include the rate of transactions executed per second and the duration of the transactions. As a quick way to estimate log file size, Digital recommends the following algorithm:

$$\textit{Transaction start rate} * \textit{Transaction duration} * 40 = \textit{log file size in disk blocks}$$

You can use the MONITOR TRANSACTION command of the Monitor Utility to determine the start rate and duration for transactions already executing on your system. (See Section 16.1 for more information about the MONITOR TRANSACTION command.)

For example, if the start rate is 5 transactions per second and the duration is 10 seconds, the calculation is:

$$5 * 10 * 40 = 2000 \textit{ blocks}$$

The recommended file size for a log file in this example is 2000 blocks.

Due to a number of factors, file size requirements can vary widely from one system to the next. Therefore, the guidelines listed here for determining log file size can provide only very rough estimates. When planning for log files, it is recommended that you overestimate, rather than underestimate, the file size.

15.1.5 Creating Transaction Log Files

Transactions cannot be started until a transaction log file exists. By default, processes for DECdtm services are started when a full VMS boot is executed.[1] The DECdtm process TP_SERVER then checks for the existence of a transaction log file on the system and continues checking every 15 seconds for the existence of a transaction log file on the system so that recovery can occur automatically, even if a log file's disk is not available when the system first boots.

[1] If you do not want to run DECdtm software, you can prevent the startup of DECdtm processes by defining the systemwide logical name SYS$DECDTM_INHIBIT. See the note at the beginning of Chapter 3 for more information.

To create a log file, use the LMCP CREATE command. Before creating a log file, you should understand the recommendations for placing and sizing log files, as described in Section 15.1.2 and Section 15.1.4.

A log file must be named with the file name SYSTEM$*node-name*, where *node-name* is the name of the node on which the log file will be used. For example, a log file created on node ORANGE should be given the file name SYSTEM$ORANGE. The default file type is LM$JOURNAL.

The default file specification for the log file is:

```
SYS$JOURNAL:.LM$JOURNAL
```

15.1.6 Example of Creating a Transaction Log File

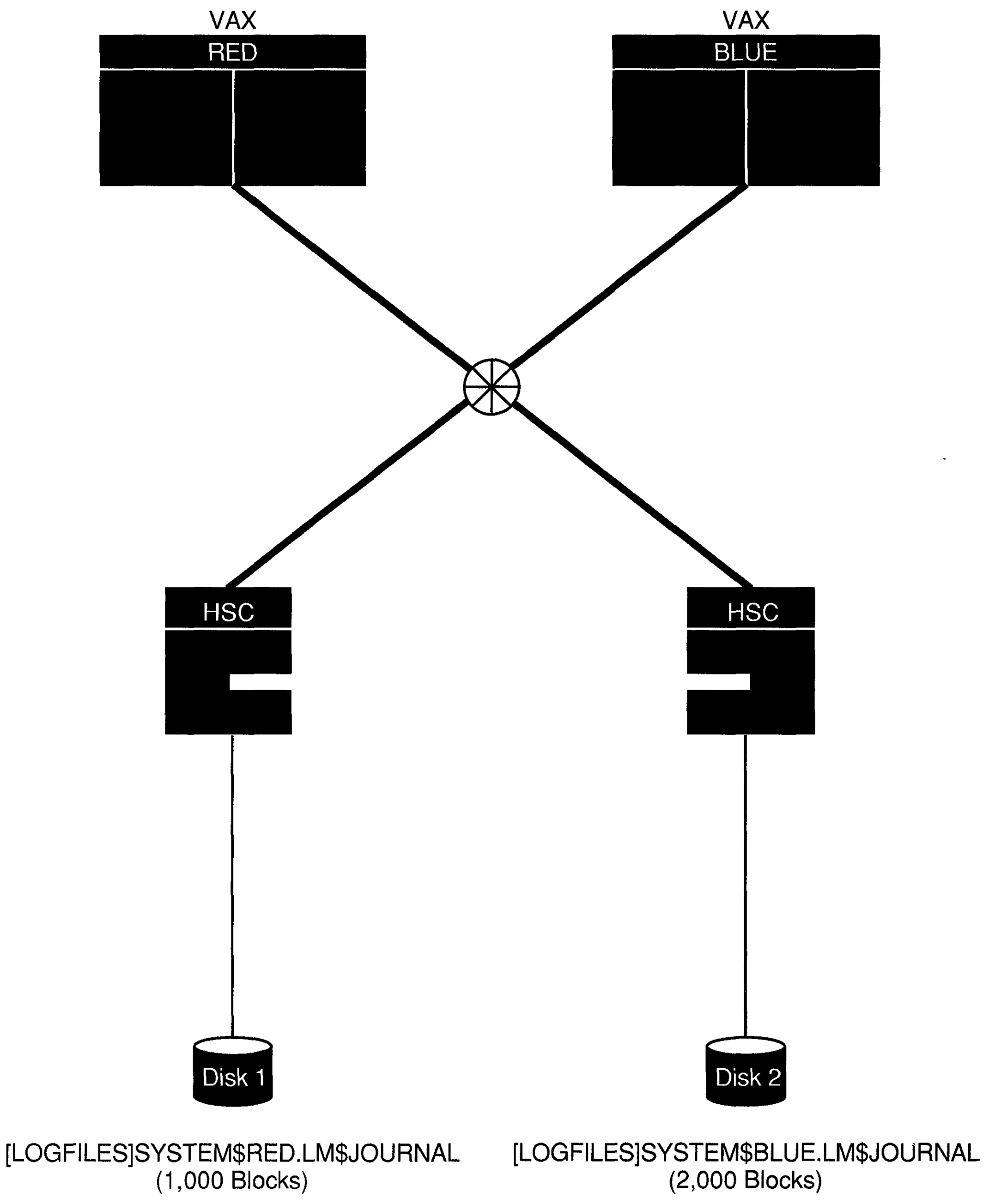
This section summarizes the steps involved in creating transaction log files for a sample VAXcluster system.

Note: **To use LMCP commands, you must have SYSPRV privilege. To use the LMCP command CONVERT, you must have CMKRNL privilege. It is assumed throughout this section that system managers or other individuals who have these privileges will be implementing the procedures described herein.**

In this example, the conditions are as follows:

- The sample VAXcluster consists of two nodes, RED and BLUE, with shared access to the devices named DISK1 and DISK2.
- The system manager wants to set up an initial configuration of transaction log files that allows DECdtm services to perform VAXcluster failover.
- The system manager needs to create two log files, one for each node.
- The system manager has determined that the initial log file size will be 1000 blocks on node RED and 2000 blocks on node BLUE. Figure 15–1 shows the desired configuration.

Figure 15–1 Sample Transaction Log File Configuration on Two-Node VAXcluster



Based on the conditions established for this example, the system manager would follow these steps to configure the VAXcluster:

1 On node RED, the system manager would establish a search list for log files by adding the following line to the SYS$STARTUP:SYLOGICALS command procedure:

```
$ DEFINE/SYSTEM/EXEC SYS$JOURNAL DISK1:[LOGFILES], DISK2:[LOGFILES]
```

2 On node BLUE, the system manager would define a similar search list for transaction log files by adding the following line to the SYS$STARTUP:SYLOGICALS command procedure. Because the CREATE command creates a log file in the first directory pointed to by SYS$JOURNAL, this search list will specify the local node log file directory first.

```
$ DEFINE/SYSTEM/EXEC SYS$JOURNAL DISK2:[LOGFILES], DISK1:[LOGFILES]
```

3 Assuming that SYS$JOURNAL is defined, the system manager would then create the log files for each node using the LMCP CREATE command. On node RED, for example, the system manager would enter the following LMCP command to create the log file SYSTEM$RED.LM$JOURNAL with the desired file size:

```
LMCP> CREATE LOGFILE/SIZE=1000 SYSTEM$RED
```

4 If SYS$JOURNAL has not been defined, all transactions will abort until the DECdtm services locate the transaction log file. Therefore, in this case, the system manager would also need to specify the device and directory when creating the log file. For example:

```
LMCP> CREATE LOGFILE/SIZE=1000 DISK1:[LOGFILES]SYSTEM$RED
```

5 The system manager would then repeat a similar procedure on node BLUE by entering following LMCP command to create the transaction log file SYSTEM$BLUE.LM$JOURNAL with the desired log file size:

```
LMCP> CREATE LOGFILE/SIZE=2000 SYSTEM$BLUE
```

15.1.7 Resizing and Moving Transaction Log Files

If transaction processing performance degrades on your system (indicated by the rate of transaction stalls), you might need to use the LMCP CONVERT command to increase the size of the transaction log file, or you might need to move the log file to a higher performance disk.

To check for the rate of transaction stalls, use the LMCP command SHOW LOG/CURRENT, which displays information about the currently active transaction log file. This display shows the number of checkpoints and stalls that have occurred since DECdtm services were started and indicates whether a checkpoint or stall is currently in progress.

Checkpoints are normal, regular, log manager events that are used to maintain the log file during transaction execution; they do not indicate degradation in log file performance.

The log manager stalls transactions when insufficient space is available in the log file for correct and successful transaction execution. A high rate of stalls or a permanent stall condition indicates that the log file size should be increased. In such a case, use the LMCP command CONVERT to increase the size of the log file. Occasional stall events might be caused by transitory system activities such as VAXcluster transition events and do not necessarily indicate a permanent shortage of space in the log file.

You can also use the Monitor Utility to check for transaction processing degradation.

The necessary capacity for a log file depends on the number of simultaneous transactions and other factors. Because these factors are variable, Digital cannot recommend the amount of increased size for a transaction log file. You should estimate the percentage of increased transaction workload that caused the log to stall.

Prior to moving or resizing a log file, the system manager must do the following:

1 Disable the transaction log file.

   The log file should be disabled before the system is rebooted so that DECdtm services will not re-open the log file after the reboot. The recommended method of disabling a log file is to rename it so that it cannot be found by DECdtm services. Rename the log file with the file type LM$OLD. For example, if the original log file is called SYS$JOURNAL:SYSTEM$ORANGE.LM$JOURNAL, it should be renamed SYS$JOURNAL:SYSTEM$ORANGE.LM$OLD.

2 Reboot the system.

   A reboot is necessary because DECdtm services are an integral part of the VMS Executive and cannot be started or stopped independently of the VMS operating system. Because of this requirement, serious considerations should be given to the initial configuration of log files.

After these steps have been completed successfully, the system manager must perform the following convert procedure to change the size of the transaction log file:

1 Use the LMCP command CONVERT to move the transaction records from the old log file to the new log file and increase its size. Name the new file SYSTEM$*node-name*.LM$JOURNAL.

2 If the convert is successful, delete the old log file.

The system manager can move the log file to an alternate location by following these steps:

1 Edit SYS$STARTUP:SYLOGICALS.COM on all nodes in the VAXcluster to include a new definition for the logical name SYS$JOURNAL, as follows:

```
$ DEFINE/SYSTEM/EXEC SYS$JOURNAL device:[directory]
```

2 Reboot the system.

3 Copy the log file to the new location, using the following command:

```
$  COPY DEVICE:[DIRECTORY]SYSTEM$node-name.LM$OLD -
_$  SYS$JOURNAL:SYSTEM$node-name.LM$JOURNAL
```

4 If the copy is successful, delete the old log file.

15.2 Format of Transaction Log Files

A transaction log file consists of a file header, section headers, and transaction records.

A log file header contains information about the log file, such as its version number, size, unique identifier, and checkpoints. Checkpoints are mechanisms that bound the search for active transaction records. Therefore, in the event of a system failure, the log manager can efficiently locate the active transaction records needed for system recovery. (An active transaction is one that has not completed.)

A log file is organized into sections and each section has a section header containing information about its own characteristics. This information is used by the log manager to find and read transaction records efficiently.

The transaction record header identifies the record number and information about the transaction, such as the transaction's state and its unique transaction identifier (TID). A transaction can be in any of three states:

- PREPARED—The transaction is in a state where it can be either committed or rolled back.
- COMMITTED—The transaction manager has enough information to complete the transaction even though the participants in the transaction have not finished all their operations.
- FORGOTTEN—The participants have enough information to complete processing the transaction and will no longer ask about the transaction. Therefore, the transaction can be forgotten.

The transaction record data gives information about the DECdtm version number, the log identifier, and the name and type of resource manager the transaction is involved with.

Example 15–1 shows a portion of a sample transaction log file.

Example 15–1 Sample Transaction Log File

```
Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$BLUE.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0                                                    ❶
Log File UID:   9D519DC0-698E-0092-DF95-00000000B20D (21-JUN-1989 09:19:44.54)
Penultimate Checkpoint: 00000012C45E 005E
Last Checkpoint:        000000133E39 0039

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$BLUE.LM$JOURNAL;1
Present Length:     166 (000000A6) Last Length:        512 (00000200)    ❷
VBN Offset:        2503 (000009C7) Virtual Block:     2505 (000009C9)
Section:              4 (00000004)

Record number 3 (00000003)❸, 77 (004D) bytes ❹
Transaction state (1):  PREPARED ❺
Transaction ID: 2B065A40-6E88-0092-EC42-00000000B208  ❻ (27-JUN-1989 17:16:11.62)
DECdtm Services Log Format V1.0 ❼
Type (3): LOCAL RM ❽      Log ID:00000000-0000-0000-0000-000000000000 ❿
Name (6): "SERVER" ❾(5245 56524553)
Type (4): PARENT NODE ❽  Log ID:6900BC00-6B4F-0092-C8BD-00000000B208 ❿
Name (10): "SYSTEM$RED" ❾ (4445 52244D45 54535953)
    .
    .
    .
```

❶ Log header—Contains information about the log's characteristics.

❷ Section header—The section header of multiple transaction records.

❸ Record number—A unique record number in decimal and hexadecimal.

❹ Record size—The record size in decimal and hexadecimal.

❺ Transaction state—The three states a transaction can be in are PREPARED, COMMITTED, and FORGOTTEN.

❻ Transaction ID (TID)—Each transaction has its own unique transaction identifier assigned by the transaction manager.

❼ DECdtm services version number—The software version number of DECdtm services.

❽ Participant type—The types of participant in the transaction.

Participant types include:

- CHILD NODE—A subordinate transaction manager
- PARENT NODE—The immediate parent transaction manager
- LOCAL RM—The recoverable resource manager on the local node

❾ Participant name—The name of the participant in the transaction, also given in hexadecimal.

❿ Log ID—A unique hexadecimal log identifier the participant uses to write its own recovery records.

In Example 15–1, the fields labeled ❶ comprise the log header, ❷ comprise the section header, ❸ through ❻ comprise the record header, and ❼ through ❿ comprise the record data.

LMCP Usage Summary

The Log Manager Control Program is a VMS utility that lets you create and maintain log files of transaction records.

FORMAT

$ RUN SYS$SYSTEM:LMCP

usage summary

To invoke LMCP, enter the following DCL command:

```
$ RUN SYS$SYSTEM:LMCP
```

LMCP returns the following prompt:

```
LMCP>
```

At the *LMCP>* prompt, you can enter LMCP commands. To exit LMCP, enter EXIT at the *LMCP>* prompt, or press Ctrl/Z.

You can also execute a single LMCP command by using a DCL string assignment statement, as shown in the following example:

```
$ LMCP :== $LMCP
$ LMCP SHOW LOGFILE SYSTEM$YELLOW
```

In this example, LMCP executes the SHOW command and returns control to DCL.

To use LMCP commands, you must have SYSPRV privilege. To use the LMCP command CONVERT, you must have CMKRNL privilege.

LMCP Commands

This section describes the following LMCP commands and provides examples of how to use them:

- CONVERT
- CREATE
- DUMP
- HELP
- REPAIR, including the following REPAIR subcommands:

  ABORT
  COMMIT
  EXIT
  FORGET
  HELP
  NEXT
- SHOW

Note: **To use LMCP commands, you must have SYSPRV privilege. To use the LMCP command CONVERT, you must have CMKRNL privilege. It is assumed throughout this section that system managers or other individuals who have these privileges will be implementing the procedures described herein.**

You can abbreviate any command, parameter, or qualifier as long as the abbreviation is unique.

CONVERT

Converts a log file on a given node by transferring the active transaction records from the specified source log file to the specified destination log file. To use the CONVERT command, you need CMKRNL privilege.

FORMAT

CONVERT LOGFILE *source_filespec destination_filespec [qualifier...]*

PARAMETER

source_filespec

Specifies the file specification of the log file from which active transaction records are to be copied.

destination_filespec

Specifies the file specification of the log file where active transaction records are to be written.

QUALIFIERS

/OWNER=owner_id

Associates an owner or user identification code (UIC) with the log file to be created. You specify the UIC using the standard UIC format as described in the *VMS DCL Concepts Manual.* The default UIC is one of the following:

- The owner UIC of an existing version of the file if the file creator has extended privileges
- The owner UIC of the parent directory if the file creator has extended privileges
- The owner UIC of the creator

/SIZE=file_size

Specifies the size of the log file in blocks. The minimum log file size is 100 blocks.

DESCRIPTION

Use the CONVERT command to resize a log file. For example, if transaction processing performance degrades on your system, then you may need to increase the log file size. See Section 15.1.7 for more information about resizing and moving log files.

EXAMPLE

```
LMCP>  CONVERT LOGFILE SYSTEM$RED.LM$OLD SYSTEM$RED/SIZE=8000
```

This command transfers all active transaction records from the log file SYSTEM$RED.LM$OLD to SYSTEM$RED and specifies a log file size of 8000 blocks.

CREATE

Creates a log file for a specific node.

FORMAT

CREATE LOGFILE *filespec [qualifier...]*

PARAMETER

filespec
Specifies the file specification of the log file to be created. DECdtm services expect the file name to be in the format *SYSTEM$node-name*, where *node-name* is the name of the node that will use the log file.

QUALIFIERS

/NEW_VERSION
Creates a new version of a log file if a log file with an identical specification already exists. The new log file is created with the same name and type but with a version number one higher than the highest existing version. Note that, once the new version of the transaction log file is created, then any transaction records in the previous log cannot be accessed.

If the /NEW_VERSION qualifier is specified for a log file that does not exist, no new file will be created. Instead, an error will be returned.

/OWNER=owner_id
Associates an owner or user identification code (UIC) with the log file to be created. Specify the UIC using the standard UIC format as described in the *VMS DCL Concepts Manual*. The default UIC will be one of the following:

- The owner UIC of an existing version of the file if the file creator has extended privileges
- The owner UIC of the parent directory if the file creator has extended privileges
- The owner UIC of the creator

/SIZE=file_size
Specifies the size of the log file in blocks. The minimum log file size is 100 blocks, and the default log file size is 4000 blocks.

DESCRIPTION

By default, log files are created in the directory specified by SYS$JOURNAL, with a file extension of LM$JOURNAL and a size of 4000 blocks. To identify the name of the node that will use the log file, the file name must be in the following format:

```
SYSTEM$node-name
```

EXAMPLES

1
```
LMCP> CREATE LOGFILE SYSTEM$BLUE/OWNER=GONZALES/SIZE=4400
```

This command creates a log file called SYSTEM$BLUE.LM$JOURNAL, associates it with user GONZALES and specifies a file size of 4400 blocks.

2
```
LMCP> CREATE LOGFILE SYSTEM$YELLOW/OWNER=[USER,FRED]/SIZE=4000
```

This command creates a log file called SYSTEM$YELLOW.LM$JOURNAL, associates it with the UIC group USER, member FRED, and specifies a log file size of 4000 blocks.

3
```
LMCP> CREATE LOGFILE SYSTEM$BLUE/NEW_VERSION/OWNER=GONZALES/SIZE=4400
```

This command creates a new log file that supersedes the current highest version of SYSTEM$BLUE.LM$JOURNAL and is given a version number one higher. Also, the new log file is associated with user GONZALES and specifies a file size of 4400 blocks.

DUMP

Displays (or "dumps") the contents of a specified log file.

FORMAT

DUMP *filespec [qualifier...]*

PARAMETER

filespec
Specifies the file specification of the log file.

QUALIFIERS

/ACTIVE
Specifies that only records relating to active transactions within the log file are to be displayed.

/FORMAT(default)
/NOFORMAT
Displays the contents of the log file as formatted records. If the /NOFORMAT qualifier is specified, only the log file header is displayed.

/HEX
Specifies that the contents of the log file dump are displayed as ASCII characters and hexadecimal longwords. Use both the /NOFORMAT and /HEX qualifiers to format a DUMP operation in hexadecimal only.

/LOGID=log_identifier
Specifies the log identifier, in hexadecimal format, associated with a specific resource manager. The /LOGID qualifier can be used only in conjunction with the /RM qualifier.

/OUTPUT[=filespec]
Specifies that the output is written to the file specified. By default, the DUMP command writes the output to SYS$OUTPUT. If you enter /OUTPUT with no file specification, LMCP_DUMP is the default file name and LIS is the default type.

/RM=rm_identifier
Selects the transactions to be displayed according to the resource manager participating in the transaction. The argument supplied for the **rm_identifier** can be either the ASCII character string for the resource manager name or its hexadecimal equivalent. When specifying a hexadecimal string, you must prefix the characters %X to the hexadecimal string.

If a partial resource manager name is supplied as the argument for the **rm_identifier**, LMCP selects all resource managers having names that begin with the supplied string.

/STATE=transaction_state

Selects the transactions to be displayed according to their transaction states. A value of either PREPARED or COMMITTED can be supplied as an argument to the /STATE qualifier. If the /STATE qualifier is not supplied, all transactions records are selected.

/TID=transaction_id

Selects the transactions to be displayed according to the transaction identifier. The argument supplied for the **transaction_id** must be a hexadecimal character string.

DESCRIPTION

If you entered the DUMP command, the contents of the log file you specified are displayed. By default the log file records are displayed as formatted records.

EXAMPLES

1

```
LMCP>  DUMP SYSTEM$BLUE/HEX/NOFORMAT

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$BLUE.LM$JOURNAL;2
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   9D519DC0-698E-0092-DF95-00000000B20D (21-JUN-1989 09:19:44.54)
Penultimate Checkpoint: 00000012C45E 005E
Last Checkpoint:        000000133E39 0039


Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$BLUE.LM$JOURNAL;2
Present Length:      68 (00000044) Last Length:        512 (00000200)
VBN Offset:        2504 (000009C8) Virtual Block:     2506 (000009CA)
Section:              3 (00000003)

Record number 1 (00000001), 48 (0030) bytes
Transaction state (2):  COMMITTED
Transaction ID: 2B065A40-6E88-0092-EC42-00000000B208 (27-JUN-1989 17:16:11.62)
 01000000 00B20842 EC00926E 882B065A 40020030 0..@Z.+.n..ìB.².... 0000
 00060000 00000000 00000000 00000000 00000300 .................... 0014
                            00305245 56524553 SERVER0.             0028


Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$BLUE.LM$JOURNAL;2
Present Length:     166 (000000A6) Last Length:        512 (00000200)
VBN Offset:        2503 (000009C7) Virtual Block:     2505 (000009C9)
Section:              4 (00000004)

Record number 3 (00000003), 77 (004D) bytes
Transaction state (1):  PREPARED
Transaction ID: 2B065A40-6E88-0092-EC42-00000000B208 (27-JUN-1989 17:16:11.62)
 01000000 00B20842 EC00926E 882B065A 4001004D M..@Z.+.n..ìB.².... 0000
 00060000 00000000 00000000 00000000 00000300 .................... 0014
 00B208BD C800926B 4F6900BC 00045245 56524553 SERVER..¼.iOk..È½.². 0028
       00 4D444552 244D4554 53595300 0A000000 .....SYSTEM$REDM.    003C

Record number 2 (00000002), 21 (0015) bytes
Transaction state (0):  FORGOTTEN
Transaction ID: 2A6DC3C0-6E88-0092-EC42-00000000B208 (27-JUN-1989 17:16:10.62)
 15000000 00B20842 EC00926E 882A6DC3 C0000015 ...ÀÃm*.n..ìB.².... 0000
                                           00 .                    0014
```

```
Record number 1 (00000001), 48 (0030) bytes
Transaction state (2):  COMMITTED
Transaction ID: 2A6DC3C0-6E88-0092-EC42-00000000B208 (27-JUN-1989 17:16:10.62)
 01000000 00B20842 EC00926E 882A6DC3 C0020030 0..ÀÃm*.n..ìB.².... 0000
 00060000 00000000 00000000 00000000 00000300 .................... 0014
                            00305245 56524553 SERVER0.             0028
    .
    .
    .
```

This command produces a dump—in hexadecimal format—of the specified log file.

2

```
LMCP> DUMP SYSTEM$BLUE/HEX/OUTPUT=EXAMPLE
```

This command writes a dump—in hexadecimal format—of the specified log file to the file EXAMPLE.LIS.

3

```
LMCP> DUMP SYSTEM$PURPLE/ACTIVE

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$PURPLE.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   2F99A820-BAB2-0092-9310-00000000B1FE ( 2-OCT-1989 15:28:26.53)
Penultimate Checkpoint: 000000000000 0000
Last Checkpoint:        000000010BD9 01D9

Transaction state (2):  COMMITTED
Transaction ID: 84C67760-BAB2-0092-8243-00000000B1FE ( 2-OCT-1989 15:30:49.43)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (11): "THREAD_5.29" (39322E 355F4441 45524854)
Type (2): CHILD NODE   Log ID: 748FF0C0-B52A-0092-9011-00000000B204
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)

Transaction state (2):  COMMITTED
Transaction ID: 84C1E380-BAB2-0092-8243-00000000B1FE ( 2-OCT-1989 15:30:49.40)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (11): "THREAD_4.29" (39322E 345F4441 45524854)
Type (2): CHILD NODE   Log ID: 748FF0C0-B52A-0092-9011-00000000B204
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)

Total of 2 transactions active, 0 prepared and 2 committed.
```

This command displays a dump of all active transactions of the specified log file.

4

```
LMCP> DUMP SYSTEM$GREEN/STATE=PREPARED

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$GREEN.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   748FF0C0-B52A-0092-9011-00000000B204 (25-SEP-1989 14:34:14.86)
Penultimate Checkpoint: 00000002DDB7 01B7
Last Checkpoint:        00000002FC41 0241

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$GREEN.LM$JOURNAL;1
Present Length:     169 (000000A9) Last Length:        512 (00000200)
VBN Offset:         380 (0000017C) Virtual Block:      382 (0000017E)
Section:              2 (00000002)
```

```
Record number 3 (00000003), 80 (0050) bytes
Transaction state (1):  PREPARED
Transaction ID: F30CAF60-BA84-0092-8FA6-00000000B24B ( 2-OCT-1989 10:04:37.59)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (6): "SERVER" (5245 56524553)
Type (4): PARENT NODE  Log ID: 68165820-BA84-0092-FC95-00000000B24B
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$GREEN.LM$JOURNAL;1
Present Length:     100 (00000064) Last Length:        512 (00000200)
VBN Offset:         379 (0000017B) Virtual Block:      381 (0000017D)
Section:              3 (00000003)

Record number 1 (00000001), 80 (0050) bytes
Transaction state (1):  PREPARED
Transaction ID: F2F8D940-BA84-0092-8FA6-00000000B24B ( 2-OCT-1989 10:04:37.46)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (6): "SERVER" (5245 56524553)
Type (4): PARENT NODE  Log ID: 68165820-BA84-0092-FC95-00000000B24B
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)
    .
    .
    .
Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$GREEN.LM$JOURNAL;1
Present Length:     100 (00000064) Last Length:          0 (00000000)
VBN Offset:           0 (00000000) Virtual Block:        2 (00000002)
Section:            376 (00000178)

Record number 1 (00000001), 80 (0050) bytes
Transaction state (1):  PREPARED
Transaction ID: 809D5600-BA84-0092-8FA6-00000000B24B ( 2-OCT-1989 10:01:25.60)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (6): "SERVER" (5245 56524553)
Type (4): PARENT NODE  Log ID: 68165820-BA84-0092-FC95-00000000B24B
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)
```

This command displays a dump of all prepared records of the specified log file.

5
```
LMCP> DUMP SYSTEM$GREEN/TID=FAC21DE2-BA88-0092-8FA6-00000000B24B/ACTIVE

Dump of log file DISK1:[MASTER.JOURNALS]SYSTEM$GREEN.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   68165820-BA84-0092-FC95-00000000B24B ( 2-OCT-1989 10:00:44.45)
Penultimate Checkpoint: 0000000711D3 13D3
Last Checkpoint:        000000072742 1542

Transaction state (2):  COMMITTED
Transaction ID: FAC21DE2-BA88-0092-8FA6-00000000B24B ( 2-OCT-1989 10:33:28.51)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (11): "THREAD_13.4" (342E33 315F4441 45524854)

Total of 2 transactions active, 0 prepared and 2 committed.
```

This command displays a dump of the record for the specified active transaction. (If the transaction is not active, only the active transaction count number is displayed.)

HELP

Provides information about LMCP commands and parameters.

FORMAT

HELP *[help-topic [help-subtopic]]*

PARAMETER

help-topic
Specifies the command to be explained.

help-subtopic
Specifies the qualifier to be explained.

EXAMPLES

1
```
LMCP> HELP

  Information available:

   CONVERT       CREATE        Description        DUMP        EXIT        HELP
   REPAIR        SHOW
```

This command invokes help and displays all commands for which further information exists.

2
```
LMCP> HELP CREATE

CREATE

   Creates a log file.

   Format:

     CREATE LOGFILE filespec [qualifier...]


  Additional information available:

  filespec  qualifiers
  /OWNER     /SIZE
  Example
```

This command provides a description of the CREATE command.

REPAIR

Selects records within a log file so that transactions can be repaired by having their transaction states changed. Once the transaction records have been selected, REPAIR subcommands can be used to change the transaction states.

Note: **Because the REPAIR command lets you change transaction states locally without regard to the global state, you must use this command with caution. If you do not change all necessary characteristics of a transaction record, the transaction could be placed in an inconsistent state, resulting in potential data loss.**

FORMAT

REPAIR *filespec [qualifier...]*

PARAMETER

filespec

Specifies the file specification of the log file containing the transaction records to be repaired.

QUALIFIERS

/LOGID=log_identifier

Specifies the log identifier, in hexadecimal format, associated with a specific resource manager. The /LOGID qualifier can be used only in conjunction with the /RM qualifier.

/RM=rm_identifier

Selects the transactions to be repaired according to the resource manager participating in the transaction. The argument supplied for the **rm_identifier** can be either the ASCII character string for the resource manager name or its hexadecimal equivalent. When specifying a hexadecimal string, you must prefix the characters %X to the hexadecimal string.

If a partial resource manager name is supplied as the argument for the **rm_identifier**, LMCP selects all resource managers having names that begin with the supplied string.

/STATE=transaction_state

Selects the transactions to be repaired according to their transaction states. A value of either PREPARED or COMMITTED can be supplied as an argument to the /STATE qualifier. If the /STATE qualifier is not supplied, all active transactions (both PREPARED and COMMITTED) are selected.

/TID=transaction_id

Selects the transactions to be repaired according to the transaction identifier. The argument supplied for the **transaction_id** must be a hexadecimal character string.

DESCRIPTION

The REPAIR command allows you to manually modify active transaction records in a log file.

When you enter the REPAIR command, LMCP enters the REPAIR command mode and produces a listing of the log file's contents, as selected by the specified REPAIR command qualifier. Each transaction record is displayed sequentially so that you can modify its characteristics. After each record in the filtered log file is displayed, the *REPAIR>* prompt returns. You can then enter REPAIR subcommands to change the transaction states of specific records. The REPAIR subcommands are as follows:

ABORT
COMMIT
EXIT
FORGET
HELP
NEXT

Once you finish modifying a transaction record, you can use the REPAIR subcommand NEXT to advance to the next sequential record in the file.

To return to the *LMCP>* prompt, you must exit the REPAIR command mode by entering the EXIT subcommand or by pressing Ctrl/Z.

The sections that follow the REPAIR command examples describe each of the REPAIR subcommands.

EXAMPLES

1
```
LMCP> REPAIR SYSTEM$ORANGE/STATE=PREPARED/RM=LOGL
```

This command selects all PREPARED transaction records in the log file SYSTEM$ORANGE. It specifies that only records from participating resource managers having names beginning with "LOGL" are to be selected.

2
```
LMCP> REPAIR SYSTEM$ORANGE/RM=LOGLOAD -
_LMCP> /LOGID=68165820-BA84-0092-FC95-00000000B24B
```

This command selects all active transaction records in the log file SYSTEM$ORANGE. It specifies that only records with a participating resource manager called LOGLOAD and associated log identifier of 68165820-BA84-0092-FC95-00000000B24B are to be selected.

3
```
LMCP> REPAIR SYSTEM$ORANGE/RM=%X534552564552
```

This command selects all active transaction records in the log file SYSTEM$ORANGE. It specifies that only records from a participating resource manager with a hexadecimal name 534552564552 are to be selected.

4
```
LMCP> REPAIR SYSTEM$ORANGE -
_LMCP> /TID=8C689380-BA84-0092-8FA6-00000000B24B
```

This command selects the active transaction record in the log file SYSTEM$ORANGE. It specifies that only the record for the transaction with a hexadecimal TID 8C689380-BA84-0092-8FA6-00000000B24B is to be selected.

ABORT

Changes the state of a transaction from PREPARED to ABORTED.

FORMAT ABORT

EXAMPLE

```
LMCP> REPAIR SYSTEM$RED

Dump of log file DISK$MASTER:[MASTER.JOURNALS]SYSTEM$RED.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   748FF0C0-B52A-0092-9011-00000000B204 (25-SEP-1989 14:34:14.86)
Penultimate Checkpoint: 000000073E2D 042D
Last Checkpoint:        000000077D7C 037C

Transaction state (1):  PREPARED
Transaction ID: FACFD981-BA88-0092-8FA6-00000000B24B ( 2-OCT-1989 10:33:28.60)
DECdtm services V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (6): "SERVER" (5245 56524553)
Type (4): PARENT NODE  Log ID: 68165820-BA84-0092-FC95-00000000B24B
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)
REPAIR> ABORT
REPAIR> EXIT
LMCP>
```

The initial REPAIR command selects all active transaction records in the log file SYSTEM$RED. The ABORT subcommand changes the state of the presented transaction from PREPARED to ABORTED. The EXIT subcommand exits the REPAIR command mode.

COMMIT

Changes the state of a transaction from PREPARED to COMMITTED.

FORMAT COMMIT

EXAMPLE

```
LMCP>  REPAIR SYSTEM$RED

Dump of log file DISK$MASTER:[MASTER.JOURNALS]SYSTEM$RED.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   748FF0C0-B52A-0092-9011-00000000B204 (25-SEP-1989 14:34:14.86)
Penultimate Checkpoint: 000000073E2D 042D
Last Checkpoint:        000000077D7C 037C

Transaction state (1):  PREPARED
Transaction ID: FACFD981-BA88-0092-8FA6-00000000B24B ( 2-OCT-1989 10:33:28.60)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (6): "SERVER" (5245 56524553)
Type (4): PARENT NODE  Log ID: 68165820-BA84-0092-FC95-00000000B24B
Name (13): "SYSTEM$ORANGE" (45 474E4152 4F244D45 54535953)
REPAIR>  COMMIT
REPAIR>  EXIT
LMCP>
```

The initial REPAIR command selects all active transaction records in the log file SYSTEM$RED. The COMMIT subcommand changes the state of the transaction from PREPARED to COMMITTED. The EXIT subcommand exits the REPAIR command mode.

EXIT

Exits the REPAIR command mode and returns the *LMCP>* prompt.

FORMAT

EXIT

FORGET

Specifies that a transaction with a state of COMMITTED can be forgotten, which means the committed transaction record can be removed from the log file.

FORMAT FORGET

EXAMPLE

```
LMCP>  REPAIR SYSTEM$RED

Dump of log file DISK$MASTER:[MASTER.JOURNALS]SYSTEM$RED.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:    748FF0C0-B52A-0092-9011-00000000B204 (25-SEP-1989 14:34:14.86)
Penultimate Checkpoint: 000000073E2D 042D
Last Checkpoint:        000000077D7C 037C

Transaction state (2):  COMMITTED
Transaction ID: F2F8D940-BA84-0092-8FA6-00000000B24B ( 2-OCT-1989 10:04:37.46)
DECdtm Services Log Format V1.0
Type (3): LOCAL RM     Log ID: 00000000-0000-0000-0000-000000000000
Name (10): "THREAD_6.4" (342E 365F4441 45524854)
REPAIR>  FORGET
REPAIR>  NEXT
    .
    .
    .
```

The initial REPAIR command selects all active transaction records in the log file SYSTEM$RED. The FORGET subcommand specifies that the transaction can be forgotten. The NEXT subcommand advances to the next record.

HELP

Provides information about REPAIR subcommands and parameters.

FORMAT

HELP *[help-topic [help-subtopic]]*

PARAMETER

help-topic
Specifies the subcommand to be explained.

help-subtopic
Specifies the qualifier to be explained.

EXAMPLES

1
```
REPAIR> HELP

REPAIR
  SUBCOMMANDS

      Entering the REPAIR command produces a listing of the log file's
      contents, as selected by the optional REPAIR command qualifiers. Each
      transaction record is displayed sequentially, so that a user can modify
      its characteristics.

      After each record in the filtered log file is displayed,
      the REPAIR> prompt is returned.  A user can then issue REPAIR
      subcommands to change the transaction states of specific records.
      A user must issue a NEXT subcommand to advance to the next sequential
      record in the file.

      To return to the LMCP> prompt, a user must exit the REPAIR command
      mode by entering the EXIT subcommand or by pressing Ctrl/Z.


    Additional information available:

    ABORT      COMMIT     EXIT       FORGET     NEXT
```

This command invokes help and displays all subcommands for which further information exists.

```
2   REPAIR> HELP ABORT

    REPAIR
      SUBCOMMANDS
        ABORT

            Changes the state of a transaction from PREPARED to ABORTED.
            Format:
                ABORT
```

This command provides a description of the ABORT subcommand.

NEXT

Advances to the next record in a transaction log.

FORMAT **NEXT**

SHOW

Lists information about transaction log files.

FORMAT

SHOW LOGFILE *filespec [qualifier...]*

PARAMETER

filespec

Specifies one or more log files to be listed. The syntax of the file specification determines which files will be listed, as follows:

- If you enter a file name or a file name containing a wildcard character, the SHOW command lists each file matching the name specified.
- If you do not enter a file specification, the SHOW command lists all log files in the directory SYS$JOURNAL.

QUALIFIER

/CURRENT

Specifies that information about the currently active log file, is shown. This information includes the number of checkpoints and stalls that have occurred since DECdtm services were started up and indicates whether a checkpoint or stall is currently in progress.

Note that no file specification is necessary when the qualifier /CURRENT is used.

/FULL

Lists all log file attributes.

/OUTPUT[=filespec]

Specifies that the output be written to the file specified. By default, the SHOW command writes the output to SYS$OUTPUT. If you enter /OUTPUT with no file specification, then LMCP_SHOW is the default file name and LIS is the default type.

DESCRIPTION

The SHOW command produces a list of existing log files matching the selection criteria specified. The asterisk and percent sign wildcard characters can be passed to the SHOW command to represent file names.

EXAMPLES

1

```
LMCP> SHOW LOGFILE SYSTEM$B*/FULL

Directory of DISK$MASTER:[MASTER.JOURNALS]

DISK$MASTER:[MASTER.JOURNALS]SYSTEM$BLUE.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   275300C0-7A71-0092-D3A8-00000000B232 (12-JUL-1989 21:01:40.94)
Penultimate Checkpoint: 0000CE644AF2 02F2
Last Checkpoint:        0000CE6457F2 03F2
```

```
DISK$MASTER:[MASTER.JOURNALS]SYSTEM$BLACK.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   9D519DC0-698E-0092-DF95-00000000B20D (21-JUN-1989 09:19:44.54)
Penultimate Checkpoint: 00000012C45E 005E
Last Checkpoint:        000000133E39 0039

DISK$MASTER:[MASTER.JOURNALS]SYSTEM$BRONZE.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   21847980-5F78-0092-3F5D-00000000B1FF ( 8-JUN-1989 13:13:36.28)
Penultimate Checkpoint: 000000ECADE5 41E5
Last Checkpoint:        000000F105FC 41FC

DISK$MASTER:[MASTER.JOURNALS]SYSTEM$BROWN.LM$JOURNAL;1
End of file block 4000 / Allocated 4000
Log Version 1.0
Log File UID:   A6173DC0-3DE2-0092-0000-00000000B1FF (26-APR-1989 19:30:25.82)
Penultimate Checkpoint: 00000C8B4819 2019
Last Checkpoint:        00000C8BC15B 335B

Total of 4 files.
```

This command lists all log files with file names beginning with SYSTEM$B.

2

```
LMCP>  SHOW LOGFILE
Directory of DISK1:[MASTER.LOGFILES]

SYSTEM$BLACK.LM$JOURNAL;1
SYSTEM$BLUE.LM$JOURNAL;1

Total of 2 files.

Directory of DISK1:[MASTER.NAMES]

SYSTEM$GREEN.LM$JOURNAL;1
SYSTEM$ORANGE.LM$JOURNAL;1
SYSTEM$RED.LM$JOURNAL;1

Total of 3 files.

Grand total of 2 directories 5 files.
```

This command lists all directories equivalent to SYS$JOURNAL and their log files.

3

```
LMCP>  SHOW LOGFILE SYSTEM$RED/FULL/OUTPUT=EXAMPLE
```

This command lists all percentage information for the specified log file and writes it to the file EXAMPLE.LIS.

4

```
LMCP>  SHOW LOGFILE/CURRENT

Checkpoints started/ended  124/123
Stalls started/ended         1/1
Log status: checkpoint in progress, no stall in progress
```

This command shows status information about the currently active log file.

16 Monitor Utility (MONITOR)

The VMS Monitor Utility (MONITOR) is a system management tool that you can use to obtain information about operating system performance. This chapter describes the following enhancements to Version 5.4 of the VMS Monitor Utility:

- New MONITOR TRANSACTION command and TRANSACTION class (for use within a DECdtm services environment)
- New MONITOR VECTOR command and VECTOR class (for use within a vector processing environment)

See the *VMS Monitor Utility Manual* for information about other classes and commands.

16.1 MONITOR TRANSACTION Command

The MONITOR TRANSACTION command initiates monitoring of the TRANSACTION class, which shows information about transactions coordinated by the DECdtm services. (For a complete description of DECdtm services, see Chapter 3.)

Use this command as follows:

1 Invoke the MONITOR Utility by entering the DCL command MONITOR. The utility then displays the following prompt:

```
MONITOR>
```

2 At the MONITOR prompt, enter the MONITOR TRANSACTION command. The format, description, and examples of how to use this command follow.

FORMAT

MONITOR TRANSACTION

Qualifiers

/qualifier[,...]
One or more qualifiers, described as follows:

Class-name qualifiers

/ALL
Specifies that a table of all available statistics (current, average, minimum, and maximum) is to be included in the display and summary output. For summary output, this qualifier is the default for all classes; otherwise, it is the default for all classes except CLUSTER, MODES, PROCESSES, STATES, SYSTEM, and VECTOR.

/AVERAGE
Selects average statistics to be displayed in a bar graph for display and summary output.

/CURRENT
Selects current statistics to be displayed in a bar graph for display and summary output. The /CURRENT qualifier is the default for the CLUSTER, MODES, STATES, SYSTEM, and VECTOR classes.

/MAXIMUM
Selects maximum statistics to be displayed in a bar graph for display and summary output.

/MINIMUM
Selects minimum statistics to be displayed in a bar graph for display and summary output.

DESCRIPTION

The TRANSACTION class consists of the following data items:

- Start Rate—The rate at which transactions are started.
- Prepare Rate—The rate at which transactions are placed in the prepare state by DECdtm services.
- One-Phase Commit Rate—The rate that one-phase commit transactions complete using the one-phase commit operation. This operation, which consumes significantly fewer system resources, is used when there is only a single resource manager participating in the transaction.
- Total Commit Rate—The rate at which transactions are committed. This value is the combined total of one-phase and two-phase commit transactions.
- Abort Rate—The rate at which transactions are aborted.
- End Rate—The rate at which transactions are ended.
- Remote Start Rate—The rate at which transactions are started by a transaction manager on a remote node.
- Remote Add Rate—The rate of remote add branch operations.
- Completion Rate—The rate of completed transactions, indexed by their duration time in seconds. Following is a list of the completion rate categories:

Completion Rate 0–1	The number of transactions completed in 0–1 second (1 second or less)
Completion Rate 1–2	The number of transactions completed in 1–2 seconds
Completion Rate 2–3	The number of transactions completed in 2–3 seconds
Completion Rate 3–4	The number of transactions completed in 3–4 seconds

Completion Rate 4–5	The number of transactions completed in 4–5 seconds
Completion Rate 5+	The number of transactions that took more than 5 seconds to complete

A transaction completed in 0.5 second is included in the count displayed for the Completion Rate 0-1 category, which indicates the number of transactions completed in the last time interval that took 0–1 second to execute. See the example displays that follow.

Examples

1

```
MONITOR> MONITOR TRANSACTION/ALL

                              VAX/VMS Monitor Utility
                         DISTRIBUTED TRANSACTION STATISTICS
                                  on node SAMPLE
                               16-JAN-1990 14:52:34

                                     CUR        AVE        MIN        MAX

    Start Rate                     34.76      34.76      34.76      34.76
    Prepare Rate                   33.77      33.77      33.77      33.77
    One Phase Commit Rate           0.00       0.00       0.00       0.00
    Total Commit Rate              35.09      35.09      35.09      35.09
    Abort Rate                      0.00       0.00       0.00       0.00
    End Rate                       35.09      35.09      35.09      35.09
    Remote Start Rate              31.12      31.12      31.12      31.12
    Remote Add Rate                31.45      31.45      31.45      31.45

    Completion Rate    0-1         35.09      35.09      35.09      35.09
     by Duration       1-2          0.00       0.00       0.00       0.00
     in Seconds        2-3          0.00       0.00       0.00       0.00
                       3-4          0.00       0.00       0.00       0.00
                       4-5          0.00       0.00       0.00       0.00
                        5+          0.00       0.00       0.00       0.00
```

This example shows the status of all transactions on node SAMPLE.

2 `MONITOR> MONITOR TRANSACTION/MAXIMUM`

```
                                   VAX/VMS Monitor Utility
                +-----+     DISTRIBUTED TRANSACTION STATISTICS
                | MAX |              on node SAMPLE
                +-----+          16-JAN-1990 14:51:04
                                          0         25        50        75       100
                                          + - - - - + - - - - + - - - - + - - - - -+
 Start Rate                           35  |**************
 Prepare Rate                         37  |**************
 One Phase Commit Rate                    |
 Total Commit Rate                    35  |**************
 Abort Rate                               |
 End Rate                             35  |**************
 Remote Start Rate                    33  |*************
 Remote Add Rate                      32  |************
                                          |         |         |         |          |
 Completion Rate     0-1              35  |**************
  by Duration        1-2                  |
  in Seconds         2-3                  |
                     3-4                  |
                     4-5                  |
                      5+                  |
                                          + - - - - + - - - - + - - - - + - - - - -+
```

This example shows the maximum statistics of all transactions on node SAMPLE.

16.2 TRANSACTION Class Record

The TRANSACTION class record contains data describing the operations of the DECdtm transaction manager. The TRANSACTION class has a record type of 22 and a size of 69 bytes. Figure 16–1 illustrates the format of a TRANSACTION class record; Table 16–1 describes the contents of each of its fields.

Figure 16–1 TRANSACTION Class Record Format

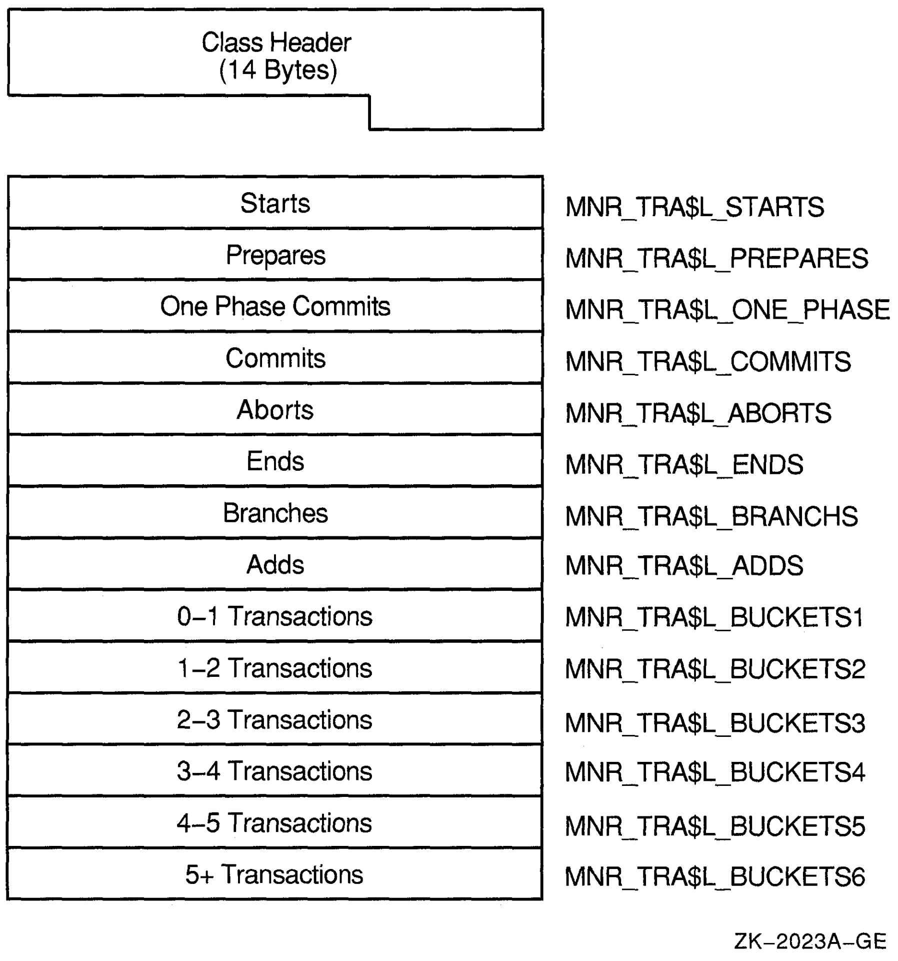


Table 16–1 Descriptions of TRANSACTION Class Record Fields

Field	Symbolic Offset	Contents
Starts	MNR_TRA$L_STARTS	Count of transaction operations started. The number of times the system service $START_TRANS has been successfully completed (longword, C).
Prepares	MNR_TRA$L_PREPARES	Count of transactions that have been prepared (longword, C).
One Phase Commits	MNR_TRA$L_ONE_PHASE	Count of one-phase commit events initiated (longword, C).
Commits	MNR_TRA$L_COMMITS	Count of transactions committed. This is the combined total of one-phase and two-phase commits (longword, C).
Aborts	MNR_TRA$L_ABORTS	Count of transactions aborted. Combined total of planned and unplanned aborts (longword, C).
Ends	MNR_TRA$L_ENDS	Count of transactions ended. The number of times the $END_TRANS has successfully completed (longword, C).
Branches	MNR_TRA$L_BRANCHS	Count of start remote (to a remote parent) branch operations (longword, C).
Adds	MNR_TRA$L_ADDS	Count of add remote (to a remote subordinate parent) branch operations (longword, C).
0-1 Transactions	MNR_TRA$L_BUCKETS1	Count of transactions with a duration of less than 1 second (longword, C).
1-2 Transactions	MNR_TRA$L_BUCKETS2	Count of transactions with a duration of 1 to 2 (1.99) seconds (longword, C).
2-3 Transactions	MNR_TRA$L_BUCKETS3	Count of transactions with a duration of 2 to 3 seconds (longword, C).
3-4 Transactions	MNR_TRA$L_BUCKETS4	Count of transactions with a duration of 3 to 4 seconds (longword, C).
4-5 Transactions	MNR_TRA$L_BUCKETS5	Count of transactions with a duration of 4 to 5 seconds (longword, C).
5+ Transactions	MNR_TRA$L_BUCKETS6	Count of transactions with a duration greater than 5 seconds (longword, C).

16.3 MONITOR VECTOR Command

The MONITOR VECTOR command displays the number of 10-millisecond clock ticks per second in which one or more vector consumers have been scheduled on each currently-configured vector processor in the system. Because the VMS operating system schedules vector consumers only on those processors identified as "vector present," the VECTOR class output never displays vector CPU time for those processors that are "vector absent."

Note that, because vector consumers can use either or both the vector CPU and scalar CPU components of a vector-present processor, the vector CPU time in the VECTOR class display is not a strict measure of the actual usage of the processor's vector CPU component. Rather, it indicates the time during which a scheduled vector consumer has reserved both vector CPU and scalar CPU components of the vector-present processor for its own exclusive use. (For a more complete description of the vector processing environment, see Chapter 2.)

Use this command as follows:

1 Invoke the MONITOR Utility by entering the DCL command MONITOR. The utility then displays the following prompt:

```
MONITOR>
```

2 At the MONITOR prompt, enter the MONITOR VECTOR command. The format, description, and an example of this command follow.

FORMAT MONITOR VECTOR

Qualifiers

/qualifier[,...]
One or more qualifiers, described as follows:

Class-name qualifiers

/ALL
Specifies that a table of all available statistics (current, average, minimum, and maximum) is to be included in the display and summary output. For summary output, this qualifier is the default for all classes; otherwise, it is the default for all classes except CLUSTER, MODES, PROCESSES, STATES, SYSTEM, and VECTOR.

/AVERAGE
Selects average statistics to be displayed in a bar graph for display and summary output.

/CURRENT
Selects current statistics to be displayed in a bar graph for display and summary output. The /CURRENT qualifier is the default for the CLUSTER, MODES, STATES, SYSTEM, and VECTOR classes.

/MAXIMUM
Selects maximum statistics to be displayed in a bar graph for display and summary output.

/MINIMUM
Selects minimum statistics to be displayed in a bar graph for display and summary output.

DESCRIPTION

The VECTOR class consists of the data item Vector Scheduled Rate, which is represented by a display of statistics that show the rates of 10-millisecond clock ticks per second during which vector consumers have been scheduled on each vector-present CPU.

Example

```
MONITOR>   MONITOR VECTOR

                           VAX/VMS Monitor Utility
                         VECTOR PROCESSOR STATISTICS
              +-----+          on node SAMPLE
              | CUR |       12-JUN-1991  22:52:42
              +-----+
 Vector Consumers Scheduled        0         25        50        75       100
                                   + - - - - + - - - - + - - - - + - - - - -+
 Vector Present CPU ID  0        13|*****
 Vector Absent  CPU ID  1          |
 Vector Absent  CPU ID  2          |
 Vector Present CPU ID  4        58|*********************
                                   |         |         |         |          |
                                   |         |         |         |          |
                                   |         |         |         |          |
                                   |         |         |         |          |
                                   |         |         |         |          |
                                   |         |         |         |          |
                                   + - - - - + - - - - + - - - - + - - - - -+
```

This example shows the VECTOR class display for a multiprocessing system containing two vector-present processors, CPU 0 and CPU 4. Displayed statistics represent rates of 10-millisecond clock ticks per second. For an average of 13 ticks per second over the last collection interval, vector consumers have been scheduled on CPU 0. For an average of 58 ticks per second over the last collection interval, vector consumers have been scheduled on CPU

16.4 VECTOR Class Record

The VECTOR class record contains data describing the time during which vector consumers have been scheduled on a vector-present processor. Its record type number is 23. A VECTOR class record is of variable length and depends on the number of active processors in the system. Assuming all processors are active, MONITOR calculates its size by adding the size of the class header and the data block, as follows:

```
13 + (5 * MNR_SYI$B_VPCPUS)
```

Figure 16–2 illustrates the format of a VECTOR class record; Table 16–2 describes the contents of each of its fields.

Figure 16–2 VECTOR Class Record Format

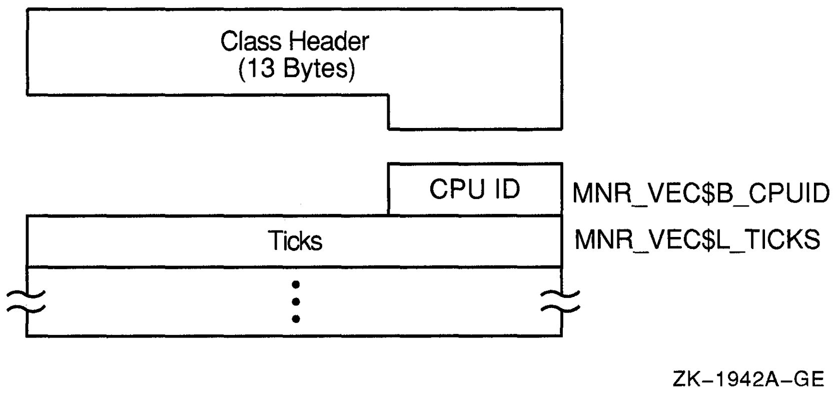


Table 16–2 Descriptions of VECTOR Class Record Fields

Field	Symbolic Offset	Contents
CPU ID	MNR_VEC$B_CPUID	Identification of the processor from which the data has been collected (1 byte).
Ticks	MNR_VEC$L_TICKS	Number of 10-millisecond clock ticks in which a vector consumer has been scheduled on this processor (1 longword).

To support the VECTOR class, MONITOR appends the records in Table 16–3 to the system information record.

Table 16–3 Descriptions of Additions to System Record Fields

Field	Symbolic Offset	Contents
VPCPUs	MNR_SYI$B_VPCPUS	Number of vector-present processors in the current system (1 byte).
VP Conf	MNR_SYI$L_VPCONF	Bit mask identifying those processors in the configuration that are vector-present processors (1 longword).

17 Network Control Program Utility (NCP)

This chapter describes new NCP line and circuit name support for VAXft 3000 systems and for two new Ethernet/820 controllers. See the *VMS Version 5.4 Release Notes* for more information about these and other hardware components that are new or enhanced for Version 5.4 of the VMS operating system.

17.1 Line and Circuit Name Support for VAXft 3000 Systems

The VMS Network Control Program Utility (NCP) supports the following new line and circuit name for VAXft 3000 systems (the controller number can be 0 or a positive number):

KFE-<controller number>

When you enter NCP commands from a VAXft 3000 system connected to your DECnet–VAX network, the KFE-*n* line and circuit name is displayed, as follows:

```
$ RUN SYS$SYSTEM:NCP
NCP> SHOW KNOW LINE

Line Volatile Summary as of 31-AUG-1990 12:50:03

Line                         State

KFE-0                        on

$ RUN SYS$SYSTEM:NCP
NCP> SHOW KNOW LINE

Circuit Volatile Summary as of 31-AUG-1990 12:52:03

                                         Loopback      Adjacent
Circuit                   State          Name        Routing Node

KFE-0                     on                         8.999 (JUPE)
```

17.2 Line and Circuit Names for New Ethernet/820 Controllers

The VMS Network Control Program Utility (NCP) now supports new line and circuit names for the following Ethernet/820 controllers. (See the *VMS Version 5.4 Release Notes* for a complete description of each new controller.)

- DEMNA controller—The NCP line and circuit name for the DEMNA controller is as follows:

  MNA-<controller number>

  For example:

  ```
  MNA-0 (for EXAn)
  MNA-1 (for EXBn)
  ```

- Second Generation Ethernet Controller (SGEC)—The NCP line and circuit name for the SGEC is as follows:

  ISA-<controller number>

  For example:

  ```
  ISA-0 (for EZAn)
  ISA-1 (for EZBn)
  ```

18 VMS Volume Shadowing Phase II

Volume shadowing is the process of maintaining multiple copies of the same data on two or more disk volumes. This duplication of data provides greater data availability and faster data accessibility. Volume shadowing provides high availability by insuring against data loss resulting from media deterioration or through controller or device failure. When data is recorded on more than one disk volume, you have access to critical data even when one volume is unavailable. Disk input/output operations continue with the remaining members of the shadow set.

The system can also find data more quickly because it can search more than one disk. Because a shadow set is made up of multiple disks containing the same data, the shadow set can use the additional read heads to respond to multiple read requests at the same time. In addition, when normal media deterioration renders sections of a volume unreadable, systems with volume shadowing can read the duplicate data and copy it to the failing volume to repair data.

Before Version 5.4, the VMS operating system supported only phase I volume shadowing (see the *VAX Volume Shadowing Manual*). This type of shadowing provides centralized shadowing using HSC controllers with compatible DSA disks. Phase I shadowing is limited to CI configurations on a single system or a VAXcluster.

VMS Volume Shadowing phase II supports the following:

- Clusterwide shadowing of all MSCP-compliant DSA disks having the same physical geometry (having the same number of logical blocks) on a single system or located anywhere in a VAXcluster system.

  Volume shadowing phase II supports clusterwide shadowing of all DSA devices. Phase II is not limited to HSC-controlled disks but extends volume shadowing capabilities to all DSA disks including local adapters, all DSSI (RF series) disk devices on any VAX computer, all interfaces (including but not limited to the KFQSA interface), and across MSCP servers.

- Distributed, not centralized, shadowing

  Volume shadowing phase II creates and maintains virtual units in a distributed fashion on each node in the cluster. Phase II supports shadowing on a single system or in a VAXcluster system where interprocessor communication is carried out over a computer interconnect (CI), Digital small systems interconnect (DSSI), mixed-interconnect configuration, or the Ethernet. Thus, volume shadowing provides fault tolerance resulting from disk media errors across the full range of VAX processors and configurations.

- Shadowing of the system disk and Files–11 On-Disk Structure Level 2 (ODS2) data disks

- Shadowing capabilities across different controllers.

Shadow set member units can be located on different controllers and VMS MSCP servers.

- Shadowing capabilities with mixed phases

  It is possible to use both phase I and phase II shadowing on the same node at the same time. You can also mix phase I and phase II shadowing in a VAXcluster system.

See the new *VMS Volume Shadowing Manual* for complete information about phase II volume shadowing.

Part 4: Programming Features

This part contains the following chapters:

19 VMS Debugger

This chapter describes enhancements to the VMS Debugger for Version 5.4 of the VMS operating system. These enhancements, which let you debug vectorized programs (programs that use VAX vector instructions), are described in more detail in the revised *VMS Debugger Manual*.

19.1 Debugging Vectorized Programs

You can now perform the following debugging tasks using either the debugger's command interface or the DECwindows interface:

- Display information about the availability and use of the vector processor on your system.
- Control and monitor the execution of vector instructions with breakpoints, watchpoints, and so on.
- Specify built-in symbols for the vector registers (%V0 through %V15) and the vector control registers (%VCR, %VLR, and %VMR).
- Examine the values contained in the vector registers and in the vector control registers; deposit values into those registers.
- Display vector instructions using a screen-mode instruction display.
- Examine and deposit vector instructions and their operands.
- Perform masked operations on vector registers to display only certain register elements or override the masking associated with a vector instruction.
- Using the EXAMINE command, specify composite address expressions of a complex form, such as what might be appropriate for a vectorized program. (Note that this feature is not restricted to vectorized programs.)
- Display the decoded results of vector floating-point exceptions.
- Control synchronization between the scalar and vector processors.
- Save and restore the current vector state when using the CALL command to execute a routine that might affect the vector state.
- Display vector register data using a screen-mode display.

19.2 Command Interface: New and Enhanced Commands and Qualifiers

The following list identifies new and enhanced commands and qualifiers for the debugger's command interface:

- CALL/[NO]SAVE_VECTOR_STATE
- CANCEL BREAK/VECTOR_INSTRUCTION

- CANCEL TRACE/VECTOR-INSTRUCTION
- EXAMINE/FMASK, /TMASK, /OPERANDS
- SET BREAK/VECTOR_INSTRUCTION, /INSTRUCTION[=(*opcode[,...]*)]
- SET STEP VECTOR_INSTRUCTION, INSTRUCTION[=(*opcode[,...]*)]
- SET TRACE/VECTOR_INSTRUCTION, /INSTRUCTION[=(*opcode[,...]*)]
- SET VECTOR_MODE [NO]SYNCHRONIZED
- SHOW PROCESS
- SHOW VECTOR_MODE
- STEP/VECTOR_INSTRUCTION, /INSTRUCTION[=(*opcode[,...]*)]
- SYNCHRONIZE VECTOR_MODE

See the *VMS Debugger Manual* for complete information about these commands and qualifiers.

19.3 DECwindows Interface: Enhancements to Menus and Dialog Boxes

The following list identifies new features for the debugger's DECwindows interface:

- The Control menu has a new menu item labeled *Synchronize Vector Processor*. This item is the DECwindows equivalent of the command SYNCHRONIZE VECTOR_MODE.
- In the Step dialog box, the target (upper right) option menu has a new entry labeled *the next vector instruction*.
- In the Break dialog box, the target (upper right) option menu has a new entry labeled *every vector instruction*.
- The Call dialog box has a new toggle button labeled *Save Vector State*.
- The Examine Variable dialog box has a new option menu labeled *Mask*. This menu has three options labeled *None*, *Tmask*, and *Fmask*.
- The Examine Code dialog box has a new option menu labeled *With Operand Values*. This menu has 7 options labeled *None*, *Brief*, *Full*, *Brief with Tmask*, *Brief with Fmask*, *Full with Tmask*, and *Full with Fmask*.
- The Examine Address or Register dialog box has a new option menu labeled *Mask*. This menu has three options labeled *None*, *Tmask*, and *Fmask*.
- The Other Attributes dialog box has a new toggle button labeled *Scalar-Vector Synchronization*. This button is equivalent to the command SET VECTOR_MODE [NO]SYNCHRONIZED.

See the online help that is available from the debugger's DECwindows interface for complete information about these features.

20 Linker Utility (LINK)

With Version 5.4 of the VMS operating system, you can now specify larger page sizes by using the new /BPAGE qualifier with the LINK command.

/BPAGE affects the allocation of image sections. Because image sections must be allocated on a page boundary, specifying a larger page size causes the origin of image sections to be increased to the next multiple of that size. /BPAGE affects only the construction of the image (shareable or executable), not the linker itself or any page-size dependencies in the linked program. An image linked to a larger page size generally runs correctly on a current VMS system, but it might consume more virtual address space. In addition, linking a shareable image to a larger page size can cause the value of transfer vector offsets to change if they were not allocated in page 0 of the image.

The format for using /BPAGE is as follows:

```
LINK [/BPAGE [=n]]
```

If you do not specify /BPAGE with the LINK command, the default 512-byte page is used.

If you specify /BPAGE without a page size value (*n* is not specified), the image is built using 8-kilobyte pages.

If you specify a page size value (*n* is specified), the image is built using that value for the page size. Page size can be any size from 512 bytes to 65 kilobytes. The value is specified as the exponent of a power of 2. For example, when *n* equals 9, the page size is 512 bytes; when *n* equals 13, the page size is 8 kilobytes.

21 Utility Routines: MAIL

The callable interface to the VMS Mail Utility (MAIL) lets you create applications that perform various Mail Utility functions, such as sending mail messages to users on your system. In addition, your applications can communicate with users on remote nodes connected to your system with DECnet–VAX.

Table 21–1 summarizes these new Mail Utility routines. See the revised *VMS Utility Routines Manual* for complete information.

Table 21–1 Mail Utility Routines

Routine	Description
Mail File Context	
MAIL$MAILFILE_BEGIN	Initiates mail file processing
MAIL$MAILFILE_CLOSE	Closes a mail file
MAIL$MAILFILE_COMPRESS	Compresses a mail file
MAIL$MAILFILE_END	Terminates mail file processing
MAIL$MAILFILE_INFO_FILE	Obtains information about the mail file
MAIL$MAILFILE_MODIFY	Changes the wastebasket folder name
MAIL$MAILFILE_OPEN	Opens a mail file
MAIL$MAILFILE_PURGE_WASTE	Purges a mail file
Message Context	
MAIL$MESSAGE_BEGIN	Initiates message processing
MAIL$MESSAGE_COPY	Copies messages
MAIL$MESSAGE_DELETE	Deletes messages
MAIL$MESSAGE_END	Terminates message processing
MAIL$MESSAGE_GET	Retrieves a message
MAIL$MESSAGE_INFO	Obtains information about a specified message
MAIL$MESSAGE_MODIFY	Identfies a message as replied, new, or marked
MAIL$MESSAGE_SELECT	Selects a message or messages from the currently open mail file

(continued on next page)

Table 21–1 (Cont.) Mail Utility Routines

Routine	Description
Send Context	
MAIL$SEND_ABORT	Aborts a send operation
MAIL$SEND_ADD_ADDRESS	Adds an addressee to the address list
MAIL$SEND_ADD_ATTRIBUTE	Constructs the message header
MAIL$SEND_ADD_BODYPART	Constructs the body of the message
MAIL$SEND_BEGIN	Initiates send processing
MAIL$SEND_END	Terminates send processing
MAIL$SEND_MESSAGE	Sends a message
User Context	
MAIL$USER_BEGIN	Initiates user profile context
MAIL$USER_DELETE_INFO	Deletes a user profile entry
MAIL$USER_END	Terminates user profile context
MAIL$USER_GET_INFO	Retrieves information about a user from the user profile
MAIL$USER_SET_INFO	Adds or modifies a user profile entry

22 System Services

The VMS Version 5.4 operating system includes new and modified system services that support the following.

- DECdtm services
- Volume initialization
- System security enhancements
- Vector processing
- VMS Volume shadowing

The information in this chapter is organized as follows:

- Section 22.1 lists and briefly describes system services that are new for Version 5.4 of the VMS operating system.
- Section 22.2 describes how to use specific transaction management services within the DECdtm environment.
- Section 22.3 describes how to use the $INIT_VOL system service.
- Section 22.4 includes the complete format and description of each new system service.
- Section 22.5 describes modifications to existing system services (for example, the addition of new item codes and flags).
- Section 22.6 contains new information about using system services to create site-specific loadable images.

22.1 Summary of New System Services

Table 22–1 lists the system services that are new for the VMS Version 5.4 operating system.

Table 22–1 New VMS Version 5.4 System Services

System Service	Description	Function
	DECdtm Services	
$ABORT_TRANS	Abort Transaction	Aborts a transaction asynchronously; can be called before the transaction is committed
$ABORT_TRANSW	Abort Transaction and Wait	Synchronous equivalent of $ABORT_ TRANS

(continued on next page)

Table 22–1 (Cont.) New VMS Version 5.4 System Services

System Service	Description	Function
	DECdtm Services	
$END_TRANS	End Transaction	Commits a transaction asynchronously
$END_TRANSW	End Transaction and Wait	Synchronous equivalent of $END_TRANS
$START_TRANS	Start Transaction	Starts a transaction (asynchronously) by allocating a transaction identifier (TID) and establishing the internal structures that define a transaction
$START_TRANSW	Start Transaction and Wait	Synchronous equivalent of $START_TRANS
	Volume Initialization	
$INIT_VOL	Initialize Volume	Formats a disk or magnetic tape volume and writes a label on the volume
	System Security	
$FORMAT_AUDIT	Format Security Audit Event Message	Converts a security auditing event message from binary format to ASCII text and filters information considered too sensitive to display
$HASH_PASSWORD	Hash Password	Applies a hash algorithm to an ASCII password string and returns a quadword hash value that represents the encrypted password
	Vector Processing	
$RELEASE_VP	Release Vector Processor	Terminates the current process's status as a vector consumer
$RESTORE_VP_EXCEPTION	Restore Vector Processor Exception State	Restores the saved exception state of the vector processor
$RESTORE_VP_STATE	Restore Vector State	Allows an AST routine or condition handler to restore the vector state of the mainline routine
$SAVE_VP_EXCEPTION	Save Vector Processor Exception State	Saves the pending exception state of the vector processor

22.2 Using Transaction Management System Services

Application programs can call the VMS transaction management system services to delimit the set of operations comprising a distributed transaction. These system services can then guarantee consistent execution of the transaction.

Chapter 3 provides a complete description of DECdtm services and discusses the concept of atomicity in distributed applications.

The transaction management system services provided by the DECdtm services include the following:

- 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)

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.

22.2.1 Transaction Processing System Model

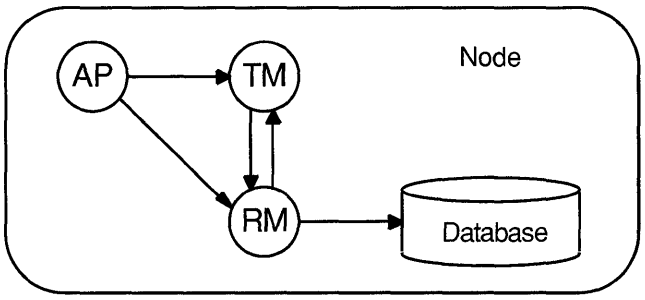
In Digital's model for transaction processing, several components work together to execute atomic transactions.

At the end-user level, user-written application programs define the task to be accomplished, such as query, update, and insertion. An *application program* (AP) also specifies 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 22–1.

Figure 22–1 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 applications 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.

22.2.2 Transaction Management

The responsibilities of a transaction manager include the following:

- Delimit transactions
- Track participating transaction managers and resource managers
- Ensure that transactions either commit or abort
- Assist 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 more or 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.

22.2.3 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/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))
    .
    .
    .
```

22.2.4 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 commit a transaction are $END_TRANS and its synchronous equivalent, $END_TRANSW. The services that abort a transaction are $ABORT_TRANS and its synchronous equivalent, $ABORT_TRANSW. Upon receiving one of these calls, the DECdtm services inform all participants to commit or abort.

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
```

22.2.5 Calling a Planned Abort

$ABORT_TRANS enables 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.

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.
```

22.2.6 Example of Using Transaction Management System Services

Example 22–1 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 22–1 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];

    !+
    ! Start a nondefault process transaction
    !-

❶     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);

    !+
    ! Commit the transaction
    !-
```

(continued on next page)

Example 22–1 (Cont.) Using Transaction Management Services

```
❷     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 NOT .STATUS THEN RETURN (.STATUS);

    !+
    ! Start another nondefault process transaction
    !-
❸     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
    !-
❹     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];

    RETURN (.STATUS);

END;

END
ELUDOM
```

❶ A call to $START_TRANS. The DECdtm transaction manager responds to this call by creating a transaction identifier.

❷ To commit the transaction, the application calls $END_TRANS.

❸ To start another transaction, the application makes another call to $START_TRANS.

❹ To abort the transaction, the application calls $ABORT_TRANS.

22.3 Using the Initialize Volume ($INIT_VOL) System Service

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.

Normally, you initialize a volume from the DCL command stream by using the INITIALIZE command. However, you can also use the Initialize Volume ($INIT_VOL) system service to enable a process to initialize a volume from within a program.

When you call the $INIT_VOL system service, you must specify a device name and a new volume name. You can also use the **itmlst** argument of $INIT_VOL to specify options for the initialization. For example, if you want data compaction to be performed, you can specify the INIT$_ COMPACTION item code.

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 Structure Level 2. Files–11 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.*

Examples

The following example illustrates a call to $INIT_VOL from VAX C. The call is equivalent to the following DCL command:

```
INITIALIZE/DENSITY=6250 MUA0: USER01
```

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. The call is equivalent to the following DCL command:

INITIALIZE/DATA_CHECK=READ DJA21: USERVOLUME

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)
```

22.4 Descriptions of New System Services

This section contains complete descriptions of the system services new for Version 5.4 of the VMS operating system.

$ABORT_TRANS Abort Transaction

Aborts a transaction; it can be called before the transaction is committed.

FORMAT

SYS$ABORT_TRANS *[efn] ,[flags] ,iosb ,[astadr] ,[astprm] ,[tid]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

efn

VMS usage:	**ef_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Number of the event flag to be set. The **efn** argument is a longword containing this number; however, $ABORT_TRANS uses only the low-order byte. If you do not specify the **efn**, $ABORT_TRANS uses the default value 0.

flags

VMS usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Flags specifying options for $ABORT_TRANS. The **flags** argument is a longword bit mask that is the logical OR of each bit set, in which each bit corresponds to an option. The $DDTMDEF macro defines a symbolic name for each flag bit.

DDTM$M_SYNC, the only flag currently defined, is described in Table 22–2.

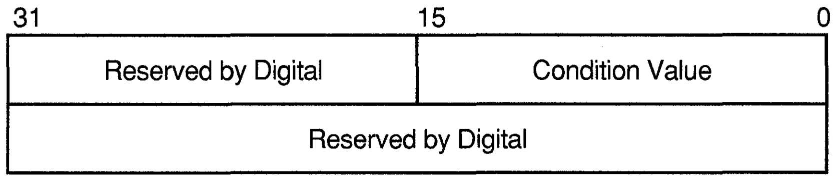
Table 22–2 $ABORT_TRANS Operation Flag

Flag	Description
DDTM$M_SYNC	Indicates successful synchronous completions by returning SS$_SYNCH. When synchronous completion is successful, the completion AST address is not called, the IOSB is not written, and the event flag is not set.

iosb

VMS usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**
I/O status block (IOSB) to receive the final completion status of the request. The **iosb** argument is the address of the quadword I/O status block.

The following diagram shows the structure of the I/O status block:



ZK–1224A–GE

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**
AST service routine to be executed. The **astadr** argument is the address of the entry mask of this routine. In the case of synchronous completion, the call might not take place. Refer to the description of DDTM$M_SYNC in Table 22–2.

If you specify **astadr**, the AST routine executes at the same access mode as the caller of the $ABORT_TRANS service.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**
AST parameter passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is a longword.

tid

VMS usage: **transaction_id**
type: **octaword (unsigned)**
access: **read only**
mechanism: **by reference**
Pointer to the transaction identifier (TID) structure that designates the transaction to be aborted. The default value for this parameter is the current transaction.

DESCRIPTION

The Abort Transaction service aborts a specific transaction by invalidating the transaction identifier (TID) and instructing all resource managers involved to nullify all the actions of the transaction. This system service can be called by an application program or by any resource manager or transaction manager participating in the execution of the transaction.

For a single-node transaction, $ABORT_TRANS can be successfully called any time before the transaction is committed. A transaction is considered to be committed when the commit record is written to the transaction log file. A committed transaction executes in its entirety. When a transaction is aborted, the transaction manager orders all participants in the transaction to roll back any changes made to database files. Thus, none of the intended actions of the distributed transactions is made permanent.

For distributed transactions, $ABORT_TRANS can be successfully called from the coordinating (home) node any time before the transaction is committed and from the participating (remote) node only until the participant transaction manager is prepared.

Required Privileges

None.

Required Quota

ASTLM

Related Services

$END_TRANS, $START_TRANS

CONDITION VALUES RETURNED

SS$_NORMAL	The operation was successfully queued.
SS$_SYNCH	The synchronous operation completed successfully.
SS$_ACCVIO	The IOSB or TID cannot be read by the caller, or the IOSB cannot be written by the caller.
SS$_BADPARAM	The operations flags are invalid.
SS$_EXASTLM	The process has exceeded its AST limit quota.
SS$_ILLEFC	The **efn** argument specifies an illegal flag number.

SS$_INSFMEM	There is insufficient system dynamic memory for the operation.
SS$_NOCURTID	No default TID is defined.
SS$_NOSUCHTID	The designated TID is unknown.
SS$_WRONGSTATE	The transaction is in the wrong state for the attempted operation.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Same as those returned in R0. A value of SS$_NORMAL returned in the I/O status block indicates that the service completed successfully.

$ABORT_TRANSW Abort Transaction and Wait

Aborts a transaction; it can be called before the transaction is committed.

$ABORT_TRANSW completes synchronously; that is, it returns to the caller after the request has completed.

For asynchronous completion, use the Abort Transaction ($ABORT_TRANS) service, which aborts a transaction without waiting for the operation to complete.

In all other respects, $ABORT_TRANSW is identical to $ABORT_TRANS. For all other information about the $ABORT_TRANSW service, refer to the section on $ABORT_TRANS.

For additional information about system service completion, refer to the Synchronize ($SYNCH) service and to *Introduction to VMS System Services*.

FORMAT

SYS$ABORT_TRANSW *[efn] ,[flags] ,iosb ,[astadr] ,[astprm] ,[tid]*

$END_TRANS End Transaction

Commits a transaction.

FORMAT

SYS$END_TRANS *[efn] ,[flags] ,iosb ,[astadr] ,[astprm] ,[tid]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

efn

VMS usage:	**ef_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Number of the event flag to be set. The **efn** argument is a longword containing this number; however, $END_TRANS uses only the low-order byte. If you do not specify **efn**, $END_TRANS uses the default value 0.

flags

VMS usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Flags specifying options for $END_TRANS. The **flags** argument is a longword bit mask that is the logical OR of each bit set, in which each bit corresponds to an option. The $DDTMDEF macro defines a symbolic name for each flag bit.

DDTM$M_SYNC, the only flag currently defined, is described in Table 22–3.

Table 22–3 $END_TRANS Operation Flag

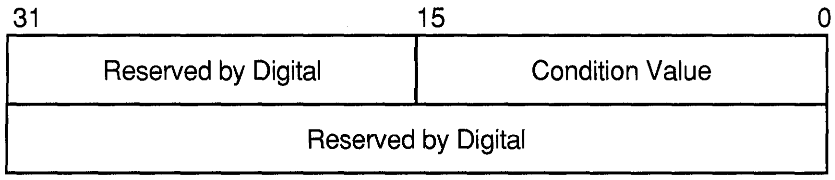
Flag	Description
DDTM$M_SYNC	Indicates successful synchronous completions by returning SS$_SYNCH. When synchronous completion is successful, the completion AST address is not called, the IOSB is not written, and the event flag is not set.

iosb

VMS usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

I/O status block (IOSB) to receive the final completion status of the request. The **iosb** argument is the address of the quadword I/O status block.

The following diagram shows the structure of the I/O status block:



ZK–1224A–GE

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**

AST service routine to be executed. The **astadr** argument is the address of the entry mask of this routine. In the case of synchronous completion, the call might not take place. Refer to the description of DDTM$M_SYNC in Table 22–3.

If you specify **astadr**, the AST routine executes at the same access mode as the caller of the $END_TRANS service.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is a longword.

tid

VMS usage: **transaction_id**
type: **octaword (unsigned)**
access: **read only**
mechanism: **by reference**
Pointer to the transaction identifier (TID) structure that designates the transaction to be committed. The default value for this parameter is the current transaction.

DESCRIPTION

The End Transaction service instructs the DECdtm services to commit a transaction. When $END_TRANS is called, the transaction manager component of the DECdtm services implements the two-phase commit protocol to inform all the transaction's participants (any resource managers and transaction managers involved in the transaction) to commit.

The first phase of the two-phase commit protocol is termed the prepare phase. In the prepare phase, the transaction manager sends a prepare message to all participants. The prepare message requests each participating resource manager to vote on its ability to complete the transaction processing actions. The transaction manager waits to receive the results of the prepare message.

The participants must notify the transaction manager whether they have succeeded or failed in performing their transaction processing work. If a participating resource manager is able to prepare, it sends a "yes" vote to the transaction manager. If the resource manager is unable to prepare, it casts a "no" vote. If the resource manager fails before replying, the transaction manager assumes a "no" vote.

When all participants have responded to the prepare request, the transaction manager proceeds to the second phase, the commit phase.

If all of the participants have successfully completed the prepare phase and voted "yes," the transaction manager orders the participants to commit the transaction. Each participating resource manager completes commit processing for its transaction by writing a commit log record to its local transaction log file. A distributed transaction is complete when all its actions, such as changes to databases, are made permanent.

If an application calls $ABORT_TRANS or $ABORT_TRANSW or if any of the participants have failed to prepare successfully, the transaction is aborted. For example, a resource manager might fail to prepare successfully due to a process failure, machine failure, or hardware failure. In the abort phase, the transaction manager orders all remaining participants to abort the transaction and roll back their transaction processing work. Thus, none of the actions of the distributed transaction is made permanent.

$END_TRANS returns a failure status (SS$_ABORT) if the prepare phase does not complete successfully or if an error occurs that makes it impossible to commit the transaction.

Required Privileges

None.

Required Quota

ASTLM

Related Services

$ABORT_TRANS, START_TRANS

CONDITION VALUES RETURNED

SS$_NORMAL	The operation was successfully queued.
SS$_SYNCH	The synchronous operation completed successfully.
SS$_ABORT	The transaction aborted during processing.
SS$_ACCVIO	The IOSB or TID cannot be read by the caller, or the IOSB cannot be written by the caller.
SS$_BADPARAM	The operations flags are invalid.
SS$_EXASTLM	The process has exceeded its AST limit quota.
SS$_ILLEFC	The **efn** argument specifies an illegal flag number.
SS$_INSFMEM	There is insufficient system dynamic memory for the operation.
SS$_NOCURTID	No default TID is defined.
SS$_NOSUCHTID	The designated TID is unknown.
SS$_WRONGSTATE	The transaction is in the wrong state for the attempted operation.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Same as those returned in R0. A value of SS$_NORMAL returned in the I/O status block indicates that the service completed successfully.

$END_TRANSW End Transaction and Wait

Commits a given transaction. It returns a failure status (SS$_ABORT) if an error occurs that makes it impossible the transaction to be committed.

$END_TRANSW completes synchronously; that is, it returns to the caller after the request has actually completed.

For asynchronous completion, you use the End Transaction ($END_TRANS) system service, which commits a transaction and allocates a transaction identifier without waiting for the operation to complete.

In all other respects, $END_TRANSW is identical to $END_TRANS. For all other information about $END_TRANSW, refer to the section on $END_TRANS.

For additional information about system service completion, refer to the Synchronize ($SYNCH) service and to *Introduction to VMS System Services*.

FORMAT

SYS$END_TRANSW *[efn] ,[flags] ,iosb ,[astadr] ,[astprm] ,[tid]*

$FORMAT_AUDIT Format Security Audit Event Message

Converts a security auditing event message from binary format to ASCII text and filters information the user considers too sensitive to display.

FORMAT

SYS$FORMAT_AUDIT *[fmttyp] ,audmsg ,[outlen] ,[outbuf] ,[width] ,[trmdsc] ,[routin] [,fmtflg]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

fmttyp

VMS usage:	**longword_unsigned**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Format for the message. The **fmttyp** argument is a value indicating whether the security audit message should be in brief format, which is one line of information, or full format. The default is full format. See the *VMS Audit Analysis Utility Manual* for examples of formatted output.

Value	Meaning
NSA$C_FORMAT_STYLE_BRIEF	Use a brief format for the message.
NSA$C_FORMAT_STYLE_FULL	Use a full format for the message.

audmsg

VMS usage:	**char_string**
type:	**character-coded text string**
access:	**read only**
mechanism:	**by descriptor**

Security auditing message to format. The **audmsg** argument is the address of a character descriptor pointing to a buffer containing the binary message that requires formatting.

outlen

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

Length of the formatted security audit message. The **outlen** argument is the address of the word receiving the final length of the ASCII message.

outbuf

VMS usage:	**char_string**
type:	**character-coded text string**
access:	**read only**
mechanism:	**by descriptor**

Buffer holding the formatted message. The **outbuf** argument is the address of a descriptor pointing to the buffer receiving the message.

width

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**read only**
mechanism:	**by reference**

Maximum width of the formatted message. The **width** argument is the address of a word containing the line width value. The default is 80 columns.

trmdsc

VMS usage:	**char_string**
type:	**character-coded text string**
access:	**read only**
mechanism:	**by descriptor**

Line termination characters used in a full format message. The **trmdsc** argument is the address of a descriptor pointing to the line termination characters to insert within a line segment whenever the **width** is reached.

routin

VMS usage:	**longword_unsigned**
type:	**procedure**
access:	**read only**
mechanism:	**by reference**

Routine that writes a formatted line to the output buffer. The **routin** argument is the address of a routine called each time a line segment is formatted. The argument passed to the routine is the address of a character string descriptor for the line segment.

When an application wants event messages in the brief format, $FORMAT_AUDIT calls the routine twice to format the first event message. The first time it is called, the routine passes a string containing the column titles for the message. The second and subsequent calls to the routine pass the formatted event message. By using this routine argument, a caller can gain control at various points in the processing of an audit event message.

fmtflg

VMS usage:	**longword (unsigned)**
type:	**mask_longword**
access:	**read only**
mechanism:	**by value**

Determines the formatting of certain kinds of audit messages. The **fmtflg** argument is a mask specifying whether sensitive information, such as passwords, should be displayed or column titles built for messages in brief format. The following table describes the significant bits:

Bit	Value	Description
0	1	Do not format sensitive information, for example, passwords.
	0	Format sensitive information.
1	1	Build a column title for messages in brief format. (You must specify a **fmttyp** of brief and a **routin** argument.)
	0	Do not build column titles.

DESCRIPTION

The Format Audit service converts a security auditing event message from binary format to ASCII text and can filter information—for example, passwords. $FORMAT_AUDIT allows the caller to format a message in a multiple line format or a single line format and tailor the information for a display device of a specific width.

$FORMAT_AUDIT is intended for utilities that need to format the security auditing event messages received from the audit server listener mailbox or the system security audit log file.

Required Privileges

None.

Required Quota

$FORMAT_AUDIT can cause a process to exceed the paging file limit (PGFLQUOTA) if it has to format a long auditing event message. The caller of $FORMAT_AUDIT can also receive quota violations from services that $FORMAT_AUDIT uses, such as $IDTOASC, $FAO, and $GETMSG.

Related Services

None.

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully.
SS$_MSGNOTFND	The service completed successfully; however, the message code cannot be found and a default message has been returned.

SS$_ACCVIO	The item list cannot be read by the caller, or the buffer length or buffer cannot be written by the caller.
SS$_BADPARAM	The item list contains an invalid identifier.
SS$_BUFFEROVF	The service completed successfully; however, the formatted output string overflowed the output buffer and has been truncated.
SS$_INSFMEM	The process dynamic memory is insufficient for opening the rights database.
SS$_IVCHAN	The contents of the context longword are not valid.
SS$_IVIDENT	The specified identifier is of invalid format.
SS$_NOSUCHID	The specified identifier name does not exist in the rights database.

Because the rights database is an indexed file that you access with VMS RMS, this service can also return RMS status codes associated with operations on indexed files. For descriptions of these status codes, refer to the *VMS Record Management Services Manual*.

$HASH_PASSWORD Hash Password

Applies the hash algorithm you select to an ASCII password string and returns a quadword hash value that represents the encrypted password.

FORMAT

SYS$HASH_PASSWORD
pwd ,alg ,[salt] ,usrnam ,hash

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

pwd

VMS usage:	**char_string**
type:	**character-coded text string**
access:	**read only**
mechanism:	**by descriptor—fixed-length string descriptor**

ASCII password string to be encrypted. The **pwd** argument is the address of a character string descriptor pointing to the ASCII password. The password string can contain between 1 and 32 characters and use the uppercase characters A through Z, the numbers 0 through 9, the dollar sign ($), and the underscore (_).

alg

VMS usage:	**byte_unsigned**
type:	**byte (unsigned)**
access:	**read only**
mechanism:	**by value**

Algorithm used to hash the ASCII password string. The **alg** argument is an unsigned byte specifying the hash algorithm. The VMS operating system recognizes the following algorithms:

Symbolic Name	Description
UAI$C_AD_II	Uses a CRC algorithm and returns a longword hash value. This algorithm was used in releases prior to VMS Version 2.0.
UAI$C_PURDY	Uses a Purdy algorithm over salted input. It expects a blank-padded user name and returns a quadword hash value. This algorithm was used during VMS Version 2.0 field test.

Symbolic Name	Description
UAI$C_PURDY_V	Uses the Purdy algorithm over salted input. It expects a variable length user name and returns a quadword hash value. This algorithm was used in releases prior to VMS Version 5.4.
UAI$C_PURDY_S	Uses the Purdy algorithm over salted input. It expects a variable length user name and returns a quadword hash value. This algorithm is used to hash all new passwords in VMS Version 5.4 and later.
UAI$C_PREFERED_ ALGORITHM[1]	Represents the latest encryption algorithm that the VMS system uses to encrypt new passwords. Currently, it equates to UAI$C_PURDY_S. Digital recommends that you use this symbol in source modules because it always equates with the most recent VMS algorithm.

[1] The value of this symbol might be changed in future releases if an additional algorithm is introduced.

Values ranging from 128 to 255 are reserved for customer use; the constant UAI$K_CUST_ALGORITHM defines the start of this range.

You can use the UAI$_ENCRYPT and UAI$_ENCRYPT2 item codes with the $GETUAI system service to retrieve the primary and secondary password hash algorithms for a user.

salt

VMS usage: **word_unsigned**
type: **word (unsigned)**
access: **read only**
mechanism: **by value**

Value used to increase the effectiveness of the hash. The **salt** argument is an unsigned word containing 16 bits of data that is used by the hash algorithms when encrypting a password for the associated user name. The $GETUAI item code UAI$_SALT is used to retrieve the SALT value for a given user. If you do not specify a SALT value, $HASH_PASSWORD uses the value of 0.

usrnam

VMS usage: **char_string**
type: **character-coded text string**
access: **read only**
mechanism: **by descriptor—fixed-length string descriptor**

Name of the user associated with the password. The **usrnam** argument is the address of a descriptor pointing to a character text string containing the user name. The current VMS password encryption algorithm (UAI$K_ PURDY_S) folds the user name into the ASCII password string to ensure that different users with the same password produce different hash values. This argument must be supplied for all calls to $HASH_PASSWORD but is ignored when using the CRC algorithm (UAI$K_AD_II).

hash

VMS usage: **quadword_unsigned**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**

Output hash value representing the encrypted password. The **hash** argument is the address of an unsigned quadword to which $HASH_PASSWORD writes the output of the hash. If you use the UAI$C_AD_II algorithm, the second longword of the hash is always set to zero.

DESCRIPTION

Applies the hash algorithm you select to an ASCII password string and returns a quadword hash value that represents the encrypted password.

Required Privileges

None.

Required Quota

None.

Related Services

$GETUAI and $SETUAI. Use $GETUAI to get the values for the **salt** and **alg** arguments. Use $SETUAI to store the resulting hash using the item codes UAI$_PWD and UAI$_PWD2.

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The input or output buffer descriptors cannot be read or written to by the caller.
SS$_BADPARAM	The specified hash algorithm is unknown or invalid.

$INIT_VOL Initialize Volume

Formats a disk or magnetic tape volume and writes a label on the volume. At the end of initialization, the disk is empty except for the system files containing the structure information. All former contents of the volume are lost.

FORMAT

SYS$INIT_VOL *devnam, volnam [,itmlst]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

devnam

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Name of the device on which the volume is physically mounted. The descriptor must point to the device name, a character string of 1 to 64 characters. The device name can be a physical device name or a logical name; if it is a logical name, it must translate to a physical name.

The device does not have to be currently allocated; however, allocating the device before initializing it is recommended.

volnam

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

Identification to be encoded on the volume. The descriptor must point to the volume name, a character string of 1 to 12 characters. For a disk volume name, you can specify a maximum of 12 alphanumeric characters; for a magnetic tape volume name, you can specify a maximum of 6 ANSI "a" characters. Any valid ANSI "a" characters can be used; these include numbers, uppercase letters, and any one of the following nonalphanumeric characters:

! " % ' ( ) * + , - . / : ; < = >

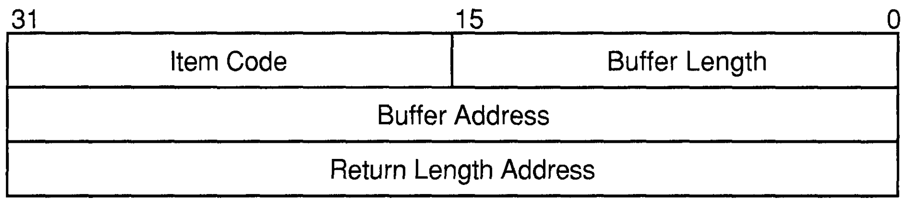
Nonalphanumeric characters are not allowed in the volume name on disk.

itmlst

VMS usage:	**item_list_3**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Item list specifying options that can be used when initializing the volume. The **itmlst** argument is the address of a list of item descriptors, each of which describes one option. The list of item descriptors is terminated by a longword of 0.

The following diagram depicts the format of a single item descriptor:



ZK−1705−GE

$INIT_VOL Item Descriptor Fields

buffer length
A word specifying the length, in bytes, of the buffer that supplies the information $INIT_VOL needs to process the specified item code. The length of the buffer needed depends upon the item code specified in the item descriptor.

item code
A word containing an option for the initialize operation. These codes are defined by the $INITDEF macro.

There are three types of item codes:

- Boolean item code. Boolean item codes specify a true or false value. The form INIT$_*code* specifies a true value and the form INIT$_NO_*code* specifies a false value. For Boolean item codes, the **buffer length** and **buffer address** fields of the item descriptor must be zero.
- Symbolic value item code. Symbolic value item codes specify one of a specified range of possible choices. The **buffer length** and **buffer address** fields of the item descriptor must be zero.
- Input value item code. Input value item codes specify a value to be used by $INIT_VOL. The **buffer length** and **buffer address** fields of the item descriptor must be nonzero.

Each item code is described after the argument descriptions.

buffer address
A longword containing the address of the buffer that supplies information to $INIT_VOL.

return length address
This field is not used.

item codes

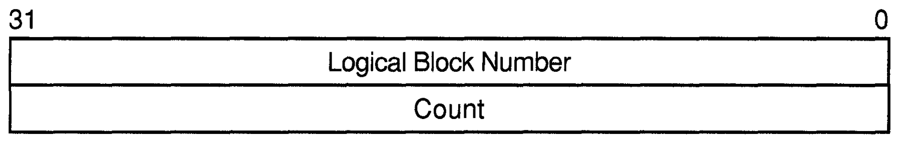
INIT$_ACCESSED
An input item code that specifies the number of directories allowed in system space on the volume.

You must specify an integer between 0 and 255 in the input buffer. The default value is 3.

The INIT$_ACCESSED item code applies only to Files–11 Structure Level 1 disks.

INIT$_BADBLOCKS_LBN
An input item code that enables $INIT_VOL to mark bad blocks on the volume; no data is written to those faulty areas. INIT$_BADBLOCKS_LBN specifies faulty areas on the volume by logical block number and block count.

The buffer from which $INIT_VOL reads the option information contains an array of quadwords containing information in the following format:



ZK–1590A–GE

The following table describes the information to be specified for INIT$_BADBLOCKS_LBN:

Field	Symbol Name	Description
Logical block number	INIT$L_BADBLOCKS_LBN	Specifies the logical block number of the first block to be marked as allocated
Count	INIT$L_BADBLOCKS_COUNT	Specifies the number of blocks to be allocated. This range begins with the first block, as specified in INIT$L_BADBLOCKS_LBN

For example, if the input buffer contains the values 5 and 3, INIT_VOL starts at logical block number 5 and allocates 3 blocks.

The number of entries in the buffer is determined by the **buffer length** field in the item descriptor.

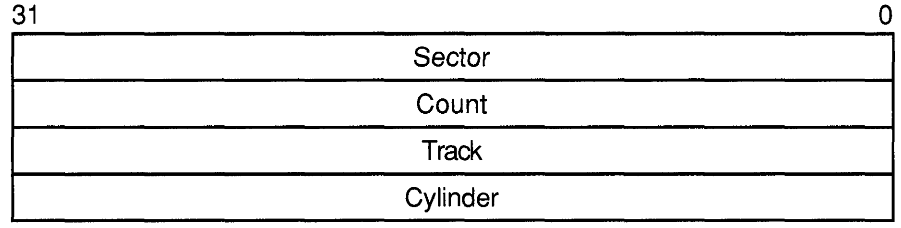
All media supplied by Digital and supported on the VMS operating system, except floppy disks and TU58 cartridges, are factory formatted and contain bad block data. The Bad Block Locator Utility (BAD) or the diagnostic formatter EVRAC can be used to refresh the bad block data or to construct it for the floppy disks and TU58 cartridges.

The INIT$_BADBLOCKS_LBN item code is necessary only to enter bad blocks that are not identified in the volume's bad block data. For more information, see the *VMS Bad Block Locator Utility Manual.*

The INIT$_BADBLOCKS_LBN item code applies only to disks.

INIT$_BADBLOCKS_SEC
An input item code that specifies faulty areas on the volume by sector, track, cylinder, and block count. $INIT_VOL marks the bad blocks as allocated; no data is written to them.

The input buffer must contain an array of octawords containing information in the following format:



ZK−1591A−GE

The following table describes the information to be specified for INIT$_ BADBLOCKS_LBN:

Field	Symbol Name	Description
Sector	INIT$L_BADBLOCKS_ SECTOR	Specifies the sector number of the first block to be marked as allocated
Count	INIT$L_BADBLOCKS_ COUNT	Specifies the number of blocks to be allocated
Track	INIT$L_BADBLOCKS_ TRACK	Specifies the track number of the first block to be marked as allocated
Cylinder	INIT$L_BADBLOCKS_ CYLINDER	Specifies the cylinder number of the first block to be marked as allocated

For example, if the input buffer contains the values 12, 3, 1 and 2, INIT_ VOL starts at sector 12, track 1, cylinder 2 and allocates 3 blocks.

The number of entries in the buffer is determined by the **buffer length** field in the item descriptor.

All media supplied by Digital and supported on the VMS operating system, except floppy disks and TU58 cartridges, are factory formatted and contain bad block data. The Bad Block Locator Utility (BAD) or the diagnostic formatter EVRAC can be used to refresh the bad block data or to construct it for the floppy disks and TU58 cartridges. The INIT$_BADBLOCKS_ SEC item code is necessary only to enter bad blocks that are not identified in the volume's bad block data. For more information, see the *VMS Bad Block Locator Utility Manual.*

The INIT$_BADBLOCKS_SEC item code applies only to disks.

INIT$_CLUSTERSIZE
An input item code that specifies the minimum allocation unit in blocks. The input buffer must contain a longword value. The maximum size that can be specified for a volume is one-hundredth the size of the volume; the minimum size is calculated with the following formula:

$$\frac{\textit{volume size in blocks}}{255 * 4096}$$

The INIT$_CLUSTERSIZE item code applies only to Files–11 Structure Level 2 disks (for Files–11 Structure Level 1 disks, the cluster size is 1). For Files–11 Structure Level 2 disks, the cluster size default depends on the disk capacity.

- Disks that are 50,000 blocks or larger have a default cluster size of 3.
- Disks smaller than 50,000 blocks have a default value of 1.

INIT$_COMPACTION
INIT$_NO_COMPACTION—Default
A Boolean item code that specifies whether data compaction should be performed when writing the volume.

The INIT$_COMPACTION item code applies only to TA90 drives.

INIT$_DENSITY
A symbolic item code that specifies the density value for magnetic tapes and diskettes.

For magnetic tape volumes, the INIT$_DENSITY item code specifies the density in bytes per inch (bpi) at which the magnetic tape is written. Possible symbolic values for tapes are as follows:

- INIT$K_DENSITY_800_BPI
- INIT$K_DENSITY_1600_BPI
- INIT$K_DENSITY_6250_BPI

The specified density value must be supported by the drive. If you do not specify a density item code for a blank magnetic tape, the system uses a default density of the highest value allowed by the tape drive. If the drive allows 6250, 1600, and 800 bpi operation, the default density is 6250. If the drive allows only 1600 and 800 bpi operation, the default density is 1600. If you do not specify a density item code for a magnetic tape that has been previously written, the system uses the previously set volume density.

For diskettes, the INIT$_DENSITY item code specifies how the diskette is to be formatted. Possible symbolic values for diskettes are as follows:

- INIT$K_DENSITY_SINGLE_DISK
- INIT$K_DENSITY_DOUBLE_DISK
- INIT$K_DENSITY_DD_DISK
- INIT$K_DENSITY_HD_DISK

For floppy disk volumes that are to be initialized on RX02, RX23 or RX33 diskette drives, the following values specify how the floppy disk is to be formatted:

- INIT$K_DENSITY_SINGLE_DISK
- INIT$K_DENSITY_DOUBLE_DISK
- INIT$K_DENSITY_DD_DISK
- INIT$K_DENSITY_HD_DISK

Diskettes are initialized as follows:

- RX23 diskettes—DD or HD density.
- RX33 diskettes—double density only.
- RX02 dual-density diskette drives—single or double density.

If you do not specify a density item code for a floppy disk, the system leaves the volume at the density at which it was last formatted. RX02 floppy disks purchased from Digital are formatted in single density.

Note: **Floppy disks formatted in double density cannot be read or written by the console block storage device (an RX01 drive) of a VAX–11/780 processor until they have been reformatted in single density.**

INIT$_DIRECTORIES
An input item code that specifies the number of entries to preallocate for user directories. The input buffer must contain a longword value in the range of 16 to 16000. The default value is 16.

The INIT$_DIRECTORIES item code applies only to disks.

INIT$_ERASE
INIT$_NO_ERASE—Default
A Boolean item code that specifies whether deleted data should be physically destroyed by performing the data security erase (DSE) operation on the volume before initializing it. The INIT$_ERASE item code applies to the following devices:

- ODS-2 disk volumes
- ANSI magnetic tape volumes on magnetic tape devices that support the hardware erase function, for example, TU78 and MSCP magnetic tapes.

For disk devices, this item code sets the ERASE volume attribute, causing each file on the volume to be erased when it is deleted.

INIT$_EXTENSION
An input item code that specifies, by the number of blocks, the default extension size for all files on the volume. The extension default is used when a file increases to a size greater than its initial default allocation during an update. For Files–11 Structure Level 2 disks, the buffer must contain a longword value in the range 0 to 65535. For Files–11 Structure Level 1 disks, the input buffer must contain a longword value in the range

of 0 to 255. The default value is 5 for both Structure Level 1 and Structure Level 2 disks.

The default extension set by this item code is used only if the following conditions are in effect:

- No default extension for the file has been set
- No default extension for the process has been set using the SET RMS command.

INIT$_FPROT
An input item code that specifies the default protection that is applied to all files on the volume. The input buffer must contain a longword protection mask that contains four 4-bit fields. Each field grants or denies read, write, execute, and delete access to a category of users. Cleared bits grant access; set bits deny access. The following diagram depicts the structure of the protection mask:

World				Group				Owner				System			
D	E	W	R	D	E	W	R	D	E	W	R	D	E	W	R
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ZK-1592A-GE

The INIT$_FPROT item code applies only to Files–11 Structure Level 1 disks and is ignored if it is used on a VMS system. VMS systems use the default file extension set by the DCL command SET PROTECTION /DEFAULT.

INIT$_HEADERS
An input item code that specifies the number of file headers to be allocated for the index file. The input buffer must contain a longword value within the range of 16 to the value set by the INIT$_MAXFILES item code. The default value is 16.

The INIT$_HEADERS item code applies only to disks.

INIT$_HIGHWATER—Default
INIT$_NO_HIGHWATER
A Boolean item code that sets the file highwater mark (FHM) volume attribute, which guarantees that a user cannot read data that he or she has not written.

INIT$_NO_HIGHWATER disables FHM for a volume.

The INIT$_HIGHWATER and INIT$_NO_HIGHWATER item codes apply only to Files–11 Structure Level 2 disks.

INIT$_INDEX_BEGINNING
A symbolic item code that places the index file for the volume's directory structure at the beginning of the volume. By default, the index is placed in the middle of the volume.

This item code applies only to disks.

INIT$_INDEX_BLOCK
An input item code that specifies the location of the index file for the volume's directory structure by logical block number. The input buffer must contain a longword value specifying the logical block number of the first block of the index file. By default, the index is placed in the middle of the volume.

The INIT$_INDEX_BLOCK item code applies only to disks.

INIT$_INDEX_END
A symbolic item code that places the index file for the volume's directory structure at the end of the volume. The default is to place the index in the middle of the volume.

This item code applies only to disks.

INIT$_INDEX_MIDDLE
A symbolic item code that places the index file for the volume's directory structure in the middle of the volume. This is the default location for the index.

This item code applies only to disks.

INIT$_LABEL_ACCESS
An input item code that specifies the character to be written in the volume accessibility field of the VMS ANSI volume label VOL1 on an ANSI magnetic tape. Any valid ANSI "a" characters can be used; these include numbers, uppercase letters, and any one of the following nonalphanumeric characters:

! " % ' ( ) * + , - . / : ; < = >

By default, the VMS operating system provides a routine SYS$MTACCESS that checks this field in the following manner:

- If the magnetic tape was created on a version of the VMS operating system that conforms to Version 3 of ANSI, this item code is used to override any character except an ASCII space.
- If the magnetic tape conforms to an ANSI standard that is later than Version 3, this item code is used to override any character except an ASCII 1 character.

INIT$_LABEL_VOLO
An input item code that specifies the text that is written in the owner identifier field of the VMS ANSI volume label VOL1 on an ANSI magnetic tape. The owner identifier field can contain up to 14 valid ANSI "a" characters.

INIT$_MAXFILES
An input item code that restricts the maximum number of files that the volume can contain. The input buffer must contain a longword value between 0 and a value determined by the following calculation:

$$\frac{volume\ size\ in\ blocks}{cluster\ factor + 1}$$

Once initialized, the maximum number of files can be increased only by reinitializing the volume.

The default maximum number of files is calculated as follows:

$$\frac{volume\ size\ in\ blocks}{(cluster\ factor + 1) * 2}$$

The INIT$_MAXFILES item code applies only to disks.

INIT$_OVR_ACCESS
INIT$_NO_OVR_ACCESS—Default
A Boolean item code that specifies whether to override any character in the accessibility field of the VMS ANSI volume label VOL1 on an ANSI magnetic tape. For more information, see the *Guide to VMS Files and Devices*.

To specify INIT$_OVR_ACCESS, the caller must either own the volume or have VOLPRO privilege.

INIT$_OVR_EXP
INIT$_NO_OVR_EXP—Default
A Boolean item code that specifies whether the caller writes to a magnetic tape that has not yet reached its expiration date. This item code only applies to the magnetic tapes that were created before VMS Version 4.0 and that use the D% format in the volume owner identifier field.

To specify INIT$_OVR_EXP, the caller must either own the volume or have VOLPRO privilege.

INIT$_OVR_VOLO
INIT$_NO_OVR_VOLO—Default
A Boolean item code that allows the caller to override processing of the owner identifier field of the VMS ANSI volume label VOL1 on an ANSI magnetic tape.

To specify INIT$_OVR_VOLO, the caller must either own the volume or have VOLPRO privilege.

INIT$_OWNER
An input item code that specifies the UIC that will own the volume. The input buffer must contain a longword value, which is the UIC. The default is the UIC of the caller.

For magnetic tapes, no UIC is written unless protection on the magnetic tape is specified. If the INIT$_VPROT item code is specified but the INIT$_OWNER item code is not specified, the UIC of the caller is assigned ownership of the volume.

INIT$_READCHECK
INIT$_NO_READCHECK—Default
A Boolean item code that specifies whether data checking should be performed for all read operations on the volume. For more information about data checking, see the *VMS I/O User's Reference Manual: Part I.*

The INIT$_READCHECK item code applies only to disks.

INIT$_SIZE
An input item code that specifies the number of blocks allocated for a RAM disk with a device type of DT$_RAM_DISK. The input buffer must contain a longword value.

INIT$_STRUCTURE_LEVEL_1
INIT$_STRUCTURE_LEVEL_2—Default
Symbolic item codes that specify whether the volume should be formatted in Files–11 Structure Level 1 or Structure Level 2. Structure Level 1 is incompatible with the following item codes:

- INIT$_READCHECK
- INIT$_WRITECHECK
- INIT$_CLUSTERSIZE

The default protection for a Structure Level 1 disk is full access to system, owner, and group users, and read access to all other users.

The INIT$_STRUCTURE_LEVEL_1 item code applies only to disks.

INIT$_USER_NAME
An input item code that specifies the user name that is associated with the volume. The input buffer must contain a character string from 1 to 12 alphanumeric characters, which is the user name. The default is the user name of the caller.

INIT$_VERIFIED
INIT$_NO_VERIFIED
A Boolean item code that indicates whether the disk contains bad block data. INIT$_NO_VERIFIED indicates that any bad block data on the disk should be ignored. For disks with 4096 blocks or more, the default is INIT$_VERIFIED.

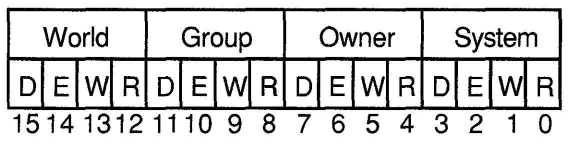
INIT$_NO_VERIFIED is the default for the following:

- Disks with less than 4096 blocks
- Digital Storage Architecture (DSA) devices
- Disks which are not last-track devices

The INIT$_VERIFIED item codes apply only to disks.

INIT$_VPROT
An input item code that specifies the protection that is assigned to the volume. The input buffer must contain a longword protection mask that contains four 4-bit fields. Each field grants or denies read, write, execute, and delete access to a category of users. Cleared bits grant access; set bits deny access.

The following diagram depicts the structure of the protection mask:



ZK-1592A-GE

The default is the default protection of the caller.

For magnetic tape, the protection code is written to a VMS-specific volume label. The system only applies read and write access restrictions; execute and delete access are ignored. Moreover, the system and the owner are always given read and write access to magnetic tapes, regardless of the protection mask specified.

When you specify a protection mask for a disk volume, access type E (execute) indicates **Create Access.**

INIT$_WINDOW
The INIT$_WINDOW item code specifies the number of mapping pointers to be allocated for file windows. The input buffer must contain a longword value in the range 7 to 80. The default is 7.

When a file is opened, the file system uses the mapping pointers to access the data in the file.

The INIT$_WINDOW item code applies only to disks.

INIT$_WRITECHECK
INIT$_NO_WRITECHECK—Default
A Boolean item code that specifies whether data checking should be performed for all read operations on the volume. For more information about data checking, see the *VMS I/O User's Reference Manual: Part I.*

INIT$_WRITECHECK item code applies only to disks.

DESCRIPTION

The Initialize Volume system service formats a disk or magnetic tape volume and writes a label on the volume. At the end of initialization, the disk is empty except for the system files containing the structure information. All former contents of the volume are lost.

A blank magnetic tape can sometimes cause unrecoverable errors when it is read. $INIT_VOL attempts to read the volume unless the following three conditions are in effect:

- INIT$_OVR_ACCESS Boolean item code is specified.
- INIT$_OVR_EXP Boolean item code is specified.
- Caller has VOLPRO privilege.

  If the caller has VOLPRO privilege, $INIT_VOL initializes a disk without reading the ownership information. Otherwise, the ownership of the volume is checked.

A blank floppy disk or a diskette with an incorrect format can sometimes cause a fatal drive error. Such a diskette can be initialized successfully by specifying the INIT$_DENSITY item code to format the diskette.

Required Privileges

To initialize a particular volume, the caller must either have volume protection (VOLPRO) privilege or the volume must be one of the following:

- Blank disk or magnetic tape; that is, a volume that has never been written
- Disk that is owned by the caller's UIC or by the UIC [0,0]
- Magnetic tape that allows write access to the caller's UIC or that was not protected when it was initialized

Required Quota

None.

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The item list or an address specified in the item list cannot be accessed.
SS$_BADPARAM	A buffer length of zero was specified with a nonzero item code or an illegal item code was specified.
SS$_IVSSRQ	A concurrent call to SYS$INIT_VOL is already active for the process.
SS$_NOPRIV	The caller does not have sufficient privilege to initialize the volume.
SS$_NOSUCHDEV	The specified device does not exist on the host system.

The $INIT_VOL service can also return the following condition values, which are specific to the Initialize Volume Utility. The symbolic definition macro $INITDEF defines these condition values.

INIT$_ALLOCFAIL	Index file allocation failure.
INIT$_BADACCESSED	Value for INIT$_ACCESSED item code out of range.
INIT$_BADBLOCKS	Invalid syntax in bad block list.
INIT$_BADCLUSTER	Value for INIT$_CLUSTER_SIZE item code out of range.
INIT$_BADDENS	Invalid value for INIT$_DENSITY item code.
INIT$_BADDIRECTORIES	Value for INIT$_DIRECTORIES item code out of range.
INIT$_BADEXTENSION	Value for INIT$_EXTENSION item code out of range.
INIT$_BADHEADERS	Value for INIT$_HEADERS item code out of range.
INIT$_BADMAXFILES	Value for INIT$_MAXFILES item code out of range.

INIT$_BADOWNID	Invalid value for owner ID.
INIT$_BADRANGE	Bad block address not on volume.
INIT$_BADVOL1	Bad VOL1 ANSI label.
INIT$_BADVOLACC	Invalid value for INIT$_LABEL_ACCESS item code.
INIT$_BADVOLLBL	Invalid value for ANSI tape volume label.
INIT$_BADWINDOWS	Value for INIT$_WINDOWS item code out of range.
INIT$_BLKZERO	Block zero is bad—volume not bootable.
INIT$_CLUSTER	Unsuitable cluster factor.
INIT$_CONFQUAL	Conflicting options were specified.
INIT$_DIAGPACK	Disk is a diagnostic pack.
INIT$_ERASEFAIL	Volume not completely erased.
INIT$_FACTBAD	Cannot read factory bad block data.
INIT$_ILLOPT	Item codes not appropriate for the device were specified.
INIT$_INDEX	Invalid index file position.
INIT$_LARGECNT	Disk too large to be supported.
INIT$_MAXBAD	Bad block table overflow.
INIT$_MTLBLLONG	Magnetic tape label specified is longer than 6 characters.
INIT$_MTLBLNONA	Magnetic tape label specified contains non-ANSI "a" characters.
INIT$_NOBADDATA	Bad block data not found on volume.
INIT$_NONLOCAL	Device is not a local device.
INIT$_NOTRAN	Logical name cannot be translated.
INIT$_NOTSTRUC1	Option(s) not available with Files–11 Structure Level 1.
INIT$_UNKDEV	Unknown device type.

$RELEASE_VP Release Vector Processor

Terminates the current process's status as a vector consumer.

FORMAT

SYS$RELEASE_VP

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

None.

DESCRIPTION

The Release Vector Processor service terminates the current process's status as a vector consumer. $RELEASE_VP waits for all pending vector instructions and vector memory operations to complete. It then declares that the process no longer needs a vector-present processor. As a result, the process relinquishes its use of the processor's vector registers and can be scheduled on another processor in the system.

In systems that do not have vector-present processors but do have the VAX vector instruction emulation facility (VVIEF) in use, this service relinquishes the process's use of VVIEF. The VVIEF remains mapped in the process's address space.

Required Privileges

None.

Required Quota

None.

Related Services

$RESTORE_VP_EXCEPTION, $RESTORE_VP_STATE, $SAVE_VP_EXCEPTION

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully.

$RESTORE_VP_EXCEPTION Restore Vector Processor Exception State

Restores the saved exception state of the vector processor.

FORMAT

SYS$RESTORE_VP_EXCEPTION *excid*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service returns are listed in the Condition Values Returned section.

ARGUMENTS

excid

VMS usage:	**context**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Internal ID of the exception state saved by $SAVE_VP_EXCEPTION. The **excid** argument is the address of a longword containing this ID.

DESCRIPTION

The Restore Vector Exception State service restores from memory the vector exception state saved by a prior call to $SAVE_VP_EXCEPTION. After a routine invokes this service, the next vector instruction issued within the process causes the restored vector exception to be reported.

By default, when an AST or condition handler interrupts the execution of a mainline routine, VMS saves the mainline routine's vector state, including its vector exception state. Any other routine that executes synchronously with, or asynchronously to, currently executing vectorized code and that performs vector operations itself must preserve the preempted routine's vector exception state across its own execution. It does so by using the $SAVE_VP_EXCEPTION and $RESTORE_VP_EXCEPTION services.

Used together, these services ensure that vector exceptions occurring as a result of activity in the original routine are serviced by existing condition handlers within that routine.

In systems that do not have vector-present processors but do have the VAX vector instruction emulation facility (VVIEF) in use, VVIEF emulates the function of this service.

Required Privileges

None.

Required Quota

BYTLM

Related Services

$RELEASE_VP, $RESTORE_VP_STATE, $SAVE_VP_EXCEPTION

CONDITION VALUES RETURNED

SS$_NORMAL	The *service completed successfully*. The service also returns this status when executed in a system that does not have vector-present processors and that does not have the VAX vector instruction emulation facility (VVIEF) loaded.
SS$_ACCVIO	The caller cannot read the exception ID longword.
SS$_NOSAVPEXC	No saved vector exception state exists for this exception ID.

$RESTORE_VP_STATE Restore Vector State

Allows an AST routine or condition handler to restore the vector state of the mainline routine.

FORMAT

SYS$RESTORE_VP_STATE

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

None.

DESCRIPTION

The Restore Vector State service allows an AST routine or a condition handler to restore the vector state of the process's mainline routine.

By default, when an asynchronous routine (AST routine or condition handler) interrupts the execution of a mainline routine, VMS creates a new vector state when the routine issues its first vector instruction. At this point, the vector state of the mainline routine is inaccessible to the asynchronous routine. If the asynchronous routine must manipulate the vector state of the mainline routine, it first calls $RESTORE_VP_STATE to restore the mainline's vector state.

In systems that do not have vector-present processors but do have the VAX vector instruction emulation facility (VVIEF) in use, VVIEF emulates the functions of this service.

This service can be called only from a routine running in user mode.

Required Privileges

None.

Required Quota

None.

Related Services

$RELEASE_VP, $RESTORE_VP_EXCEPTION, $SAVE_VP_EXCEPTION

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully. Vector state of the mainline has been restored. The service also returns this status when executed in a system that does not have vector-present processors and that does not have the VAX vector instruction emulation facility (VVIEF) loaded.
SS$_BADSTACK	Bad user stack encountered.
SS$_BADCONTEXT	The mainline vector state is corrupt.
SS$_WRONGACMODE	The system service was called from an access mode other than user mode.

$SAVE_VP_EXCEPTION Save Vector Processor Exception State

Saves the pending exception state of the vector processor.

FORMAT

SYS$SAVE_VP_EXCEPTION *excid*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

excid

VMS usage:	**context**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Internal ID of the exception state saved by $SAVE_VP_EXCEPTION. The **excid** argument is the address of a longword containing this ID.

DESCRIPTION

The Save Vector Exception State service saves in memory any pending vector exception state and clears the vector processor's current exception state.

By default, when an AST or condition handler interrupts the execution of a mainline routine, VMS saves the mainline routine's vector state, including its vector exception state. Any other routine that executes synchronously with, or asynchronously to, currently executing vectorized code and that performs vector operations itself must preserve the preempted routine's vector exception state across its own execution. It does so by using the $SAVE_VP_EXCEPTION and $RESTORE_VP_EXCEPTION services. Used together, these services ensure that vector exceptions occurring as a result of activity in the original routine are serviced by existing condition handlers within that routine.

In systems that do not have vector-present processors but do have the VAX vector instruction emulation facility (VVIEF) in use, VVIEF emulates the functions of this service.

Required Privileges

None.

Required Quota

None.

Related Services

$RELEASE_VP, $RESTORE_VP_EXCEPTION, $RESTORE_VP_STATE

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully. There were no pending vector exceptions. The service also returns this status when executed in a system that does not have vector-present processors and that does not have the VAX vector instruction emulation facility (VVIEF) loaded.
SS$_ACCVIO	The caller cannot write the exception ID longword.
SS$_INSFMEM	Insufficient system dynamic memory exists for completing the service.
SS$_WASSET	The service completed successfully. Pending vector exception state has been saved.

$START_TRANS Start Transaction

Starts a transaction by allocating a transaction identifier (TID) and establishing the internal structures that define a transaction.

FORMAT

SYS$START_TRANS *[efn] ,[flags] ,iosb ,[astadr] ,[astprm] ,tid*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed in the Condition Values Returned section.

ARGUMENTS

efn

VMS usage:	**ef_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Number of the event flag to be set. The **efn** argument is a longword containing this number; however, $START_TRANS uses only the low-order byte. If you do not specify **efn**, $START_TRANS uses the default value 0.

flags

VMS usage:	**mask_longword**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Flags specifying options for $START_TRANS. The **flags** argument is a longword bit mask that is the logical OR of each bit set, in which each bit corresponds to an option.

The $DDTMDEF macro defines a symbolic name for each flag bit. Table 22–4 describes each flag.

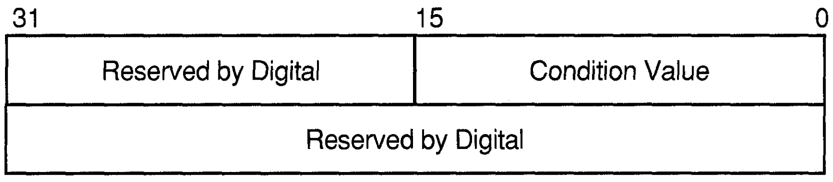
Table 22–4 $START_TRANS Operation Flags

Flag	Description
DDTM$M_NONDEFAULT	Indicates that this transaction is not the process default (current) transaction.
DDTM$M_SYNC	Indicates successful synchronous completions by returning SS$_SYNCH. When synchronous completion is successful, the completion AST address is not called, the IOSB is not written, and the event flag is not set.
DDTM$M_PROCESS	Indicates that the transaction might survive image rundown. Caller must be in supervisor, executive, or kernel mode.

iosb

VMS usage: **io_status_block**
type: **quadword (unsigned)**
access: **write only**
mechanism: **by reference**
I/O status block (IOSB) to receive the final completion status of the request. The **iosb** argument is the address of the quadword I/O status block.

The following diagram shows the structure of the I/O status block:



ZK–1224A–GE

astadr

VMS usage: **ast_procedure**
type: **procedure entry mask**
access: **call without stack unwinding**
mechanism: **by reference**
AST service routine to be executed. The **astadr** argument is the address of the entry mask of this routine.

If you specify **astadr**, the AST routine executes at the same access mode as the caller of the $START_TRANS service.

Note that the completion AST is not called if SS$_SYNCH is returned in R0.

astprm

VMS usage: **user_arg**
type: **longword (unsigned)**
access: **read only**
mechanism: **by value**

AST parameter passed to the AST service routine specified by the **astadr** argument. The **astprm** argument is a longword.

tid

VMS usage:	**transaction_id**
type:	**octaword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Pointer to the transaction identifier (TID).

DESCRIPTION

The Start Transaction service starts a transaction and allocates a unique TID for it.

The DECdtm services maintain the concept of a current transaction for each process. When a transaction is started using $START_TRANS, that transaction is considered the process default or current transaction. The TID assigned by $START_TRANS identifies the current transaction. There cannot be a current transaction already active for a process when you start a new transaction, or an error is returned. The current transaction becomes undefined when the current transaction is ended by $END_TRANS or $ABORT_TRANS. However, it is possible to start a nondefault transaction while the current transaction is in progress by specifying the NONDEFAULT flag. A nondefault transaction is not considered a current transaction.

Required Privileges

None.

Required Quota

$START_TRANS uses the job's buffered byte count quota limit (BYTLM) and AST quota limit (ASTLM).

Related Services

$ABORT_TRANS, $END_TRANS

CONDITION VALUES RETURNED

SS$_NORMAL	The operation was successfully queued.
SS$_SYNCH	The synchronous operation completed successfully.
SS$_ABORT	The transaction aborted during processing.
SS$_ACCVIO	The IOSB or TID cannot be read by the caller, or the TID or IOSB cannot be written by the caller.
SS$_ALRCURTID	An attempt was made to start a default (current) transaction when there was already one started.
SS$_BADPARAM	The operations flags are invalid.
SS$_EXASTLM	The process has exceeded its AST limit quota.
SS$_EXQUOTA	The process quota was exceeded.

SS$_ILLEFC	The **efn** argument specifies an illegal flag number.
SS$_INSFMEM	There is insufficient system dynamic memory for the operation.
SS$_WRONGACMODE	The wrong access mode was specified; a process flag was specified from user mode.

CONDITION VALUES RETURNED IN THE I/O STATUS BLOCK

Same as those returned in R0. A value of SS$_NORMAL returned in the I/O status block indicates that the service completed successfully.

$START_TRANSW Start Transaction and Wait

Starts a transaction. It allocates a transaction identifier and establishes the internal structures that define a transaction.

$START_TRANSW completes synchronously; that is, it returns to the caller after the request has actually completed. For asynchronous completion, you use the Start Transaction ($START_TRANS) service; $START_TRANS starts a transaction and allocates a transaction identifier without waiting for the operation to complete.

In all other respects, $START_TRANSW is identical to $START_TRANS. For all other information about the $START_TRANSW service, refer to the section on $START_TRANS.

For additional information about system service completion, refer to the Synchronize ($SYNCH) service and to the *Introduction to VMS System Services*.

FORMAT

SYS$START_TRANSW *[efn] ,[flags] ,iosb ,[astadr] ,[astprm] ,tid*

22.5 Modified System Services

This section describes system services that have been modified in the VMS Version 5.4 operating system.

22.5.1 $CHANGE_ACL

Modifications to $CHANGE_ACL include a new object type to support vector processing and new item codes to support enhancements to system security, described in the following sections.

22.5.1.1 Vector Processing: New Object Type

To control use of a system's vector processors, $CHANGE_ACL supports a new CAPABILITY object type, which describes a restricted resource. With the VMS Version 5.4 operating system, if the object type is CAPABILITY, use the reserved name VECTOR. The only capability currently defined by the VMS operating system is the VECTOR capability, governing the ability of a subject to access a vector processor in the system. The symbolic name is ACL$C_CAPABILITY.

22.5.1.2 System Security: New Item Codes

In Version 5.4 of the VMS operating system, $CHANGE_ACL accepts the following item codes:

ACL$C_DELETE_ALL
When you specify ACL$C_DELETE_ALL, $CHANGE_ACL deletes the entire Access Control List (ACL), including protected entries.

ACL$C_GRANT_ACE
When you specify ACL$C_GRANT_ACE, $CHANGE_ACL reads the next ACE that matches the process's identifiers into the buffer pointed to by **bufadr**. The returned ACE might grant or deny access to the object. Since an ACL can have more than one matching ACE, you should proceed as follows:

1. Specify an initial value of zero (0) for **contxt**.
2. Call $CHANGE_ACL repeatedly, without changing the value of **contxt**, and test for the return status SS$_NOMOREACE, which means that the ACL has no more matching entries.

ACL$C_NEXT_ACE
When you specify ACL$C_NEXT_ACE, $CHANGE_ACL advances through an ACL, one ACE at a time. The **contxt** argument defines the initial and final positions. The value of **contxt** itself is derived from the previous ACL$C_FNDACETYP, ACL$C_FNDACLENT, or ACL$C_GRANT_ACE operation.

The $CHANGE_ACL service returns the following new status codes:

SS$_NOPRIV	You do not have privileges for the requested action.

SS$_INCONOLCK	VMS encountered an irrecoverable error. Please submit a Software Performance Report (SPR) that describes conditions leading to the error.

22.5.2 $CHECK_ACCESS: Vector Processing and System Security Support

Like $CHANGE_ACL, $CHECK_ACCESS supports the new CAPABILITY object type. Also note that $CHECK_ACCESS requires privilege to access the UAF, for example, SYSPRV.

The $CHECK_ACCESS service returns the following new status codes:

SS$_INSFMEN	Identifiers granted to the user exceed the number allowed.
SS$_NOCALLPRIV	Caller lacks privilege for attempted operation.
SS$_NOSUCHSEC	The specified global section does not exist.
SS$_UNSUPPORTED	Operations on remote object are not supported.

22.5.3 $ENQ: Enhanced Lock Manager Support

The addition of the following new flags to $ENQ enables you to expedite lock requests and force queueing of conversions.

LCK$M_EXPEDITE

This flag is valid only for new lock requests. Specifying this flag allows a request to be granted immediately, provided the requested mode, when granted, would not block any currently queued requests in the resource conversion and wait queues. Currently, this flag is valid only for NLMODE requests. If this flag is specified for any other lock mode, the request fails and an error of SS$_UNSUPPORTED returned.

LCK$M_QUECVT

This flag is valid only for conversion operations. A conversion request with the LCK$M_QUECVT flag set will be forced to wait behind any already queued conversions.

The conversion request is granted immediately, if there are no already queued conversions.

The QUECVT behavior is valid only for a subset of all possible conversions. Table 22–5 defines the legal set of conversion requests for LCK$M_ QUECVT. Illegal conversion requests fail, with SS$_BADPARAM returned.

Table 22–5 Legal QUECVT Conversions

Lock Mode at Which Lock Is Held	Lock Mode to Which Lock Is Converted					
	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

22.5.4 $GETDVI: New Device Classes

In Version 5.4 of the VMS operating system, $GETDVI accepts the three new device classes listed in Table 22–6.

Table 22–6 Values Returned by the DEVCLASS Item

Device Class	Value	Symbolic Name
Workstation	70	DC$_WORKSTATION
DECvoice	97	DC$_DECVOICE
Remote console storage	170	DC$_REMCSL_STORAGE

22.5.5 $GETJPI

The following sections describe new items codes that support vector processing and enhancements to system security.

22.5.5.1 Vector Processing: New Item Codes

In Version 5.4 of the VMS operating system, $GETJPI accepts the following item codes and returns information regarding a process's use of system vector processing resources.

JPI$_FAST_VP_SWITCH

When you specify JPI$_FAST_VP_SWITCH, $GETJPI returns an unsigned longword containing the number of times this process has issued a vector instruction that resulted in an inactive vector processor being enabled without the expense of a vector context switch. In other words, this count reflects those instances where the process has reenabled a vector processor on which the process's vector context has remained intact.

JPI$_SLOW_VP_SWITCH
When you specify JPI$_SLOW_VP_SWITCH, $GETJPI returns an unsigned longword containing the number of times this process has issued a vector instruction that resulted in an inactive vector processor being enabled with a full vector context switch. This vector context switch involves the saving of the vector context of the process that last used the vector processor and the restoration of the vector context of the current process.

JPI$_VP_CONSUMER
When you specify JPI$_VP_CONSUMER, $GETJPI returns a byte, the low-order bit of which, when set, indicates that the process is a vector consumer.

JPI$_VP_CPUTIM
When you specify JPI$_VP_CPUTIM, $GETJPI returns an unsigned longword that contains the total amount of time the process has accumulated as a vector consumer.

22.5.5.2 System Security: New Item Codes

In Version 5.4 of the VMS operating system, $GETJPI accepts the following new item codes and returns information about process security characteristics.

JPI$_RIGHTS_SIZE
When you specify JPI$_RIGHTS_SIZE, $GETJPI returns the number of bytes required to buffer the rights list. The rights list includes both the system rights list and the process rights list. Because the space requirements for the rights list can change between the time you request the size of the rights list and the time you fetch the rights list with JPI$_RIGHTSLIST, you might want to allocate a buffer that is 10% larger.

JPI$_PROCESS_RIGHTS
When you specify JPI$_PROCESS_RIGHTS, $GETJPI returns the binary content of the process rights list as an array of quadword identifiers. Each entry consists of a longword identifier value and longword identifier attributes, shown in Table 22–7. Allocate a buffer that is sufficient to hold the process rights list because $GETJPI only returns as much of the list as will fit in the buffer.

Table 22–7 Attributes of an Identifier

Symbolic Name	Description
KGB$M_RESOURCE	Resources can be charged to the identifier.
KGB$M_DYNAMIC	Identifier can be enabled or disabled.

JPI$_SYSTEM_RIGHTS
When you specify JPI$_SYSTEM_RIGHTS, $GETJPI returns the system rights list as an array of quadword identifiers. Each entry consists of a longword identifier value and longword identifier attributes, shown in Table 22–7. Allocate a buffer that is sufficient to hold the system rights list because $GETJPI only returns as much of the list as will fit in the buffer.

JPI$_RIGHTSLIST
When you specify JPI$_RIGHTSLIST, $GETJPI returns, as an array of quadword identifiers, all identifiers applicable to the process. This includes the process rights list (JPI$_PROCESS_RIGHTS) and the system rights list (JPI$_SYSTEM_RIGHTS). Each entry consists of a longword identifier value and longword identifier attributes, shown in Table 22–7. Allocate a buffer that is sufficient to hold the rights list because $GETJPI only returns as much of the list as will fit in the buffer.

JPI$_LAST_LOGIN_I
When you specify JPI$_LAST_LOGIN_I, $GETJPI returns, as a quadword absolute time value, the date of the last successful interactive login prior to the current session. It returns a quadword of 0 when processes have not executed the LOGINOUT image.

JPI$_LAST_LOGIN_N
When you specify JPI$_LAST_LOGIN_N, $GETJPI returns, as a quadword absolute time value, the date of the last successful noninteractive login prior to the current session. It returns a quadword of 0 when processes have not executed the LOGINOUT image.

JPI$_LOGIN_FAILURES
When you specify JPI$_LOGIN_FAILURES, $GETJPI returns the number of login failures that occurred prior to the current session. It returns a longword of 0 when processes have not executed the LOGINOUT image.

JPI$_LOGIN_FLAGS
When you specify JPI$_LOGIN_FLAGS, $GETJPI returns a longword bitmask containing information related to the login sequence. It returns a longword of 0 when processes have not executed the LOGINOUT image. The following bits are defined:

Symbolic Name	Description
JPI$M_NEW_MAIL_AT_LOGIN	User had new mail messages waiting at login.
JPI$M_PASSWORD_CHANGED	User changed the primary password during login.
JPI$M_PASSWORD_EXPIRED	User's primary password expired during login.
JPI$M_PASSWORD_WARNING	System gave the user a warning at login that the account's primary password would expire within 5 days.
JPI$M_PASSWORD2_CHANGED	Account's secondary password was changed during login.
JPI$M_PASSWORD2_EXPIRED	Account's secondary password expired during login.
JPI$M_PASSWORD2_WARNING	System gave the user a warning at login that the account's secondary password would expire within 5 days.

22.5.6 $GETSYI

The following sections describe new items codes that support vector processing and enhancements to system security.

22.5.6.1 Vector Processing: New Item Codes

In Version 5.4 of the VMS operating system, $GETSYI accepts the following item codes and returns information regarding the system's vector processing configuration.

SYI$_VP_MASK
When you specify SYI$_VP_MASK, $GETSYI returns a longword mask, the bits of which, when set, indicate which processors in the system have vector coprocessors.

SYI$_VP_NUMBER
When you specify SYI$_VP_NUMBER, $GETSYI returns an unsigned longword containing the number of vector processors in the system.

SYI$_VECTOR_EMULATOR
When you specify SYI$_VECTOR_EMULATOR, $GETSYI returns a byte, the low-order bit of which, when set, indicates the presence of the VAX vector instruction emulator facility (VVIEF) in the system.

22.5.6.2 System Security: New Item Code

In Version 5.4 of the VMS operating system, $GETSYI accepts the following item code and returns information about system security.

SYI$_SYSTEM_RIGHTS
When you specify SYI$_SYSTEM_RIGHTS, $GETSYI returns the system rights list as an array of quadword identifiers. Each entry consists of a longword identifier value and the following longword identifier attributes:

Symbolic Name	Description
KGB$M_RESOURCE	Resources can be charged to the identifier.
KGB$M_DYNAMIC	Identifier can be enabled or disabled.

Allocate a buffer that is sufficient to hold the system rights list because $GETSYI only returns as much of the list as will fit in the buffer.

22.5.7 $GETUAI: New Item Codes for Enhanced Password Screening

With Version 5.4 of the VMS operating system, passwords selected by users can be screened for acceptability. The VMS system automatically compares new passwords against a system dictionary to ensure that a password is not a native language word. It also maintains a history list of a user's passwords and compares each new password against this list to guarantee that an old password is not reused.

In addition, a site with contractual obligations to use special algorithms for encrypting passwords will be able to use them.

To support these enhancements, $GETUAI accepts the following item codes and returns information about passwords and logins:

UAI$_ENCRYPT
When you specify UAI$_ENCRYPT, $GETUAI returns one of the values shown in the following table, identifying the encryption algorithm for the primary password.

Because the encryption algorithm is a byte in length, the buffer length field in the item descriptor should specify 1 byte.

Symbolic Name	Description
UAI$C_AD_II	Uses a CRC algorithm and returns a longword hash value. It was used in VMS releases prior to Version 2.0.
UAI$C_PURDY	Uses a Purdy algorithm over salted input. It expects a blank-padded user name and returns a quadword hash value. This algorithm was used during VMS Version 2.0 field test.
UAI$C_PURDY_V	Uses the Purdy algorithm over salted input. It expects a variable length user name and returns a quadword hash value. This algorithm was used in VMS releases prior to Version 5.4.
UAI$C_PURDY_S	Uses the Purdy algorithm over salted input. It expects a variable length user name and returns a quadword hash value. This is the current algorithm that VMS uses for all new password changes.

UAI$_ENCRYPT2
When you specify UAI$_ENCRYPT2, $GETUAI returns one of the following values identifying the encryption algorithm for the secondary password. Refer to the UAI$_ENCRYPT item code for a description of the algorithms.

— UAI$C_AD_II

— UAI$C_PURDY

— UAI$C_PURDY_V

— UAI$C_PURDY_S

Because the encryption algorithm is a byte in length, the buffer length field in the item descriptor should specify 1 byte.

UAI$_FLAGS
This item code has the following new symbolic names:

Symbolic Name	Description
UAI$V_DISPWDDIC	Automatic checking of user-selected passwords against the system dictionary is disabled.
UAI$V_DISPWDHIS	Automatic checking of user-selected passwords against previously used passwords is disabled.

22.5.8 $MOD_IDENT: New Status Code

In Version 5.4 of the VMS operating system, $MOD_IDENT may return the following status code:

SS$_DUPIDENT	The specified identifier value already exists.

22.5.9 $MOUNT: Volume Shadowing Flags

In Version 5.4 of the VMS operating system, $MOUNT accepts the following new flags:

MNT$V_INCLUDE	Applicable only if you have the VMS Volume Shadowing option.
MNT$V_NOCOPY	Applicable only if you have the VMS Volume Shadowing option.
MNT$V_OVR_SHAMEM	Applicable only if you have the VMS Volume Shadowing option.

For more information about volume shadowing, see Chapter 18 (summary of phase II support) and the *VMS Volume Shadowing Manual* (detailed information).

22.5.10 $SETUAI: New Item Codes for Enhanced Password Screening

With Version 5.4 of the VMS operating system, passwords selected by users can be screened for acceptability. The VMS system automatically compares new passwords against a system dictionary to ensure that a password is not a native language word. It also maintains a history list of a user's passwords and compares each new password against this list to guarantee that an old password is not reused.

In addition, a site with contractual obligations to use special algorithms for encrypting passwords will be able to use them.

To support these enhancements, $SETUAI accepts the following item codes and returns information about encryption algorithms for passwords:

UAI$_ENCRYPT
When you specify UAI$_ENCRYPT, $SETUAI sets one of the values shown in the following table to identify the encryption algorithm for the primary password.

Symbolic Name	Description
UAI$C_AD_II	Uses a CRC algorithm and returns a longword hash value. It was used in VMS releases prior to Version 2.0.
UAI$C_PURDY	Uses a Purdy algorithm over salted input. It expects a blank-padded user name and returns a quadword hash value. This algorithm was used during VMS Version 2.0 field test.

Symbolic Name	Description
UAI$C_PURDY_V	Uses the Purdy algorithm over salted input. It expects a variable length user name and returns a quadword hash value. This algorithm was used in VMS releases prior to Version 5.4.
UAI$C_PURDY_S	Uses the Purdy algorithm over salted input. It expects a variable length user name and returns a quadword hash value. This is the current algorithm that VMS uses for all new password changes.
UAI$C_PREFERED_ ALGORITHM[1]	Represents the latest encryption algorithm that the VMS system uses to encrypt new passwords. Currently, it equates to UAI$C_PURDY_S. Digital recommends that you use this symbol in source modules.

[1] The value of this symbol may be changed in future releases if an additional algorithm is introduced.

Because the encryption algorithm is a byte in length, the buffer length field in the item descriptor should specify 1 byte.

UAI$_ENCRYPT2
When you specify UAI$_ENCRYPT2, $SETUAI sets one of the following values, indicating the encryption algorithm for the secondary password. Refer to the UAI$_ENCRYPT item code for a description of the algorithms.

- UAI$C_AD_II
- UAI$C_PURDY
- UAI$C_PURDY_V
- UAI$C_PURDY_S
- UAI$C_PREFERED_ALGORITHM

Because the encryption algorithm is a byte in length, the buffer length field in the item descriptor should specify 1 byte.

UAI$_FLAGS
This item code has two new symbolic names:

Symbolic Name	Description
UAI$V_DISPWDDIC	Automatic checking of user-selected passwords against the system dictionary is disabled.
UAI$V_DISPWDHIS	Automatic checking of user-selected passwords against previously used passwords is disabled.

UAI$_SALT
When you specify UAI$_SALT, $SETUAI sets the salt field of the user's record to the value you provide. The salt value is used in the VMS hash algorithm to generate passwords. $SETUAI does not generate a new salt value for you.

Because this decimal number is a word in length, the buffer length field in the item descriptor should specify 2 bytes.

By copying the item codes UAI$_SALT, UAI$_ENCRYPT, UAI$_PWD, UAI$_PWD_DATE, and UAI$_FLAGS, a site-security administrator can construct a utility that propagates password changes throughout the network. Note, however, that Digital does not recommend using the same password on more than one node in a network.

22.6 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.

22.6.1 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 SYSMAN Utility. 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.

22.6.1.1 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 ιne 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"
```

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.

22.6.1.2 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.

22.6.2 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 Chapter 14.) 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.

Warning: The two global routines let a security administrator obtain both the proposed plaintext password and its equivalent quadword hash value. All security administrators should be aware of this feature, as its subversion by a malicious privileged user will compromise your system's security. See the following recommended procedures.

Digital recommends that you use the following commands to place security alarm ACEs on the shareable image and 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 ACL alarms using the following command:

```
$  SET AUDIT/ALARM/ENABLE=ACL
```

Once in place, these alarms will catch all attempts to replace or to modify the VMS$PASSWORD_POLICY image.

22.6.2.1 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.

22.6.2.2 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 to ensure that the SYSGEN parameters are set correctly on subsequent system startups.

```
$   @SYS$UPDATE:AUTOGEN SAVPARAMS SETPARAMS
```

23 Run-Time Library Routines

This chapter describes new features of the Run-Time Library (RTL) Parallel Processing (PPL$) and Mathematics (MTH$) facilities.

23.1 Parallel Processing (PPL$)

The VMS Version 5.4 RTL Parallel Processing (PPL$) facility contains 19 new routines that complement the basic suite of functionality originally provided for VMS Version 5.0. New features of the PPL$ facility include the following:

- A routine, PPL$CREATE_APPLICATION, that informs the PPL$ facility that the caller is forming or joining a parallel application. This routine replaces PPL$INITIALIZE, which is obsolete beginning with Version 5.4 of the VMS operating system,
- Routines that implement **work queues**. Work queues allow one or more processes to serve as dispatchers of work items to be performed by other processes. Work queues also provide process synchronization.

  Routines that implement work queue synchronization are as follows:

  PPL$CREATE_WORK_QUEUE
  PPL$DELETE_WORK_QUEUE
  PPL$READ_WORK_QUEUE
  PPL$DELETE_WORK_ITEM
  PPL$INSERT_WORK_ITEM
  PPL$REMOVE_WORK_ITEM

  A new example program written in VAX C shows how to use the work queue routines and PPL$CREATE_APPLICATION.
- Routines that delete a PPL$ application or object. These routines are as follows:

  PPL$DELETE_APPLICATION
  PPL$DELETE_BARRIER
  PPL$DELETE_EVENT
  PPL$DELETE_SEMAPHORE
  PPL$DELETE_SPIN_LOCK
  PPL$DELETE_VM_ZONE
- Routines that set and adjust a semaphore maximum (analogous to setting and adjusting a barrier quorum). These routines are as follows:

  PPL$ADJUST_SEMAPHORE_MAXIMUM
  PPL$SET_SEMAPHORE_MAXIMUM

- Routines that disable event notification and reset an event state ("untrigger" an event). These routines are as follows:

  PPL$DISABLE_EVENT
  PPL$RESET_EVENT

- Routines that read a spin lock state and find a synchronization element or shared memory zone's identifier, as follows:

  PPL$READ_SPIN_LOCK
  PPL$FIND_OBJECT_ID

  PPL$FIND_OBJECT_ID replaces PPL$FIND_SYNCH_ELEMENT_ID, which is beginning with Version 5.4 of the VMS operating system.

For a detailed description of the new PPL$ routines, refer to the *VMS RTL Parallel Processing (PPL$) Manual.*

23.2 Mathematics (MTH$)

The RTL MTH$ facility provides the following new and modified sets of routines to support vector processing:

- Basic Linear Algebra Subroutines (BLAS) Level 1
- First Order Linear Recurrence (FOLR)
- Routines that have been vectorized to support Digital vectorizing compilers such as the VAX FORTRAN High Performance Option (HPO)

See Section 2.3.1 for more information about using RTL MTH$ routines in a vector processing environment. See *VMS RTL Mathematics (MTH$) Manual* for complete descriptions of all new, modified, and existing RTL MTH$ routines.

24 VMS Record Management Services

This chapter describes the following enhancements to VMS Record Management Services for Version 5.4 of the VMS operating system:

- Asynchronous support for process-permanent files
- Increase in local buffer limit
- Access-mode protection
- Expired-date suppression

24.1 VMS RMS Asynchronous Support for Process-Permanent Files

Prior to Version 5.4 of the VMS operating system, VMS RMS ignored the asynchronous option for process-permanent files. VMS RMS now supports this option, which affects the performance options within the following two RMS control blocks:

RMS Control Block	Field	Performance Option
File Access Block (FAB)	FAB$L_FOP	FAB$V_ASY
Record Access Block (RAB)	RAB$L_ROP	RAB$V_AST

24.2 Local Buffer Maximum Increased

With Version 5.4 of the VMS operating system, the maximum number of local buffers is increased to 32,767. Prior to Version 5.4, you were limited to specifying no more than 127 local buffers for a record stream from the VMS RMS interface using the RAB multibuffer count field (RAB$B_MBF). You obtain the additional local buffering capability by using the **multibuffer count XABITM**. The multibuffer count XABITM is used as an input to the Connect service only. It is not used as an output by any service.

The maximum number of local buffers established by the DCL command SET RMS_DEFAULT for a *process* has also increased from 127 to 255. However, the maximum number of local buffers established by the DCL command SET RMS_DEFAULT for the *system* remains 127.

The XAB$_MULTIBUFFER_COUNT XABITM requires a 4-byte buffer to store the value that specifies the number of local buffers. To specify the number of local buffers, set up the XAB$_MULTIBUFFER_COUNT XABITM with the number of local buffers desired. Then, link the XABITM into the XAB chain for the record stream prior to invoking the Connect service. When you use the multibuffer count XABITM, the value specified overrides any value that resides in the RAB$_MBF for the related record stream. See Chapter 11 of the *VMS Record Management Services Manual* for details about using an XABITM.

Before you increase the size of the local buffer pool, you should consider current memory management parameters because excessively large buffer pools introduce additional paging that can reduce I/O performance.

24.3 Access-Mode Protection for VMS RMS

VMS RMS now provides access-mode protection for its services and associated memory. This feature is analogous to the protection provided by the system services $ASSIGN and $SETPRT.

No code changes are required for RMS calls involving a single access mode. A code change might be required for RMS calls that initiate operations from an inner access mode and allow subsequent RMS operations from an outer access mode.

If an inner-mode caller initiates an RMS operation without overriding the access mode, subsequent outer-mode calls fail with an RMS$_PRV error. The arguments in the following code example are used to override the caller's access mode. These arguments, together with related topics, are described in the section on access modes in *Introduction to VMS System Services*.

```
FAB$V_CHAN_MODE = PSL$C_<USER,SUPER,EXEC,KERNEL>  ! Select one
```

VMS uses the maximized value of the caller's access-mode and the FAB$V_CHAN_MODE argument (RMS access-mode argument) to establish the access mode.

24.3.1 Access-Mode Protected Services

The following services initiate operations on files. These services establish the access mode that VMS RMS uses to validate the access modes of subsequent accessing services.

$CREATE	$OPEN	$PARSE	$SEARCH

The following services access open files to perform various VMS RMS operations. The access modes for each service trying to access an open file must be validated before RMS operations are allowed.

$CLOSE	$CONNECT	$DELETE	$DISCONNECT
$DISPLAY	$EXTEND	$FIND	$FREE
$FLUSH	$GET	$NXTVOL	$PUT
$READ	$RELEASE	$REWIND	$SPACE
$TRUNCATE	$UPDATE	$WAIT	$WRITE

VMS RMS does not validate the access mode for the following services because access-mode comparison is not relevant to them.

$ENTER	$ERASE	$REMOVE	$RENAME

24.3.2 Access-Mode Protected Memory

VMS RMS now protects the following data structures and their associated I/O buffers at EW (executive read/write). Previously, the data structures were protected at UREW (user read, executive write).

- RMS-controlled data structures
- Process-permanent data structures
- Image-activated data structures

The following memory protection exceptions apply to user-mode accessors of RMS and are protected at UREW:

- Internal RMS I/O buffers—to facilitate RAB$V_LOC mode
- RMS buffers containing collated tables used for indexed files

24.4 Expired-Date Suppression

The file system, in conjunction with parameters established using the DCL interface (see the Set Volume command in *VMS DCL Dictionary*), gives users a facility for determining whether a data file has expired and is eligible to be transferred to another storage medium. Expiration of a file is determined by the Expiration Date and Time, which should not be updated for maintenance functions or for any function where the data is not really being modified.

Prior to VMS Version 5.4, the ability to suppress the expiration update was available only to applications that interface directly with the file system through the $QIO system service. (See ACP Functions in *VMS I/O User's Reference Manual: Part I.*) Now the ability to selectively suppress the update of the Expiration Date and Time is available to all applications through the RMS interface.

24.4.1 The Role of XAB$_NORECORD XABITM

The XAB$_NORECORD XABITM suppresses the update of the Expiration Date and Time on the $CLOSE service. The Expiration Date and Time is used by VMS to determine if the data in a disk file has been accessed recently. Normally, when data has been read or written to a disk file, the $CLOSE service updates the Expiration Date and Time to the current date and time. This moves back the date and time when the file is considered expired. Specifying the XAB$_NORECORD XABITM suppresses the update of the Expiration Date and Time.

The XAB$_NORECORD XABITM uses a 4-byte buffer to set the NORECORD flag to logic 1 using the symbol XAB$_ENABLE. Any other value in this XABITM buffer returns an RMS$_XAB error. An application cannot disable this option because the ODS–II ACP does not support disabling this function once it has been selected on a $OPEN or $CREATE.

24.4.2 Applications for XAB$_NORECORD XABITM

Typically, the XAB$_NORECORD XABITM is used by directory or maintenance routines that do not manipulate the data and, therefore, does not change the expiration status of a disk file. For example, the DCL command DIRECTORY/FULL uses the XAB$_NORECORD XABITM as it opens files to access prolog data containing key information. In this case, DIRECTORY displays prolog information but does not display or modify user data in the disk file and should not modify the Expiration Date and Time. Maintenance utilities should consider using this XABITM. For example, a disk defragmentation utility should not modify the expiration status of a disk file.

Digital recommends using the XAB$_NORECORD XABITM on the $OPEN service instead of on the $CLOSE service—because the suppression of the Expiration Date and Time update is guaranteed should the file deaccess or should a close occur because of process deletion or RMS rundown.

XAB$_NORECORD can be enabled on input to the $CLOSE, $OPEN and $CREATE services. If the $CREATE service opens an existing file through the Create-if option and the Expiration Date and Time are not to be modified, the XAB$_NORECORD XABITM can be specified. When the XAB$_NORECORD XABITM is used on a $CREATE that *creates* a file, it disables the update on the subsequent $CLOSE but does not prevent initialization of the Expiration Date and Time on the file creation in the ACP.

The XAB$_NORECORD XABITM can be sensed on output from RMS for the $OPEN, $CREATE, $DISPLAY, and $CLOSE services. An application typically senses the XAB$_NORECORD XABITM to determine if the XABITM was specified on a previous $OPEN or $CREATE option or if it is specified by the current RMS operation.

25 I/O Driver Support

This chapter describes new VMS Version 5.4 I/O driver support in the following areas:

- Pseudoterminal driver
- Shadow Set Virtual Unit Driver (SHDRIVER)
- TRM$_MODIFIERS item code
- Itemlist read function I/O status block
- ACP-QIO function attributes

See the revised *VMS I/O User's Reference Manual: Part I* for complete information.

25.1 Pseudoterminal Driver

The Pseudoterminal Driver (FTDRIVER) is now part of the VMS operating system. This driver, along with several control connection (PTD$) routines, enables you to create, use, and manipulate pseudoterminals with the VMS operating system. The *VMS I/O User's Reference Manual: Part I* describes pseudoterminal driver functions and capabilities, and lists the VAX calling standards for the control connection routines.

25.2 Shadow Set Virtual Unit Driver

The Shadow Set Virtual Unit Driver (SHDRIVER) is now part of the VMS operating system. This driver supports VMS Volume Shadowing phase II, which provides the same capabilities as VAX Volume Shadowing phase I but includes support for all DSA disks. The *VMS I/O User's Reference Manual: Part I* describes shadow set virtual unit driver functions and capabilities.

25.3 New Modifier Bits for TRM$_MODIFIERS Item Code

Six new modifier bits have been added for the TRM$_MODIFIERS item code for itemlist terminal driver read verify operations:

- TRM$M_TM_ARROWS—The terminal interprets the left and right arrow keys.
- TRM$M_TM_NOCLEAR—Fill characters are not replaced with clear characters after a non-fill character occurs.
- TRM$M_TM_OTHERWAY—Enables left-justify insert mode and right-justify overstrike mode.
- TRM$M_TM_TERM_ARROW—The read operation is terminated when a left or right arrow key is entered at the corresponding margin.

- TRM$M_TM_TERM_DEL—The read operation is terminated when a DELETE key is entered at the left margin.
- TRM$M_TM_TOGGLE—Enables Ctlr/A to function as a toggle key between insert mode and overstrike mode.

Table 8–8 in the *VMS I/O User's Reference Manual: Part I* describes the TRM$_MODIFIERS item code and its associated modifier bits.

25.4 Itemlist Read Function I/O Status Block

In the I/O status block (IOSB) for the itemlist read function, the byte at IOSB+5, which formerly returned –1, now returns status information. Table 8-15 in the *VMS I/O User's Reference Manual: Part I* lists this status information.

25.5 New ACP-QIO Function Attributes

There are three new attributes for ACP-QIO functions:

- ATR$C_DELETE_ALL—Deletes the entire access control list (ACL)
- ATR$C_GRANT_ACE—Returns an access control list entry (ACE) that grants or denies access
- ATR$C_NEXT_ACE—Points to the next ACE in the ACL

Table 1–7 in the *VMS I/O User's Reference Manual: Part I* describes ACP-QIO function attributes.

26 System Dump Analyzer Utility (SDA)

This chapter describes two new qualifiers to the SHOW PROCESS command now available with Version 5.4 of the VMS System Dump Analyzer Utility (SDA).

26.1 New SHOW PROCESS Qualifier: /IMAGES

The /IMAGES qualifier to the SDA command SHOW PROCESS displays the address of the Image Control Block, the starting and end addresses of the image, the activation code, the protected and shareable flags, the image name, and the major and minor ID of the image.

The following is an example of output displayed by the SHOW PROCESS /IMAGES command:

```
                       Process activated images
                       ------------------------

ICB       Start     End       Type            Image Name  Major ID,Minor ID
--------  --------  --------  --------------  ------------------------------
7FF83878  00000200  00000DFF  MAIN            SHOW_PROC_IMAGES  0,0
7FF84100  0003AC00  0003FBFF  GLOBAL PRT SHR  DECW$TRANSPORT_COMMON  12,12
7FF84400  00036200  0003ABFF  GLOBAL          CONVSHR  1,0
7FF84470  0002E400  000361FF  GLOBAL          FDLSHR  1,0
7FF84560  00021A00  0002E3FF  GLOBAL          SORTSHR  2,28
7FF845D0  00000E00  000089FF  GLOBAL          LIBRTL2  1,12
7FF835F8  00008A00  000219FF  GLOBAL      SHR LIBRTL  1,14
7FF84800  00060C00  000767FF  MERGED      SHR ADARTL  0,0
7FF84720  00076800  000A03FF  GLOBAL      SHR MTHRTL  129,32781

Total images = 9  Pages allocated = 1017
```

The following are possible values for the activation code:

- MAIN—Image is the object of a RUN command
- MERGED—Image is an additional mapped image
- GLOBAL—Image is a global image section

The protected flag (PRT) indicates that the image is installed protected. The shareable flag (SHR) indicates that the image is installed shareable.

For more information on the SDA command SHOW PROCESS, see the *VMS System Dump Analyzer Utility Manual.*

26.2 New SHOW PROCESS Qualifier: /VECTOR_REGISTERS

The System Dump Analyzer lets you examine vector instructions and vector context from a system dump file or in a running system. One way to accomplish this is by specifying the new /VECTOR_REGISTERS qualifier to the SHOW PROCESS command, which obtains the values of the registers from the process's vector context area. See Section 2.3.5.2 for a complete description of SDA support for vector processing.

27 Device Support

The VMS Version 5.4 operating system provides device support for writing and debugging driver software for VAX 9000 and VAX 6000 systems. This chapter describes this support for programmers who write and debug driver software for non-Digital-supplied devices attached to a VAX 9000 system. For more detailed information about device support driver programming, see the revised *VMS Device Support Manual.*

27.1 VAX 9000 Hardware Considerations

The VAX 9000 bus architecture illustrated in Figure 27–1 features multiple XMI buses for large systems. A system control unit (SCU) and I/O control unit translate each address and connect a VAX 9000 CPU or memory bus to a target XMI and device or bus adapter. The SCU and I/O control unit connect to each XMI through an XJA adapter. Then, various bus adapters on the XMI provide connection to VAXBI, Ethernet, and Computer Interconnect (CI) buses, which are the second level of the bus architecture. A KDM70 adapter on the XMI bus provides a direct connection to disk or tape devices. Device support is provided for non-Digital-supplied devices connected to second level I/O buses and below. Generic XMI support is not provided.

27.2 VAX 9000 System Address Space

A VAX 9000 system supports 30-bit addressing on each XMI bus and provides 1 gigabyte of physical address space. The total address space is divided in equal halves by memory and I/O address space, as shown in Figure 27–2.

All memory locations on a VAX 9000 XMI bus are addressed using physical addresses in XMI memory space (from $0000\ 0000_{16}$ through $1FFF\ FFFF_{16}$). An XMI device that accesses memory directly (or indirectly through a system-interconnect adapter) or its driver must perform virtual-to-physical translation before transmitting a memory address on the bus.

VAX 9000 XMI I/O address space (physical addresses $2000\ 0000_{16}$ through $3FFF\ FFFF_{16}$) is partitioned as illustrated in Figure 27–3. Macro $IO9AQDEF contained in SYS$LIBRARY:LIB.MLB defines symbols describing the layout of I/O address space.

Figure 27–1 VAX 9000 System Architecture

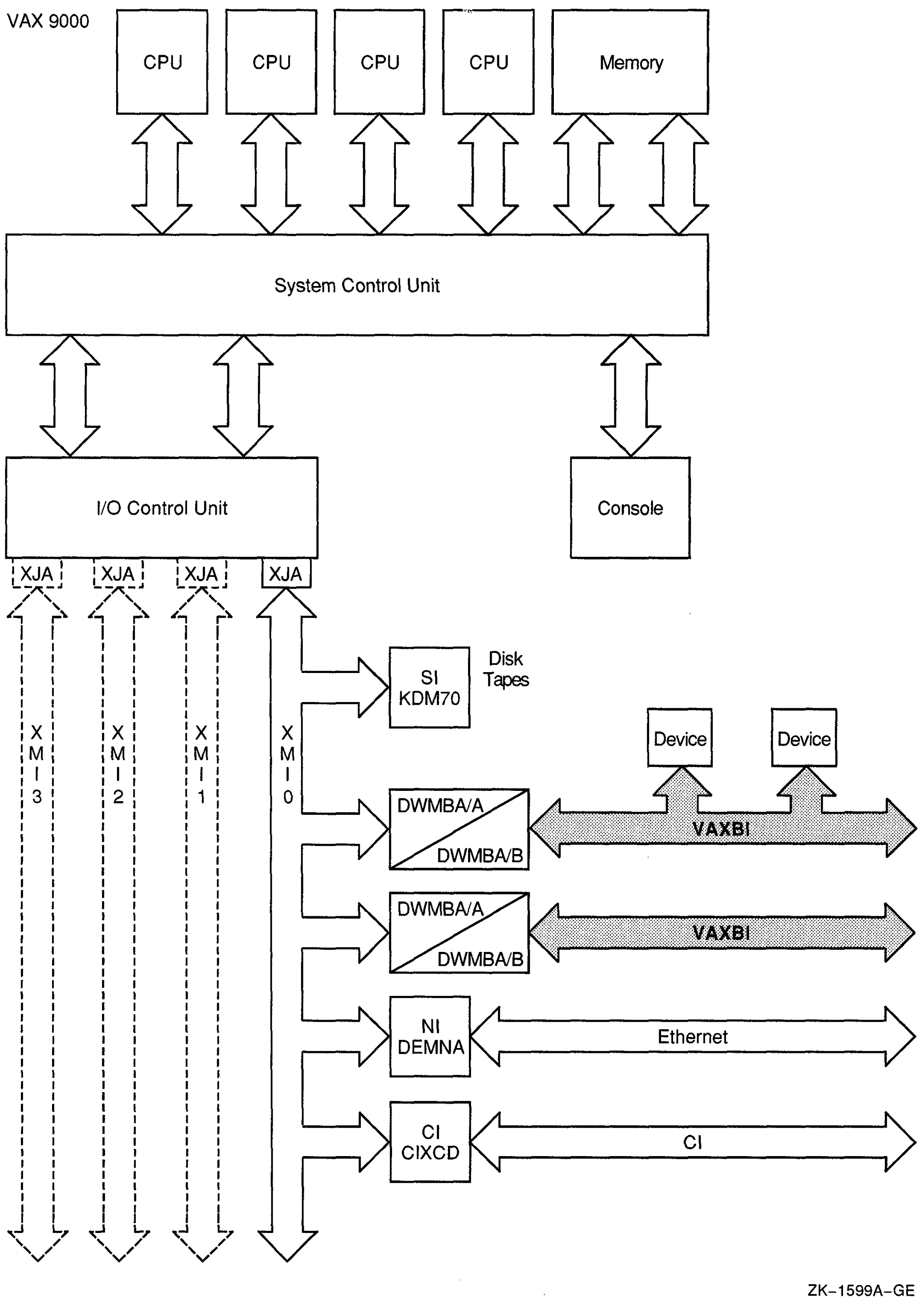
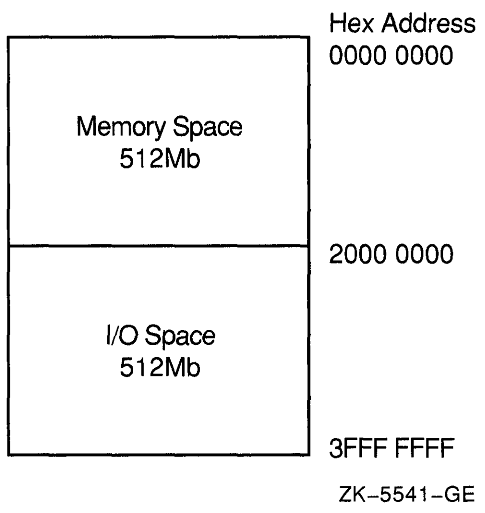


Figure 27–2 VAX 9000 XMI Address Space



The assignment of I/O addresses, shown in Figure 27–3, for any VAX 9000 system supports two levels of bus structure: the XMI and the VAXBI. A VAX 9000 system uses the XMI as the primary I/O bus and can have up to 4 XMIs, depending on the model. Each XMI can have up to 12 nodes or devices numbered 1 through D hexadecimal. Note that the XJA adapter occupies node 0 on an XMI.

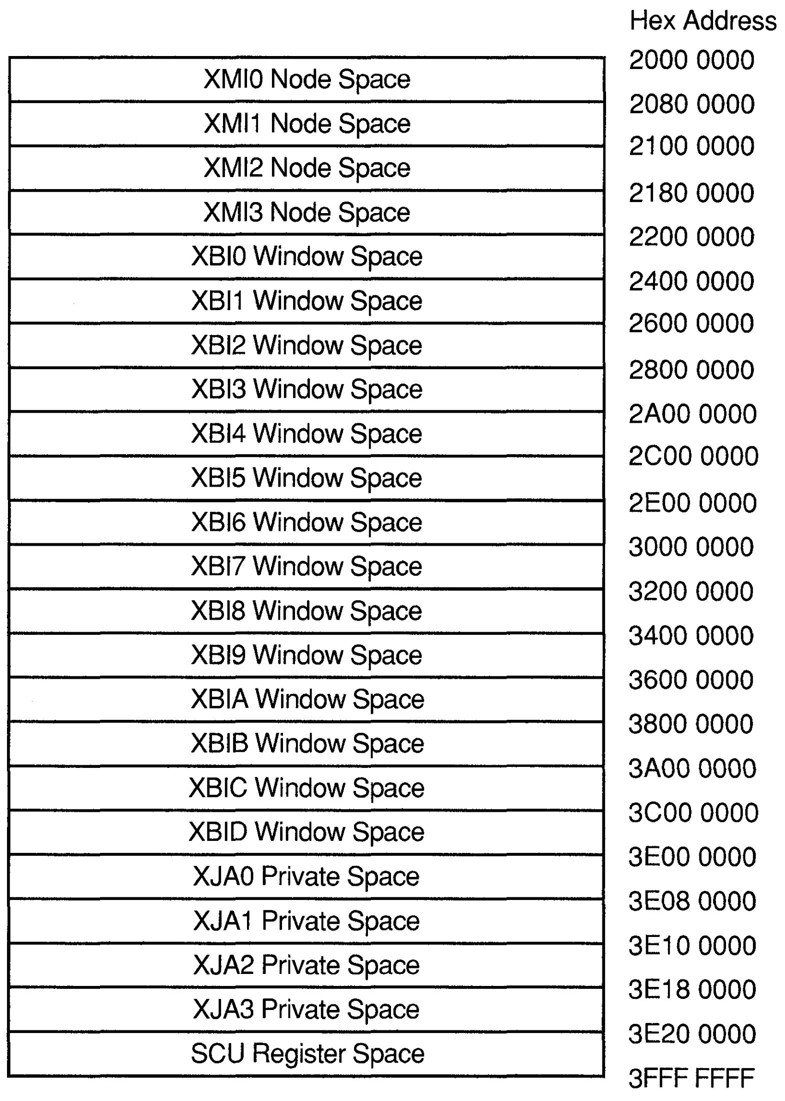
Each XMI-to-VAXBI adapter (DWMBA/A) provides connection to the second level I/O subsystem. At the second level, device support is provided for non-Digital-supplied devices connected to the VAXBI bus. See the *VMS Device Support Manual* for more specific information about these generic VAXBI devices.

Each XBI (14 maximum) is physically mapped into its own XBI window space (XBIx) in the I/O block. The four XMI buses are assigned the first region of the I/O block, XMI0 through XMI3, that spans the node space region. The window spaces for each VAXBI are next, contiguously assigned XBI0 through XBID$_{16}$, spanning the XBI window space region. This allows CPUs to address the individual XMI device CSRs as well as the individual BI device CSRs.

In XMI I/O space, a given XBI's window space is determined by the XBI's XMI node number. There is a limit of 8 XBIs on a given XMI bus and a limit of 14 XBIs across all XMI buses. In the assignment of XBI window spaces, the XBI with the lowest node number on XMI0 is assigned to XBI0 window space. The XBI with the second lowest node number in XMI0 is assigned to XBI1 window space, and so on through all XBIs on XMI0, then to XMI1, XMI2, and until XMI3 is exhausted or the 14th XBI is found.

A 40-bit bus (bits 39 to 0) in the SCU is transparent to devices and nodes on the XMI and VAXBI buses. As shown in Figure 27–4, I/O space on the 40-bit bus starts at address 80 0000 0000$_{16}$ and is entered when bit 39 is set to a 1. Beginning at 80 0180 0000 is the array of 16 x 512Kb XMI

Figure 27–3 SCU/XMI Systems I/O Address Space



ZK–1938A–GE

node space allocations. This provides a 512Kb node space for each possible XMI node. The System Control Unit, I/O control unit, and XJA adapter of the SCU use this map to translate and select one of the eight XBI window spaces from an address presented by a CPU.

An XJA of the SCU/XMI bus architecture is an adapter that connects the SCU ports to the XMI bus. The XJAs have a private space region in the I/O block that allows CPUs to address the XJA CSRs. Since there are up to four XMIs, there can be four XJAs. In the XJA private space region, XJAs are mapped XJA0 through XJA3.

Figure 27–4 SCU Bus Address Allocation

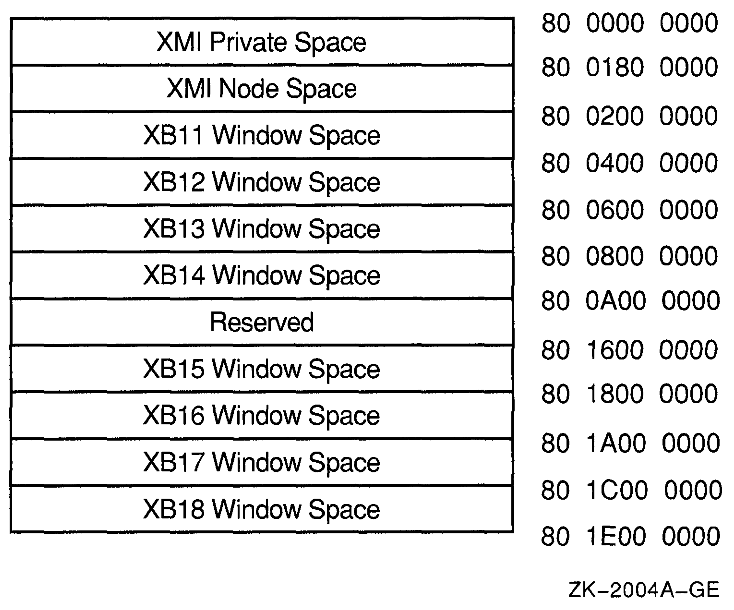
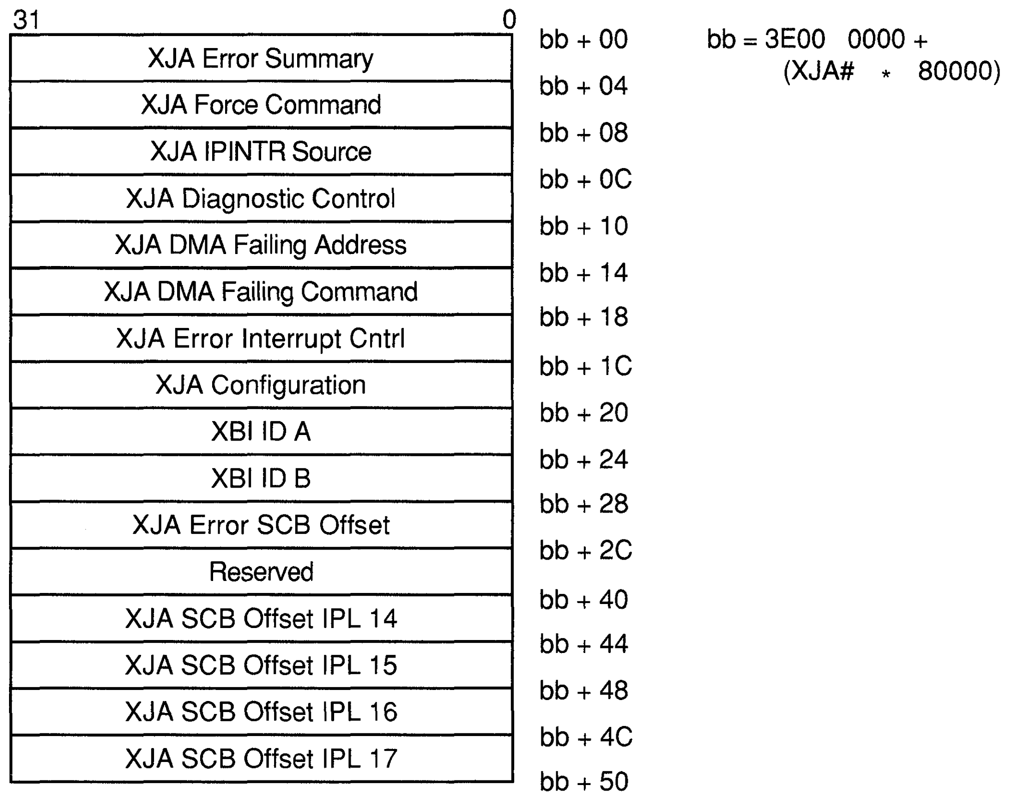


Figure 27–5 describes the contents of each XJA private space. CSRs XBI ID A and XBI ID B of this region are used by the XJA in translating a XBI window-space address into its appropriate XMI XBI window-space address.

The last region is the SCU register space that allows CPUs to address the CSRs of the System Control Unit and Console.

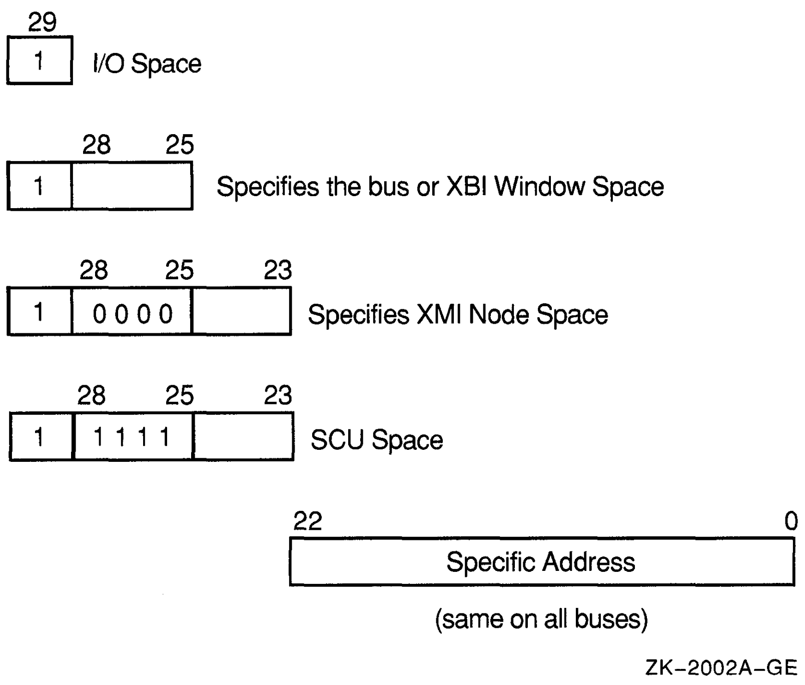
Figure 27–6 shows the bit structure and occurrences of a 30-bit I/O-space address. The address bits are numbered 0 through 29, right-to-left. Bits 0 through 22 translate the same for all buses. The occurrence of a 1 in bit 29 indicates an I/O bound address. With bit 29 set, bits 25 through 28 specify an XBI Window space address. When bit 29 is set and bits 25 through 28 are 0, an XBI Window address is indicated, or if bits 25 through 29 are 1, a SCU bound address is present.

Figure 27–5 XJA Private Space Address Allocation



ZK–2003A–GE

Figure 27–6 SCU/XMI Systems Address Bit Structure



27.3 Driver Debugging with Pool Checking

The Pool Check mechanism provided with VMS has been enhanced to facilitate the debugging of a device driver in the context of driver memory allocation and deallocation techniques. A new bit (bit 5) in the **flags** byte of the POOLCHECK system parameter has been added which, when set, validates the look-aside list deallocation operation (when freeing an SRP, IRP, or LRP). For more information, see the *VMS Device Support Manual.*

The Poolcheck bugcheck has also been enhanced to distinguish between several types of crashes, in addition to a corrupted packet condition. When a crash occurs, the top-of-stack longword now contains a cause value, as described further in *VMS Device Support Manual.*

28 VAX Text Processing Utility (VAXTPU)

This chapter describes VAXTPU Version 2.6 new features and enhancements included in Version 5.4 of the VMS operating system. See the revised *VAX Text Processing Utility Manual* for more detailed information.

28.1 New Qualifier: /INTERFACE

The /INTERFACE qualifier, which you use to specify either character-cell or DECwindows interface, has been added for compatibility with other DECwindows applications. It is virtually the same as the /DISPLAY qualifier.

28.2 New and Enhanced Built-In Procedures

The following built-in procedures are new or enhanced for the Version 5.4 VMS operating system:

- GET_INFO

  New GET_INFO built-in procedures enable you to determine the following:

  - Widget input focus
  - Keystroke journal recovery
  - Work file name
  - Key name
  - Cursor position after vertical motion operations
  - Scroll setting
  - Automatic pop-up menu positioning

- MARK

  An enhancement to the MARK built-in procedure lets you create markers at arbitrary positions within a buffer. Previously, markers could be created only at the current editing point.

- SET (KEYSTROKE_RECOVERY)

  This new built-in procedure turns keystroke journal recovery on or off, regardless of whether such recovery was specified when the VAXTPU session was started.

- SET (MENU_POSITION)

  This new built-in procedure lets you set automatic pop-up menu positioning for one or more pop-up widgets.

- SET (MOVE_VERTICAL_CONTEXT)

  This new built-in procedure allows an application to restore the column context value for a buffer.

- SET (SCROLLING)

  An enhancement to the SET (SCROLLING) built-in procedure enables you to specify jump scrolling or smooth scrolling.

28.3 Work File Support

You can significantly increase the size of the files you edit with VAXTPU by creating work files. Work files make it possible for VAXTPU to handle files that are larger than the available virtual memory space.

29 VAX RMS Journaling: Support for DECdtm Services

This chapter describes VAX RMS Journaling enhancements that support DECdtm services for Version 5.4 of the VMS operating system. (See Chapter 3 for a complete description of DECdtm services.) VAX RMS Journaling continues to support existing applications developed on previous versions of VAX RMS Journaling.

29.1 Support for DECdtm Transactions

The DECdtm **transaction** has superseded the Recovery Unit Facility (RUF) **recovery unit**. In VAX RMS Journaling Version 5.4, an RMS recovery unit is the recoverable work performed by a single process within a DECdtm transaction.

The RUF recovery unit services have been superseded by corresponding DECdtm transaction services, as follows:

RUF Recovery Unit Service	DECdtm Transaction Service
$START_RU	$START_TRANS(W)
$END_RU	$END_TRANS(W)
$ABORT_RU	$ABORT_TRANS(W)

In addition, a single DECdtm transaction service, $END_TRANS(W), has replaced two other RUF services, $PREPARE_RU and $COMMIT_RU, which together were equivalent to the $END_RU service.

For more information about the DECdtm transaction services, see Chapter 22.

29.2 RUF Services Emulated

Recovery Unit Facility (RUF) services are still supported. They are emulated transparently using DECdtm transaction services.

You do not have to recompile or relink your applications to run them under VAX RMS Journaling Version 5.4.

You can convert an application that uses only one active transaction at a time to use the DECdtm services by replacing calls to RUF services with calls to the corresponding DECdtm transaction services.

However, combining DECdtm transaction services and RUF recovery unit services in a single image requires care. You should avoid having transactions that were started using the DECdtm services active at the same time as transactions that were started using the RUF services.

29.3 Network Support

Remote RMS files marked for recovery unit journaling can be modified within a transaction. They will be included in the *atomic unit of work* defined by the transaction. A **remote** file is a file accessed by a **client** RMS process through the DAP/FAL protocol to a "server" system.

The following conditions apply to remote files:

- Remote files can be marked for any combination of RU (recovery unit), AI (after-image), or BI (before-image) journaling.
- All journaling takes place locally with respect to each file.
- All recovery takes place locally with respect to each file.
- Both client and server nodes must support DECdtm (that is, must be running VMS Version 5.4 or later).
- The server node must be licensed for RMS Journaling.
- The DIRECTORY/FULL and ANALYZE/RMS commands have been enhanced to display the type of journaling enabled but not the names of any AI or BI journals.
- The SET FILE/AI_JOURNAL/BI_JOURNAL/RU_JOURNAL command can only be applied to a locally accessed file.

The following examples compare transactions using local or remote access:

```
Local Access                    Remote Access

$OPEN file1                     $OPEN file1
$CONNECT stream1 to file1       $CONNECT stream1 to file1
$OPEN file2                     $OPEN n2::file2
$CONNECT stream2 to file2       $CONNECT stream2 to file2

$START_TRANSW                   $START_TRANSW

$GET from stream1               $GET from stream1
$UPDATE to stream1              $UPDATE to stream1
$PUT to stream2                 $PUT to stream2

$END_TRANSW                     $END_TRANSW
```

The only difference between the two code examples is that, in the remote example, the second file specification includes a node name. As a result, RMS transparently manages two recovery units within the transaction.

The following table summarizes the differences between using recovery unit journaling locally and remotely:

Local Access	Remote Access
One transaction	One transaction
One recovery unit	Two recovery units
One RU journal	Two RU journals

29.4 Record Stream Association

In applications that use the DECdtm transaction services, an RMS record stream is associated with a transaction as a result of an RMS record operation. The application can use either the DECdtm default transaction or the new XABITM item list entry XAB$_TID to determine which transaction the record stream should join.

29.4.1 How Streams Become Associated with a Transaction

Under RMS Journaling Version 5.4, record streams are associated with transactions as follows:

- If the DECdtm services are being used, then eligible streams associate with a transaction at the time of a record operation, not when the transaction is started or the stream is established (as was the case using RUF services).
- A record operation can cause stream association if its action is recoverable. The $PUT, $UPDATE, $DELETE, $FIND, $FREE, $GET, $RELEASE, and $REWIND services might cause an eligible stream to associate with a transaction.
- A record operation must result in stream association if it affects record data in the file. The $PUT, $UPDATE, and $DELETE services must cause an eligible stream to be associated with a transaction.

29.4.2 Stream Association Using RUF and DECdtm Services

The following example compares the way streams are associated with transactions under DECdtm and RUF:

Using DECdtm **Using RUF**

```
$START_TRANSW            $START_RU
$GET from stream1        $GET from stream1
$UPDATE to stream1       $UPDATE to stream1
```

- Using Version 5.4 of RMS Journaling (DECdtm services), the stream associates on the $GET service.
- Using RUF services with versions of RMS Journaling prior to 5.4 and emulation on Version 5.4, the stream associates on the $START_RU.
- In most cases, this difference does not matter and a RUF application can be converted to the direct use of DECdtm services by simple substitution.
- In the cases where it does matter, the association at record operation time is more flexible than association at transaction start (using RUF).

29.5 Detached Recovery

The following sections describe modifications that have been made in the operation of detached recovery—specifically to the performance of synchronous, asynchronous, and partial recoveries.

29.5.1 Synchronous and Asynchronous Recovery

The RMS Detached Recovery server (new image SYS$SYSTEM:RMSREC$SERVER.EXE) can perform both synchronous and asynchronous recovery. Asynchronous recovery is the default mode; it proceeds as follows:

1. Detached recovery "adopts" orphaned transactions by acquiring the record locks for all records modified within a recovery unit. The detached recovery server is multithreaded and performs asynchronous system service calls (including RMS operations).
2. The detached recovery server indicates completion as soon as the record locks have been reacquired. Thus, access to records and files is reenabled sooner.
3. Actual recovery proceeds asynchronously with respect to the original request. This is in contrast to the synchronous recovery that was performed in versions of VAX RMS Journaling prior to Version 5.4.

Synchronous recovery is used in the following circumstances:

- Partial recovery—One or more secondary files are unavailable, so detached recovery cannot acquire all the record locks from an orphaned transaction. See Section 29.5.2 for a detailed description of partial recovery.
- Limited resources—The detached recovery server does not have enough resources to acquire all the record locks on the file to be recovered (for example, a very large database with many active transactions).
- Exclusive access—The process that initiates detached recovery has tried to access the file such that it either has exclusive access to the file or it is the only process that can modify the file. (It may or may not allow shared read access.) In this case, the accessor will not look for record locks from other processes, and the locks owned by detached recovery can create difficulties for the accessor.

29.5.2 Partial Recovery

When detached recovery receives a request to recover a file, it tries to recover all the effects of all orphaned transactions that involve the file. The specific file for which RMS requests recovery is called the **primary** file. In addition to the changes made to the primary file, each of the orphaned transactions can also include changes to a number of other files. These additional files are called **secondary** files.

Recovery of secondary files is not required to allow access to the primary file. If detached recovery cannot access a secondary file referenced in a recovery unit journal for one of the orphaned transactions, then detached recovery cannot adopt that transaction. In such a case, detached recovery recovers that particular recovery unit journal in synchronous mode and omits all operations that involve the inaccessible secondary file. Omitting a secondary file is permissible, since it is necessary only to recover the primary file to satisfy the client's request. All the information necessary to recover the secondary file is left in the recovery unit journal for eventual use in recovering that file.

29.6 Placement of Recovery Unit Journals

In RMS Journaling Version 5.4, the location of a recovery unit journal is determined as follows:

- The first local stream that associates with the transaction selects the location for the RUJ file.
- By default, the recovery unit journal is on the same volume as the file.
- The SET FILE/RU_JOURNAL=(LABEL=*volnam*) command can specify a different volume for all accessors of the file.
- Each accessor can redirect the recovery unit journal by defining a different equivalence name for the logical DISK$*volnam*.
- The XAB$_RUJVOLNAM item list entry on a XABITM block connected to the RAB can be used to override all the preceding factors.
- Recovery unit journals can be reused. When the transaction is completed, the recovery unit journal becomes idle.
- If the process does not have an idle recovery unit journal on the selected volume, then a new one is created.

The following example compares the placement of a recovery unit journal under DECdtm and RUF:

Using DECdtm / **Using RUF**

```
$START_TRANSW                    $START_RU
$GET from parameter(stream1)     $GET from parameter(stream1)
$UPDATE to parameter(stream1)    $UPDATE to parameter(stream1)
```

- Using VAX RMS Journaling Version 5.4 (DECdtm services), the recovery unit journal is created when the $GET service is called.
- Using a version of VAX RMS Journaling prior to Version 5.4 (that is, RUF services), the recovery unit journal is created when the $UPDATE service is called.
- Using RUF emulation on Version 5.4, the recovery unit journal is created when the $START_RU service is called.
- With the VMS Version 5.4 operating system, even read-only transactions require a recovery unit journal, but it will not be written to.

29.7 Multiple Long-Term Journals Allowed

The files involved in a single transaction are no longer restricted to a single after-image journal and a single before-image journal.

29.8 Mixed-Version Clusters

Nodes using versions of VAX RMS Journaling prior to Version 5.4 of the VMS operating system can run together in a VAXcluster with nodes using Version 5.4. Shared access to files marked for journaling is supported in such a mixed-version cluster with one exception: you cannot use a node running an earlier version to recover a file that participated in a transaction that required a two-phase commit. VAX RMS Journaling Version 5.4 includes certain records ("prepare" records) in the journal that earlier versions do not understand.

The following examples show responses to three ways of trying to access the file [FINANCE]PAYROLL.DAT, which has a prepare record in its recovery unit journal, using a version of VAX RMS Journaling prior to Version 5.4:

- If your application tries to access the file directly, RMS returns the following error messages to your application:

```
$ TYPE PAYROLL.DAT
%TYPE-W-OPENIN, error opening WORK1:[FINANCE]PAYROLL.DAT;1 as input
-RMS-E-RRF, recovery unit recovery failed
-RMSREC-F-INVJNLFIL, invalid journal file
```

  In addition, detached recovery sends the following messages to OPCOM:

```
%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.84  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
%RMSREC-F-OPRHDRDET, error occurred during detached recovery unit recovery; init
iated by process ID (PID) 4A2004A0

%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.91  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
%RMSREC-F-INVJNLFIL, invalid journal file

%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.92  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
-RMSREC-F-JNLFILE, journal file DISK$WORK1:[SYSJNL]RMS$0000001E.RMS$JOURNAL;24

%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.93  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
-RMSREC-F-INVJNLIDX, invalid journal index number
```

- If you try to use the Recover Utility (RECOVER) on the file, RECOVER responds with the following messages:

```
%RMSREC-F-NOTCOMREC, file was not completely recovered as requested
%RMSREC-F-LSTVALTIM, time of last valid record: 28-MAY-1990 13:18:06.27
%RMSREC-F-INVJNLFIL, invalid journal file
-RMSREC-F-JNLFILE, journal file DISK$WORK1:[FINANCE]PAYROLL.AIJ1;1
-RMSREC-F-CURNOTSUPP, journal entry: 12 currently not supported
```

- If the file is being accessed by a process on a node running a version of the VMS operating system prior to Version 5.4 and by a process on a Version 5.4 node and the Version 5.4 node fails, the surviving accessor on the other node attempts to perform detached recovery. Detached recovery fails, deletes the surviving process, and sends the following messages to OPCOM:

```
%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.84  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
%RMSREC-F-OPRHDRDET, error occurred during detached recovery unit recovery; init
iated by process ID (PID) 4A2004A0

%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.91  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
%RMSREC-F-INVJNLFIL, invalid journal file

%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.92  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
-RMSREC-F-JNLFILE, journal file DISK$WORK1:[SYSJNL]RMS$0000001E.RMS$JOURNAL;24

%%%%%%%%%%%  OPCOM  30-MAY-1990 09:16:20.93  %%%%%%%%%%%
Message from user BEETHOVEN on EROICA
-RMSREC-F-INVJNLIDX, invalid journal index number
```

To recover the file, you must perform recovery on, or access the file from, a node running VAX RMS Journaling Version 5.4, or you must upgrade the remaining nodes in your VAXcluster to Version 5.4 of the VMS operating system.

30 VMSINSTAL

This chapter describes enhancements to the VMSINSTAL procedure for Version 5.4 of the VMS operating system. See the revised *VMS Developer's Guide to VMSINSTAL* for complete information.

30.1 New Parameter for the VMSINSTAL SPKITBLD.COM Procedure

The Software Product Kit Building Command Procedure (SPKITBLD.COM) now has an additional parameter, the data-file parameter (P4). You use this parameter to specify the name of a data file; SPKITBLD.COM uses the data file to obtain information that it needs to perform a product kit build. For more information on the SPKITBLD.COM data-file parameter, see the *VMS Developer's Guide to VMSINSTAL.*

30.2 New and Enhanced VMSINSTAL Callbacks

The following callbacks are new or enhanced:

- CHECK_VMS_VERSION

  The VMSINSTAL callback CHECK_VMS_VERSION has a new valid value for the option parameter (P4), the value S. When you specify S for the option parameter, messages are not output to the screen or saved during product installation. For more information on the CHECK_VMS_VERSION callback, see the *VMS Developer's Guide to VMSINSTAL.*

- SET

  The VMSINSTAL callback SET has a new option, SET SHUTDOWN. The SET SHUTDOWN option specifies that a reboot is necessary in order to complete the product installation. For a description of the SET SHUTDOWN option, see the *VMS Developer's Guide to VMSINSTAL.*

- GET_PASSWORD

  A new VMSINSTAL callback, GET_PASSWORD, obtains a system-generated or installer-specified password. For more information about the GET_PASSWORD callback, see the *VMS Developer's Guide to VMSINSTAL.*

31 DECwindows and CDA Programming Features

This chapter describes DECwindows and CDA programming components that have been enhanced for Version 5.4 of the VMS operating system, as follows:

- New DECwindows programming examples (Section 31.1)
- Enhancements to the XUI Toolkit color mixing widget (Section 31.2)
- Display PostScript® support (Section 31.3)
- PostScript® support for the CDA VIEW command, CDA Viewer support of Adobe font metrics and DECmath fonts, and the availability of new CDA documentation (Section 31.4)

See the *Overview of VMS Documentation* for more information about the new and revised books referenced in this chapter.

31.1 New Programming Examples in DECW$EXAMPLES Directory

This section describes the following new programming examples that have been added to the DECW$EXAMPLES directory:

- BTrap (Section 31.1.1)
- TestVHist (Section 31.1.2)
- TestVList (Section 31.1.3)
- VDragExample (Section 31.1.4)

31.1.1 BTrap (Broadcast Message Trapper)

BTrap (Broadcast Message Trapper) is a program that traps broadcast messages and displays them in a scrollable output window. The example program shows how to do the following:

- Use XtAddInput to make AST completion routines compatible with the DECwindows toolkit
- Implement pop-up menus
- Implement a custom widget (the TList widget)
- Save current geometry settings in a resource file, under program control

The BTrap package consists of the following files:

- BTrap.c— Main module for the sample program

® Display PostScript is a registered trademark of Adobe Systems Incorporated.

® PostScript is a registered trademark of Adobe Systems Incorporated.

- BTrap.uil—UIL file for the sample program
- TList.c—TList widget sources
- TList.h—TList widget include file
- TList.note—TList widget documentation
- TList.uil—TList widget UIL include file

31.1.2 TestVHist (Histogram Widget Exerciser)

The TestVHist (Histogram Widget Exerciser) program exercises the VHist widget, a dynamically updatable bar histogram widget. It shows how to include a VHist widget in an application program.

The TestVHist package consists of the following files:

- TestVHist.c— Main module for the sample program
- TestVHist.uil—UIL file for the sample program
- VHist.c—VHist widget sources
- VHist.h—VHist widget include file
- VHist.note—VHist widget documentation
- VHist.uil—VHist widget UIL include file

31.1.3 TestVList (VList Widget Exerciser)

The TestVList (VList widget exerciser) program exercises the VList widget, a dynamic, customizable list box widget. It shows how to include a VList widget in an application program.

The TestVList package consists of the following files:

- TestVList.c—Main module for the sample program
- TestVList.uil—UIL file for the sample program
- VList.c—VList widget sources
- VList.h—VList widget include file
- VList.note—VList widget documentation
- VList.uil—VList widget UIL include file
- VFrame.c—VFrame widget sources
- VFrame.h—VFrame widget include file
- VFrame.uil—VFrame widget UIL include file
- VHeader.c—VHeader widget sources
- VHeader.h—VHeader widget include file
- VHeader.uil—VHeader widget UIL include file

31.1.4 VDragExample (VDrag Exerciser)

The VDragExample (VDrag Exerciser) program exercises the VDrag routine, which allows the user to drag widgets about within a window. It shows how to intercept button and motion events to move a widget while tracking the pointer.

The VDragExample package consists of these files:

- VDragExample.c—Main module for the sample program
- VDrag.c—VDrag routine sources
- VDrag.note—VDrag routine documentation

31.2 XUI Toolkit: Enhancements to Color Mixing Widget

The XUI Toolkit color mixing widget, created by the COLOR MIX CREATE routine, now supports both the hue lightness saturation (HLS) color model as well as the red, green, blue (RGB) model. The HLS color model allows users to choose the hue (color), lightness of the color, and the color saturation, which determines how solidly the color appears. The RGB color model provides three scales, each representing a percentage of red, green, and blue.

The same X11 RGB callback information is returned regardless of which color model is used. An option menu presents the HLS and RGB color model choices and allows users to switch between models as often as they wish.

The COLOR MIX CREATE routine includes eleven new attributes that support the HLS model:

- **color_model**—The color model currently being used
- **hue_label**—The label of the hue scale widget
- **light_label**—The label of the lightness scale widget
- **sat_label**—The label of the saturation scale widget
- **black_label**—The label for the zero end of lightness scale widget
- **white_label**—The label for the 100% end of lightness scale widget
- **gray_label**—The label for the zero end of saturation scale widget
- **full_label**—The label for the 100% end of the saturation scale widget
- **option_label**—The label for the color model option menu
- **hls_label**—The label for the color model option menu HLS option
- **rgb_label**—The label for the color model option menu RGB option

See the *VMS DECwindows Toolkit Routines Reference Manual* for a complete description of the COLOR MIX CREATE routine.

31.3 VMS DECwindows Display PostScript System

VMS Version 5.4 includes the Display PostScript® system, which provides text and image display capability for bitmapped workstations. The DECwindows implementation of the Display PostScript system allows programmers to write applications in a general purpose language like C, yet describe the images and text using the device-independent PostScript® software. Programmers can mix X and PostScript routines, even within a single window, using a single network connection to an X server.

The Display PostScript system provides additional imaging capabilities to the X programming routines, as follows:

- Coordinate system that can be moved, rotated, and scaled
- Bezier curves to make it easy to display any curve, no matter how complex
- Device-independent color model
- Text that can be scaled and rotated
- Image operators for scanned images (scaling, rotating, transformations, gray-scale manipulation)

The Display PostScript system is a device-independent graphics architecture that can be implemented on a variety of windowing systems. The Display PostScript server consists mainly of a PostScript interpreter, which executes PostScript language code to display images on a user's screen. The Display PostScript client consists mainly of a Client Library through which an application communicates with the server.

DECwindows implements the Display PostScript system as an extension to the X Window System, on which DECwindows is based. The Display PostScript server is an extension to the X server; the Client Library is an extension to Xlib. The Display PostScript extensions allow an application written in any VAX language to handle Display PostScript programming tasks on an X-based system.

Table 31–1 provides a summary, by topic, of the documentation available for the Display PostScript system and programming in the VMS DECwindows environment.

® Display PostScript is a registered trademark of Adobe Systems Incorporated.

® PostScript is a registered trademark of Adobe Systems Incorporated.

Table 31–1 Display PostScript Documentation

Topic	Type of Information	Document
Programming in the VMS DECwindows environment	Overview and guidelines	*XUI Style Guide* *VMS DECwindows Guide to Application Programming* *VMS DECwindows Xlib Programming Volume*
Programming using the PostScript language	Tutorial	*PostScript Language Tutorial and Cookbook* by Adobe Systems, available in most bookstores
	Reference	*PostScript Language Reference Manual* by Adobe Systems (available in most bookstores)
Programming using the Display PostScript system	Device-independent overview	*Display PostScript System Perspective for Software Developers*
	Device-independent reference	*Display PostScript System Client Library Reference Manual* *Display PostScript System pswrap Reference Manual* *PostScript Language Extensions for the Display PostScript System* *PostScript Language Color Extensions* *PostScript Document Structuring Conventions Specification Version 2.1*
	Device-specific overview and reference	*VMS DECwindows Display PostScript System Programming Supplement*

31.4 Compound Document Architecture (CDA)

The following sections describe CDA new features, which include PostScript support for the VIEW command, CDA Viewer support of Adobe font metrics and DECmath fonts, and the availability of new CDA documentation.

31.4.1 PostScript Support for CDA VIEW Command

The VMS VIEW command invokes the CDA Viewer, which lets you view a compound document file on a character cell terminal or DECwindows display. With the Version 5.4 of the VMS operating system, PS is now a supported input format (.PS file extension).

Note that PostScript file viewing is supported only in the DECwindows CDA Viewer and only when running to displays with servers that contain the Display PostScript Extension. To view a PostScript file, you must specify the /INTERFACE=DECWINDOWS qualifier. PostScript file viewing does not support processing options.

For more information about the VIEW command, see the *CDA Reference Manual.*

For new information about using the DECwindows CDA Viewer to view PostScript files, see Section 7.2, CDA Viewer.

31.4.2 CDA Viewer Support of Adobe Font Metrics and DECmath Fonts

The CDA Viewer now uses the Adobe font metrics in processing a DDIF file for viewing. The font name in a DDIF file follows the X-11 font naming convention. When processing a file from a creating application that uses font metrics other than Adobe font metrics, the CDA Viewer defaults to the Adobe Courier font.

When processing a file for viewing, the DECwindows CDA Viewer queries the X server for a list of available fonts. Although the CDA Viewer does not use these fonts in its calculations, it tries to match the font from the file with an X11 font on the server. If there is not an exact match, the CDA Viewer uses the font from the list that is the closest lower point size for that font name. If there is no match at all, the DECwindows CDA Viewer display type defaults to 12-point Adobe Courier.

The character cell CDA Viewer displays all files in a 12-point Courier font. The contents of each file are spaced and displayed correctly, based on the font that is stored in the file.

The Adobe font metrics are stored in SYS$PS_FONT_METRICS:.AFM on VMS systems and in /usr/lib/font/metrics/ on ULTRIX systems.

The CDA Viewer also supports the DECmath fonts used by DECwrite for equation editing.

For more information, see the *CDA Reference Manual.*

31.4.3 New CDA Documentation

The following separately orderable manuals are now available:

- The *Introduction to the CDA Services* provides an overview of compound document processing terminology and the various components of CDA, which include the CDA Toolkit, the CDA converter architecture, DDIF, and DTIF.
- The *Guide to Creating Compound Documents with the CDA Toolkit* is a tutorial manual that describes how to use the CDA Toolkit to create CDA-conforming compound document applications on VMS and on ULTRIX systems.
- The *CDA Reference Manual* provides reference material that supplements the CDA user guides. It describes the DDIF and DTIF aggregates, the CDA Toolkit routines, the front and back end converter routines, the callable DECwindows and character-cell viewer routines, and the VMS and ULTRIX system commands used to convert and to view compound document files.

See the *Overview of VMS Documentation* for more information about these manuals.

A VMS Version 5.3 Features

This appendix describes features that were new to Version 5.3 of the VMS operating system but are not yet documented in other printed manuals.

A.1 VMS Version 5.3 System Management Features

This section describes enhancements to the following components of the VMS operating system:

- Lock Manager
- NCP Executor Commands

A.1.1 Extension of Lock Manager Limit

The Lock ID space for the Lock Manager is now extended from 65,535 to 262,144 locks. The SYSGEN parameters listed in the following table are increased to the values indicated:

SYSGEN Parameter	New Maximum Value
LOCKIDTBL	262,144
LOCKIDTBL_MAX	262,144
SRPCOUNT	270,336
SRPCOUNTV	270,336
IRPCOUNT	135,168
IRPCOUNTV	135,168

A.1.2 NCP Executor Command Changes

The NCP executor commands now include the following:

- A new parameter to SET/DEFINE EXECUTOR command
- New display characteristics for SHOW EXECUTOR CHARACTERISTICS command

A.1.3 Parameter for SET/DEFINE EXECUTOR

The network control ancillary program (NETACP) manages an index into a properly synchronized table in nonpaged-pool memory. System managers can modify the size of the table using the NCP command SET/DEFINE EXECUTOR with the new parameter MAXIMUM DECLARED OBJECTS.

Parameter	Description
MAXIMUM DECLARED OBJECTS	Specifies the number of objects that processes can declare. To determine the current number of declared objects on your system, use the NCP SHOW KNOWN OBJECTS command. Each of the objects with a PID listed is one declared object. A single process can declare more than one object. Failure to provide a sufficient number of objects can result in the failure of network servers to be initialized. The default of 31 objects is sufficient for most configurations. The valid range is 8 to 16383. Note that dynamically setting the number lower has no effect.

A.1.4 SHOW EXECUTOR CHARACTERISTICS Command

The SHOW EXECUTOR CHARACTERISTICS command now displays information as shown in the following example. Note that a new entry Maximum Declared Objects is displayed and the Pipeline quota now shows 10000.

```
NCP>  SHOW EXECUTOR CHARACTERISTICS

Node Volatile Characteristics as of 16-JUN-1990 10:48:27

Executor node = 2.11 (BOSTON)

Identification           = DECnet-VAX V5.3,  VMS V5.3
Management version       = V4.0.0
Incoming timer           = 45
Outgoing timer           = 45
Incoming Proxy           = Enabled
Outgoing Proxy           = Enabled
NSP version              = V4.1.0
Maximum links            = 128
Delay factor             = 80
Delay weight             = 5
Inactivity timer         = 60
Retransmit factor        = 10
Routing version          = V2.0.0
Type                     = routing IV
Routing timer            = 600
Broadcast routing timer  = 40
Maximum address          = 1023
Maximum circuits         = 16
Maximum cost             = 1022
Maximum hops             = 15
Maximum visits           = 63
Maximum area             = 63
Max broadcast nonrouters = 64
Max broadcast routers    = 32
Maximum path splits      = 1
Area maximum cost        = 1022
Area maximum hops        = 30
Maximum buffers          = 100
Buffer size              = 576
Default access           = incoming and outgoing
Pipeline quota           = 10000
Alias incoming           = Enabled
Alias maximum links      = 32
Alias node               =  2.10 (CLUSTR)
Path split policy        = Normal
Maximum Declared Objects = 31
```

A.2 VMS Version 5.3 Support for the VMS Distributed Name Service

The Distributed Name Service (DNS) is a facility for storing the names of resources in your network such as files, disks, nodes, queues, and mailboxes. The Distributed Name Service clerk is the VMS programming interface to DNS that allows an application to register a resource in the name service and then access the resource from any point in the network by a single name. DNS is a layered product and must be installed in your network before you can start the DNS clerk or utilize the name service.

Applications that need the Distributed Name Service must use the $DNS clerk system service and the DNS run-time routines to register, modify, and locate information in the DNS database. A DNS clerk, which is resident on every VMS Version 5.3 system and later, receives application requests through the $DNS system service. The clerk locates a DNS server that can process the request. Once the request is satisfied, the clerk returns the requested information to the client application.

The information in this section is intended for VMS programmers who are writing applications that call the Distributed Name Service. It includes the following:

- Conceptual information on DNS
- DNS clerk system services, $DNS and $DNSW
- DNS run-time routines
- Startup information for the DNS clerk
- DECnet event messages from the DNS clerk

See the *VMS System Messages and Recovery Procedures Reference Manual* for information about system error messages generated by the DNS clerk.

A.2.1 Introduction to the Distributed Name Service

The VAX Distributed Name Service (DNS) provides a means of assigning unique names to network resources so that a network application or network user can find resources within the network. (Resources are such things as disks, systems, applications, and so on.) Once an application has named a resource using DNS, the name is available for all users of the application. Multiple users located throughout a network can refer to a common resource by the same name. Resources can be moved within the network. No additional preparation is required, and it is not necessary to learn a new naming convention.

You should consider using DNS applications that need to access such remote resources as printers, files, disks, and nodes. In addition, application databases or servers are good candidates for naming. All of these resources would be commonly named and their locations identified within DNS. With DNS, the resource could be moved without users being aware of the change.

Although it is desirable to name application databases, you should ordinarily use DNS to store only the location of the database, not the database itself. (Most database applications require higher levels of consistency than DNS provides.) If the database is relocated, then only the DNS information has to be modified.

A.2.2 The DNS Namespace

The collection of names in the Distributed Name Service database is called a **namespace**. A namespace is located on VMS nodes where the DNS server software is installed. The collection of databases stored on each server makes up the namespace itself.

DNS refers to the named resources in a namespace as **objects**. Each object name refers to a specific entity. The object name is important because applications use the object name in all DNS operations.

Associated with every object is a set of **attributes** describing properties of an object. An application reads object attributes for information such as an address, class, or version.

Most applications use the address attribute of an object, which allows you to find the node on which a resource resides. When a network resource is relocated, an application has DNS update the object's address attribute. All requests for the object receive the new address. Since the object has the same DNS name, the application user can be unaware that the resource has moved.

A.2.2.1 Planning Namespace Objects

When writing applications that use DNS, it is important to determine ahead of time what resources an application needs and how an application will use each resource. Then you can determine what objects an application needs to create and the kind of information each object needs to store. Once the object is designed, you can decide which object attributes to assign and what their values will be.

A.2.2.2 Restrictions

Because of the high cost of keeping copies of DNS names synchronized, you should use DNS applications that store information that does not change frequently. Frequent updates add traffic to the network, which can degrade overall network performance. Because resources such as files, disks, nodes, queues, and mailboxes remain on one node for a long time, a good example of information to store with DNS is a network address.

Not only should the information stored in DNS be relatively static, it should also be verifiable. When DNS updates its database, it attempts to send the update to all copies of the name within 24 hours. This means that your application can request data from a copy of a name that has not been updated. An application must be able to recognize when data is invalid. For this reason, a network address is a good example of data that can be validated. If you use an address and the resource is not there, the data is obviously outdated.

A.2.2.3 Using the Namespace

An application choosing to use the namespace performs four basic operations:

- Object creation—An application needs to create an object to represent each network resource it requires.
- Object modification—Once an object is created to represent a resource, an application modifies the object to contain the attributes and values the application requires.
- Object deletion—When a resource is no longer useful, an application should delete the object.
- Information retrieval—The most common operation an application performs is requesting the DNS clerk to obtain the values of an attribute so that, for example, the application can locate the resource in the network.

A.2.2.4 Object Names

The name DNS assigns to an object is one that the user supplies. The client application translates the name it receives through the user interface from string format into *opaque* format before passing it to the DNS clerk. DNS works only with opaque format because it is guaranteed to be unique, whereas string format often contains logical names that easily change.

The $DNS system service supplies functions for conversion between string and opaque format. If an application maintains its own databases, then the application must store DNS names in opaque format.

A.2.2.5 Object Attributes

Client applications store information about a resource as object *attributes*. When creating an object, an application needs to assign a class name and a version to a new object. The class name reflects the purpose of the object within an application. The purpose can be specific to an application or it can be shared among a group of applications. For example, a group of user names might be shared. An application uses the class name to search for its objects or list its objects. The class version helps to pair a version of an object with a software version.

To store additional information with an object, an application must modify the object.

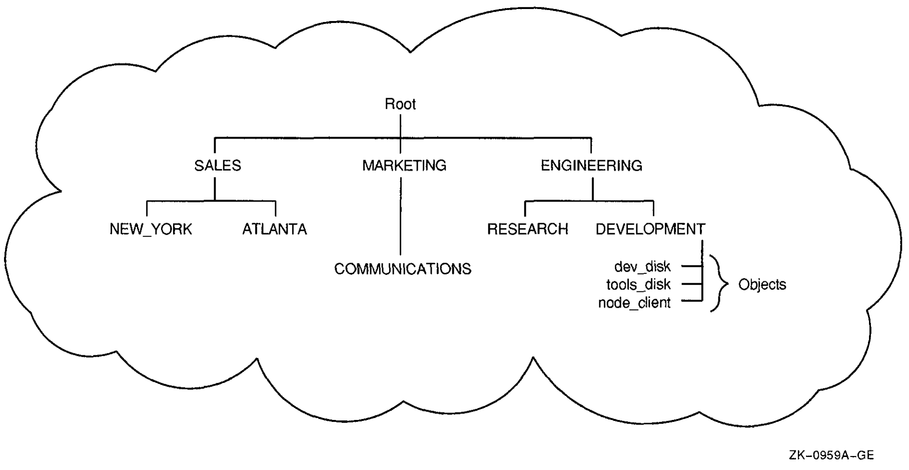
DNS always assigns certain attributes to an object during creation. It assigns a unique identifier (UID) and an update time-stamping (UTS) indicating when an object was last edited. DNS also assigns a third attribute that specifies access control for the new object. Initially, the owner of the object has read, write, delete, control, and test access. The namespace administrator can modify this access according to site requirements.

An attribute name is limited to 31 characters and its value cannot exceed 4000 bytes. The name service assigns a prefix of DNS$ to the name of each attribute it assigns. An application creates a prefix to assign to attributes it creates. For example, DECnet uses the prefix DNA$ and the Distributed File Service uses the prefix DFS$. Names assigned by Digital all contain the dollar sign ($). User-supplied names should use an underscore (_). To ensure uniqueness, you should register your facility name through Digital's product registration program.

A.2.3 Structure of a Namespace

A DNS namespace is a hierarchical set of directories, as depicted in Figure A–1. At the top of the hierarchy is the root directory, which is symbolized by a period (.). Below the root directory are levels of subdirectories. The namespace administrator establishes the directory structure of the namespace and, in some cases, assigns names to directories. While the organization of the namespace directories is similar to the VMS directory structure, namespace directories are completely separate from the VMS directory structure.

Figure A–1 A DNS Namespace



Directories in a namespace can contain three types of entries:

- Objects
- Directory pointers
- Soft links

An **object** represents a network resource. It consists of a name that is unique within the namespace and its associated attributes.

Directory pointers are used internally by DNS to link one level of directories to the next. DNS refers to the hierarchical relationship of directories in terms of *child directories* and *parent directories*.

A **soft link** provides an alternate name for an object, directory, or soft link. For example, a namespace structured with both an organizational and a geographical dimension might access a single object through multiple soft links. A soft link can also be useful in renaming an object. The soft link would point to the original object name so that users could successfully use an outdated name. This kind of soft link would be deleted after sufficient time has passed for applications and users to become aware of the new object name. You create and delete links through the DNS management program.

Although an application requests the creation of an object in order to register a resource, it does not position the object in the namespace. The system administrator determines which directory DNS stores the object in. The structure of a namespace differs for each network, so you should not hard-code names into applications.

A.2.3.1 Naming Syntax

The DNS name of an object, directory, or soft link in the namespace is a complete path specification from the root directory to the entity in the directory of interest. For example, the DNS name .ENGINEERING.DEVELOPMENT.TOOLS_DISK identifies an object named TOOLS_DISK in the namespace directory called .ENGINEERING.DEVELOPMENT. The ENGINEERING directory is in the root directory, and DEVELOPMENT is a child directory of the ENGINEERING directory.

While the full name is a complete path name from the root directory, each element in a full name is called a **simple name**. The last simple name in a full name designates an object, a child directory, or a soft link. In the previous example, TOOLS_DISK is a simple name assigned to a disk object. The maximum length of a simple name is 255 bytes.

You can represent a full name in three ways:

```
namespacename:.simplename.simplename
```

or

```
.simplename.simplename
```

or

```
simplename.simplename
```

If the full name does not start with a namespace name or a period, DNS attempts to translate the first simple name as a logical name. Any equivalence name found is added to the name string in place of the matched simple name. This process is repeated until the first term does not match a logical name or the clerk encounters either a namespace name or a leading period. (A namespace name, assigned during DNS server installation, defaults to *node-name*_NS.)

The following example shows what happens with the name RESEARCH.PROJECT_DISK:

1. Look up RESEARCH as a logical name.

   RESEARCH translates to ENG.RESEARCH, so the name string expands to ENG.RESEARCH.PROJECT_DISK.

2. Look up ENG as a logical name.

   ENG translates to .ENGINEERING, so the name string becomes .ENGINEERING.RESEARCH.PROJECT_DISK. Because the new name has a leading period, translation stops.

3 The namespace name, INMAX_NS, is added to the front of .ENGINEERING because it is not explicitly specified. (A namespace administrator establishes the namespace name during installation.) The name becomes INMAX_NS:.ENGINEERING .RESEARCH.PROJECT_DISK.

A.2.3.2 Logical Names

When the DNS clerk is started on a VMS operating system (see Section A.2.10), the VMS system creates a unique logical name table for DNS to use in translating full names. This logical name table, called DNS$SYSTEM, prevents unintended interaction with other system logical names. The DNS use of logical names in parsing full names is described in Section A.2.3.1.

To define systemwide logical names for DNS objects, use the DCL command DEFINE. For example, to create the logical RESEARCH.PROJECT_DISK shown in the previous section, you would enter 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 ordinarily translates to DNS$SYSTEM in order to access the systemwide DNS 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 DNS using the DCL command DEFINE, as follows:

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

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

A.2.3.3 Valid Characters for DNS Names

DNS namespace names, full names, or simple names can contain letters, numbers, and certain punctuation marks from the ISO Latin 1 character set, as shown in Figure A–2. Additional characters and punctuation marks can appear as long as the name is enclosed in quotation marks, for example, “project%”. See Figure A–3.

Figure A–2 Valid Character Codes for DNS Simple Names

Code Range (Decimal)	Character
036	$
045	–
048–057	0 1 2 3 4 5 6 7 8 9
065–077	A B C D E F G H I J K L M
078–090	N O P Q R S T U V W X Y Z
095	_
097–109	a b c d e f g h i j k l m
110–122	n o p q r s t u v w x y z
192–207	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï
208–214	Ð Ñ Ò Ó Ô Õ Ö
216–223	Ø Ù Ú Û Ü Ý Þ ß
224–239	à á â ã ä å æ ç è é ê ë ì í î ï
240–246	ð ñ ò ó ô õ ö
248–255	ø ù ú û ü ý þ ÿ

ZK–0961A–GE

Note: All simple names containing the dollar sign ($) are reserved for use by Digital.

Figure A–3 Additional Character Codes Allowed in Quoted Simple Names

Code Range (Decimal)	Character
032–033	{*space*} !
035	#
037–044	% & ' ( ) * + ,
046–047	. /
058–064	: ; < = > ? @
091–094	[\ ] ^
096	‘
123–126	{ | } ~
160–167	{*no–break space*} ¡ ¢ £ ¤ ¥ ¦ §
168–174	¨ © ª « ¬ - ®
175–187	¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º »
188–191	¼ ½ ¾ ¿
215	X
247	÷

ZK–0962A–GE

DNS maintains the case of an entity when it registers an object, but it is case insensitive in lookups. For example, the name eng.research would match the name ENG.RESEARCH.

DNS also supports binary simple names. A binary name consists of the leading character pair %x or %X, followed by pairs of hexadecimal digits. A binary simple name does not match any regular or quoted simple name, even if a given name has the same binary value.

DNS makes use of wildcards for identifying groups of objects during search operations. Wildcards consist of the following:

Symbol	Name	Meaning
?	Question mark	Match one character.
*	Asterisk	Match any number of characters.

A.2.4 Creating Objects

Each application that uses DNS must register its resources in the namespace using either the $DNS or the $DNSW system service. Registration involves creating an object in the namespace to represent the resource. You create an object to represent each resource in the network that your application needs to find. At the same time, you should define attributes the object needs and assign their values.

A DNS object consists of a name and its associated attributes. You create the object first, along with some key attributes. Later, you can modify the object to hold additional attributes that are relevant to the application.

To create an object with $DNS:

1 Prompt for a name from the user interface.

  The name that an application assigns to an object should come from a user interface, 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 interface is in string format.

2 Use the $DNS parse function to convert the full name string into the opaque format of DNS.

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 DNS 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, providing 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 DNS 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 DNS status block to be sure there are no errors at the DNS server.

The following C routine 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;
unsigned short class_len;
unsigned char *object_name;
unsigned short object_len;
{
    struct $dnsitmdef createitem[4]; /* Item list used by system service */
    struct $dnscversdef version;     /* Version assigned to the object */
    struct $dnsb iosb;           /* Used to determine DNS server status */
    int status;                  /* Status return from system service */
    /*
     * Construct the item list that creates the object:
     */
    createitem[0].dns$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 Zero terminates an item list.

❻ Call the system service to create the object.

❼ Check to see that both the system service and DNS were able to perform the operation without error.

A.2.5 Modifying Objects

After applications use DNS to create objects that identify resources, they add attributes to the newly created objects that describe properties of the object.

You modify an object whenever you need to add an attribute, change an attribute value, or delete an attribute. You can add as many attributes as you like. If you add the same attribute to an object twice, the time-stamping on the attribute is updated.

DNS attributes can have a single value or they can have a set of values. For example, an attribute holding the class version number of a resource would have a single value, while an attribute holding the location of a service in the network could have a set of values. The set would hold the addresses of all nodes in the network that offer the service. Depending on the attribute type, DNS performs a slightly different action. DNS adds or deletes a value when there is only one. When there is a set of values, DNS adds or deletes a value from an existing group of values.

To modify an object with $DNS:

1. Build an item list containing the following elements:
   - The opaque name of the object you are modifying
   - The type of entry, as described in Section A.2.3
   - The operation to perform
   - The type of attribute you are adding—a single value or a set of values
   - The attribute name
   - The value being added to the attribute
2. Supply any of the optional parameters described in Section A.2.4.
3. Call the modify attribute function, supplying the parameters established in steps 1 and 2.

The following C example shows how to add an attribute and its 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 object entries */
    unsigned char opertype = dns$k_present; /* Adding an object */
    unsigned char attype = dns$k_set;     /* Attribute will be type set */
    struct $dnsb iosb;               /* Used to determine DNS status */
    int status;                      /* Status of system service */
    /*
     * Construct the item list to add an attribute to an object.
     */
    moditem[0].dns$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);

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

    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 the entry is an object—not a soft link or 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 Check to see that both the system service and DNS performed the operation without error.

A.2.6 Distributing the Namespace

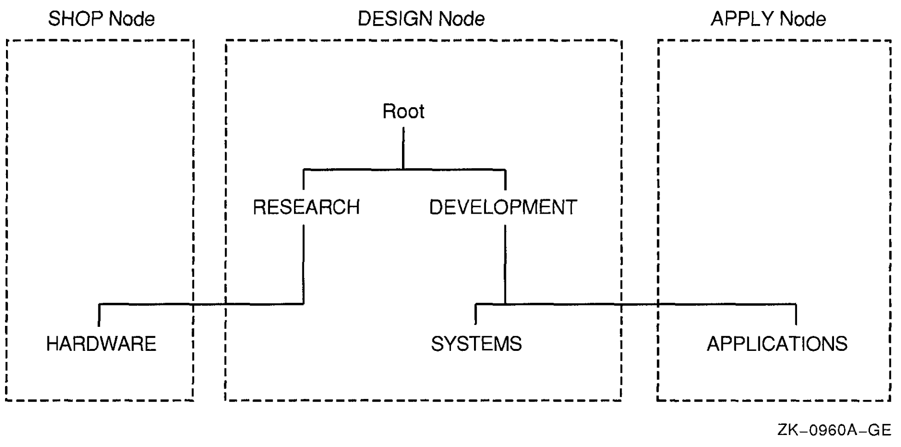
A VMS node running DNS server software can contain the entire namespace. However, performance and reliability are enhanced when several VMS nodes act as DNS servers.

DNS supports the *partitioning* of the namespace across several DNS servers. In this situation, no DNS server contains the entire namespace, but each contains a portion of the namespace, usually the directories frequently accessed by local client applications. Directory pointers connect parts of the the database that are distributed among two or more servers.

Figure A–4 depicts a namespace with three DNS servers. The DESIGN node contains most of the namespace—the root directory plus the research and development directories. The applications directory resides on the APPLY node, while the hardware directory resides on the SHOP node.

DNS refers to a collection of directories on a server as a **clearinghouse**.

Figure A–4 A Partitioned Namespace



A.2.6.1 Replicating Directories

In large networks, many applications rely on DNS and names must be available for the application to work. To ensure availability, DNS allows the duplication of data and provides a mechanism to keep all copies of names synchronized. Then, if one server becomes disabled, applications can still access the namespace through another server. Whenever data is duplicated, DNS copies one or more directories with all their contents.

The namespace administrator determines how many copies of each directory should exist and where they should be located. For example, Figure A–5 shows the same namespace as Figure A–4. However, in Figure A–5 the root directory is duplicated so that it exists on all three DNS servers.

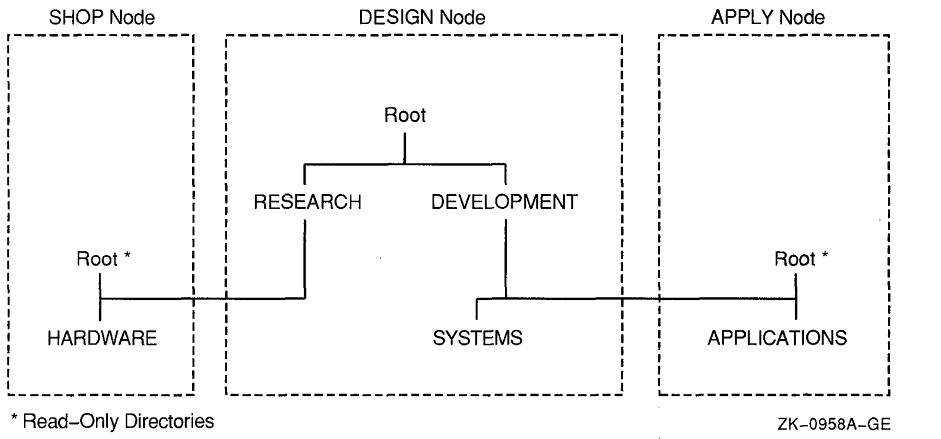
A.2.6.2 Types of Directories

Once you duplicate parts of a namespace, you generate different types of directories. Some are writable, while others are read-only. In a replicated namespace, there are three types of directories:

- Master
- Secondary
- Read-only

For example, in Figure A–5 there are three copies of the root directory. The master copy resides on node DESIGN. Read-only copies reside on the other two nodes.

Figure A–5 A Namespace with Replicated Directories



In a master directory an application can create or modify all types of entries: objects, directory pointers, and soft links. In a secondary directory an application can create or modify objects and soft links but not directory pointers. An application can retrieve namespace data from any type of directory.

When an application attempts to create a new object or update an existing one, the DNS clerk sends the request to a DNS server that has a secondary or master directory. The request to create an object succeeds as long as no other entry with the same name exists; the request to modify an object succeeds as long as the object is found in the directory.

A.2.6.3 Setting Confidence

An application can use the confidence argument in a $DNS call to stipulate the type of directory that the DNS clerk should use to service the call. For example, when an application wants to create an object, it can force the DNS clerk to create the object in the master directory by stipulating a high confidence level. Otherwise, DNS creates the object either in the master or in a secondary directory.

In a create or modify call, confidence has the following meaning:

- High confidence—Use the master directory.
- Medium confidence—Use the master or a secondary directory. There can be multiple copies of secondary directories.

An application's expression of confidence has a slightly different meaning in a request to find data. In this operation, there are three levels of confidence:

- High confidence—Use the master directory.
- Medium confidence—Use cached information to find the location of a DNS server but get the information from a DNS server.
- Low confidence—Use cached information.

A.2.6.4 Maintaining Consistency in Data

Whenever a directory is modified, the name service attempts to send the updated information to all directory replicas as long as the *convergence* attribute of the directory is set to high. Sometimes it is impossible to deliver the updates to all directory replicas, however, because a network link may be down or a node may be unreachable.

DNS does have a method of ensuring data consistency—it is called a **skulk**. In a skulk, DNS checks to see if data is consistent. If not, it gathers all updates made to a directory since the last skulk and propagates the updates to all replicas of the directory. If there is any discrepancy between information in a master and a secondary directory during a skulk, then the entry with the most recent time-stamping is used. Once the skulk is completed, DNS informs all directories of the time-stamping of the latest universal update.

When the convergence attribute is high, DNS skulks the namespace every 12 hours. When the convergence is low, the skulk occurs every 24 hours.

Directory replicas can lose their consistency between skulks. Two objects of the same name could be created simultaneously in different directory replicas or updates to the namespace might not be seen by all copies immediately. When DNS detects a conflict in replicas, it preserves the object with the most recent update time-stamping and deletes the older object. There is a chance that an application may get information from the namespace that DNS has not synchronized. In this case, an application has to have a mechanism to deal with the inconsistency.

A.2.7 Requesting Information from DNS

Once an application adds its objects to the namespace and modifies the objects to contain any necessary attributes, the application is ready to use the namespace. An application can request that the DNS clerk read 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 entry.

For example, the VAX Distributed File Service (DFS) is a layered product that provides VMS users with the ability to use remote VMS disks as if they were attached to their local VMS system. The DFS application registers VMS directory structures (a directory and all of its subdirectories) with DNS. Each DFS object registered in the namespace names 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 to map DNS names to the appropriate VMS directory structures.

Whenever the DFS application receives the following mount request, DFS sends a request for information to the DNS 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 DNS name that is supplied through the user interface 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 will be a second call to read other attributes of the object

3 Makes a second call to the $DNS service to read the address attribute of the object

4 Sends the DNS name to the DFS server, which looks up the disk where the access point is located

5 Verifies that the DNS name is valid on the DFS server

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

A.2.7.1 Reading Objects

When requesting information from DNS, an application always takes an object name from the user interface, translates the name into opaque format, and passes it in an item list to the DNS clerk.

The following C program 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 $dnsb iosb;             /* Used to determine DNS status */
    char objtype = dns$k_object;   /* Using object entries         */

    struct $dnsitmdef readitem[6]; /* Item list for system service */
    struct dsc$descriptor set_dsc, value_dsc, newset_dsc, uid_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 uidbuf[20];     /* 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 uid_len;  /* Contains length of UID 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   ❸
            {
                set_dsc.dsc$w_length = setlen;
                set_dsc.dsc$a_pointer = &attsetbuf[0]; /* Address of set */

                value_dsc.dsc$w_length = dns$k_simplenamemax;
                value_dsc.dsc$a_pointer = attvalbuf;   /* Buffer to hold  */
                                                       /* attribute value */

                uid_dsc.dsc$w_length = 20;
                uid_dsc.dsc$a_pointer = uidbuf; /* Buffer to hold value's UID*/
                newset_dsc.dsc$w_length = dns$k_maxattribute;
                newset_dsc.dsc$a_pointer = &attsetbuf[0]; /* Same buffer for */
                                                          /* each call      */

                set_status = dns$remove_first_set_value(&set_dsc, &value_dsc,
                             ❹                    &val_len, &uid_dsc,
                                                        &uid_len, &newset_dsc,
                                                        &setlen);

                if(set_status == SS$_NORMAL)
                {   ❺
                    readitem[4].dns$w_itm_code = dns$_contextvartime;
                    readitem[4].dns$w_itm_size = uid_len;
                    readitem[4].dns$a_itm_address = uidbuf;

                    *((int *)&readitem[5]) = 0;

                    printf("\tValue: ");   ❻
                    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 namespace entry 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 zero 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 This loop extracts all values from the set returned by $DNSW, one value at a time. This routine sets up descriptors for buffers that are used by the DNS$REMOVE_FIRST_SET_VALUE routine to extract values from the set. The loop executes until all values are extracted from the set or it encounters an error.

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

❺ This attribute name might be the context the routine uses to read additional attributes. The attribute's UID, not its value, provides the context.

❻ 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.)

A.2.7.2 Listing Information

The list functions of $DNS allow applications to list the objects, subdirectories, or soft links in a specific directory. Either the asterisk (*) or question mark (?) wildcard, described in Section A.2.3.3, allows an application to screen on the basis of its facility name.

The values DNS returns from read or enumerate functions are in different structures. For example, an enumeration of objects returns different structures than an enumeration of directories.

The following C program shows how an application can read the objects in a directory with the $DNS system service. It demonstrates how you parse any set that the enumerate objects function returns with a run-time routine in order to remove the first entry 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;       /* DNS 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. */
                                                 ❺
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 RTL 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$_NORMAL)
        {                                     ❻
            cvtitem[0].dns$w_itm_code = dns$_fromsimplename;
            cvtitem[0].dns$w_itm_size = val_len;
            cvtitem[0].dns$a_itm_address = objnamebuf;

            cvtitem[1].dns$w_itm_code = dns$_tostringname;
            cvtitem[1].dns$w_itm_size = dns$k_simplestrmax;
            cvtitem[1].dns$a_itm_address = simplebuf;
            cvtitem[1].dns$a_itm_ret_length = &sname_len;

            *((int *)&cvtitem[2]) = 0;

            status = sys$dnsw(0, dns$_simple_opaque_to_string, &cvtitem,
                              &iosb, 0, 0);

            if(status == SS$_NORMAL)
                status = iosb.dns$l_dnsb_status;  /* Check for errors */

            if(status != SS$_NORMAL) /* If error, terminate processing */
            {
                printf("Converting object name to string returned %d\n",
                       status);
                exit(status);
            }
            else
            {
                simplebuf[sname_len] = 0;   /* Null terminate for printing */
                printf("%s\n", 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 DNS 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 zero 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.

A.2.7.3 How the Clerk Locates Data

When the DNS clerk receives an application's call for service, it tries to find a DNS server that can process the request.

Often, the DNS clerk does not know which DNS server holds the object information. To find an unknown server, the clerk looks in its own cache first. The clerk cache holds namespace information gathered from servicing earlier application requests. If the clerk cache does not list the needed server, then the DNS clerk requests information from a local DNS server in its cache. (A clerk always knows about at least one DNS server because this information is loaded at system startup.)

The clerk's last recourse is to trace directory pointers through the namespace. Any DNS server is capable of telling the clerk about another DNS server holding other directories in the namespace hierarchy. The clerk follows directory pointers until it finds a DNS server holding the specified directory. If the clerk cannot find the specified directory, then it follows directory pointers up to the root directory. Once the root directory is found, the clerk traces directory pointers away from the root, until it finds a DNS server that has the directory holding the requested object.

Once the clerk finds a directory that holds the required information, it delivers the request to the DNS server. As soon as the clerk receives a response, it transmits the result to the application.

A.2.8 DNS System Services

The Distributed Name Service Clerk system services are the programming interface to the VAX Distributed Name Service facility. The DNS Clerk system services allow an application to register a resource in a distributed database and then access the resource from any point in the network by a single name. There are two system service calls to the clerk that are described in this section.

- $DNS (Distributed Name Service Clerk)
- $DNSW (Distributed Name Service Clerk and Wait)

The $DNS system service is the asynchronous client interface for applications using the Distributed Name Service. The $DNSW system service is the synchronous client interface.

$DNS Distributed Name Service Clerk

The Distributed Name Service Clerk service registers a resource in a distributed database. The $DNS service completes asynchronously; that is, it returns to the client immediately after making a name service call. The status returned to the client call indicates whether a request was successfully queued to the name service.

Note that the Distributed Name Service Clerk and Wait ($DNSW) call is the synchronous equivalent of $DNS. $DNSW is identical to $DNS in every way except that $DNSW returns to the caller after the operation completes.

FORMAT

SYS$DNS *[efn] ,func ,itmlst ,[dnsb] ,[astadr] ,[astprm]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services return by immediate value a condition value in R0. Condition values returned by this call are listed in the section Condition Values Returned. Errors returned here are from the DNS clerk. Refer to the **dnsb** argument for errors returned by the name service.

ARGUMENTS

efn

VMS usage:	**ef_number**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Number of the event flag to be set when $DNS completes. The **efn** argument is a longword containing this number. The **efn** argument is optional; if not specified, event flag 0 is set.

When $DNS begins execution, it clears the event flag. Even if the service encounters an error and completes without queuing a name service request, the specified event flag is set.

func

VMS usage:	**function_code**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Function code specifying the action that $DNS is to perform. The **func** argument is a longword containing this function code.

A single call to $DNS can specify one function code. Most function codes require or allow for additional information to be passed in the call with the **itmlst** argument.

$DNS Function Codes

DNS$_CREATE_OBJECT

This request creates an object in the namespace. Initially, the entry has the attributes of DNSUID, DNSUTS, DNS$CLASS, DNS$ACS, and DNS$CLASSVERSION. The name service creates the DNSUID, DNSUTS, and DNS$ACS attributes. The client application supplies the DNS$CLASS and DNS$CLASSVERSION attributes. You can add additional attributes using the DNS$_MODIFY_ATTRIBUTE function.

The DNS clerk cannot guarantee that an object has been created. Another DNS$_CREATE_OBJECT request could supersede the object created by your call. To verify an object creation, wait until the directory is skulked and then check to see if the requested object entry is present. If the value of the directory's DNS$ALLUPTO attribute is greater than the UID of the object entry, your object entry has been successfully created.

Creating an object in the namespace requires write access to the directory in which the object will reside.

If specified, DNS$_OUTUID holds the UID of the created object.

You must specify the following item codes:

DNS$_CLASS (Class_Name)
DNS$_OBJECTNAME (Opaque_Full_Name)
DNS$_VERSION (Class_Version)

You can specify the following input item codes:

DNS$_CONF
DNS$_WAIT

You can specify the following output item code:

DNS$OUTUID (UID)

$DNS returns the following:

SS$_NORMAL
DNS$_ENTRYEXISTS
DNS$_INVALID_OBJECTNAME
DNS$_INVALID_CLASSNAME
Any condition listed in the section Condition Values Returned.

$DNS returns the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_DELETE_OBJECT

This request removes the specified object from the namespace. The function requires delete access to the object in question.

You must specify the following input item code:

DNS$_OBJECTNAME (Opaque_Full_Name)

You can specify the following input item codes:

DNS$_CONF
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_INVALID_OBJECTNAME
Any condition listed in the section Condition Values Returned.

$DNS returns the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_ENUMERATE_ATTRIBUTES

This request returns a set of attributes in DNS$_OUTATTRIBUTESET that is associated with the entry. The entry type is specified in the DNS$_LOOKINGFOR entry.

To manipulate the values returned by this call, use the DNS$REMOVE_FIRST_SET_VALUE run-time routine. The values returned are the Enum_Att_Name structure, which is described in Table A–1.

You must have read access to the entry to enumerate its attributes.

The DNS clerk enumerates attributes in alphabetical order. A return status of DNS$_MOREDATA implies that not all attributes have been enumerated. You should make further calls, setting DNS$_CONTEXTVARNAME to the last attribute in the set returned, until the procedure returns SS$_NORMAL.

You must specify the following input item codes:

DNS$_ENTRY (Opaque_Full_Name)
DNS$_LOOKINGFOR (Entry_Type)

You must specify the following output item code:

DNS$_OUTATTRIBUTESET (set of Enum_Att_Name)

You can specify any of the following input item codes:

DNS$_CONF
DNS$_CONTEXTVARNAME (Opaque_Simple_Name)
DNS$_WAIT

$DNS can return the following:

SS$_NORMAL
DNS$_MOREDATA
DNS$_INVALID_ENTRYNAME
DNS$_INVALID_CONTEXTNAME
Any condition listed in the section Condition Values Returned.

$DNS returns the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_ENUMERATE_CHILDREN

This request takes as input a directory name with an optional simple name that uses a wildcard. The DNS clerk matches the input against child directory entries in the specified directory.

The DNS clerk returns a set of simple names of child directories in the target directory that match the name with the wildcard. A null set is returned when there is no match or when the directory has no children.

To manipulate the values returned by this call, use the DNS$REMOVE_FIRST_SET_VALUE run-time routine. The value returned is a simple name.

The function requires read access to the parent directory.

The child directories are enumerated in alphabetical order. If the call returns DNS$_MOREDATA, not all children have been enumerated and the client should make further calls, setting DNS$_CONTEXTVARNAME to the last child directory in the set returned, until the procedure returns SS$_NORMAL. Subsequent calls return the child directories, starting with the directory specified in DNS$_CONTEXTVARNAME and continuing in alphabetical order.

You must specify the following input item code:

DNS$_DIRECTORY (Opaque_Full_Name)

You must specify the following output item code:

DNS$_OUTCHILDREN (set of Opaque_Simple_Name)

You can specify the following input item codes:

DNS$_CONF
DNS$_CONTEXTVARNAME (Opaque_Simple_Name)
DNS$_WAIT
DNS$_WILDCARD (Opaque_Simple_Name)

$DNS returns the following:

SS$_NORMAL
DNS$_MOREDATA
DNS$_INVALID_DIRECTORYNAME
DNS$_INVALID_CONTEXTNAME
DNS$_INVALID_WILDCARDNAME

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_ENUMERATE_OBJECTS

This request takes as input the directory name, a simple name that uses a wildcard, and a class name that uses a wildcard. The DNS clerk matches these against objects in the directory. If a wildcard and class filter are not specified, then all objects in the directory are returned.

The function returns (in DNS$_OUTOBJECTS) a set of simple names of objects in the directory that match the name with the wildcard. If no objects match the wildcard or the directory contains no objects, a null set is returned. The DNS clerk returns DNS$V_DNSB_OUTLINKED qualifying status if it encounters one or more soft links in resolving the names of object entries to be enumerated.

To manipulate the values returned by this call, use the DNS$REMOVE_FIRST_SET_VALUE run-time routine. The value returned is a simple name structure.

This function requires read access to the parent directory.

The objects are enumerated in alphabetical order. If the call returns DNS$_MOREDATA, not all objects have been enumerated and the client should make further calls, setting DNS$_CONTEXTVARNAME to the last object in the set returned, until the procedure returns SS$_NORMAL. If the class filter is specified, only those objects of the specified classes are returned.

You must specify the following input item code:

DNS$_DIRECTORY (Opaque_Full_Name)

You must specify the following output item code:

DNS$_OUTOBJECTS (set of Opaque_Simple_Names)

You can specify any of the following input item codes:

DNS$_WILDCARD (Opaque_Simple_Name)
DNS$_CLASSFILTER (Opaque_Simple_Name)
DNS$_CONTEXTVARNAME (Opaque_Simple_Name)
DNS$_CONF
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_MOREDATA
DNS$_INVALID_DIRECTORYNAME
DNS$_INVALID_CONTEXTNAME
DNS$_INVALID_WILDCARDNAME
DNS$_INVALID_CLASSNAME

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_ENUMERATE_SOFTLINKS

This request takes as input the name of a directory and a wildcarded simple name. The DNS clerk matches these against soft links in the directory. It returns (in DNS$_OUTSOFTLINKS) a set consisting of simple names of soft links in the directory that match the wildcarded name. If no soft link entries match the wildcard or the directory contains no soft links, a null set is returned.

If no wildcard is specified, then all soft links in the directory are returned.

To manipulate the values returned by this call, use the DNS$REMOVE_FIRST_SET_VALUE run-time routine. The value returned is a simple name.

This function requires read access to the parent directory.

The soft links are enumerated in alphabetical order. If the call returns DNS$_MOREDATA, not all matching soft links have been enumerated and the client should make further calls, setting DNS$_CONTEXTVARNAME to the last soft link in the set returned, until the procedure returns SS$_NORMAL.

You must specify the following input item code:

DNS$_DIRECTORY (Opaque_Full_Name)

You must specify the following output item code:

DNS$_OUTSOFTLINKS (set of Opaque_Simple_Name)

You can specify the following input item codes:

DNS$_WILDCARD (Opaque_Simple_Name)
DNS$_CONTEXTVARNAME (Opaque_Simple_Name)
DNS$_CONF
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_INVALID_DIRECTORYNAME
DNS$_INVALID_CONTEXTNAME
DNS$_INVALID_WILDCARDNAME

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_FULL_OPAQUE_TO_STRING

This request converts a full name in opaque format to its equivalent in string format, as described in Section A.2.2.4. Setting the byte referred to by DNS$_SUPPRESS_NSNAME to 1 prevents the namespace name from being included in the string name.

You must specify the following item codes:

DNS$_FROMFULLNAME (Opaque_Full_Name)
DNS$_TOSTRINGNAME (Full_Name_Str)

You can specify the following input item code:

DNS$_SUPPRESS_NSNAME (byte)

$DNS returns the following:

SS$_NORMAL
DNS$_INVALIDNAME

You do not receive qualifying status.

DNS$_MODIFY_ATTRIBUTE

This request applies one update to the specified entry in the namespace. You can add or remove an attribute; you can add or remove a value from either a single-value attribute or a set-valued attribute.

This operation requires write or delete access to the entry whose attribute is being modified, depending on whether the operation adds or removes the attribute.

When adding a value to a single-value attribute, include a value in DNS$_MODVALUE or you will receive the error DNS$_INVALIDUPDATE. The item code DNS$_MODVALUE is not required when writing to an attribute set because the name service creates the attribute if no value is provided.

In a delete operation, include the DNS$_MODVALUE item code to remove a certain value from an attribute set. Unless you specify the item code, the name service removes the attribute and all its values from the entry.

You must specify the following item codes:

DNS$_ENTRY (Opaque_Full_Name)
DNS$_LOOKINGFOR (Entry_Type)
DNS$_MODOPERATION (DNS$K_PRESENT or DNS$K_ABSENT)
DNS$_ATTRIBUTETYPE (DNS$K_SET or DNS$K_SINGLE)
DNS$_ATTRIBUTENAME (Opaque_Simple_Name)

You can specify the following input item codes:

DNS$_CONF
DNS$_MODVALUE
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_WRONGATTRIBUTETYPE
DNS$_INVALIDUPDATE
DNS$_INVALID_ENTRYNAME
DNS$_INVALID_ATTRIBUTENAME

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_PARSE_FULLNAME_STRING

This request takes a full name in string format and converts it to its equivalent in opaque format. If DNS$_NEXTCHAR_PTR is used, the longword referenced by this entry contains the address of the character immediately following the DNS name given in DNS$_FROMSTRINGNAME.

You must specify the following item codes:

DNS$_FROMSTRINGNAME (Full_Name_Str)
DNS$_TOFULLNAME (Opaque_Full_Name)

You can specify the following input item code:

DNS$_NEXTCHAR_PTR

$DNS can return the following:

SS$_NORMAL
DNS$_INVALIDNAME

You do not receive qualifying status.

DNS$_PARSE_SIMPLENAME_STRING

This request takes a simple name in string format and converts it to its equivalent in opaque format. If DNS$_NEXTCHAR_PTR is used, the longword referenced by this entry contains the address of the character immediately following the DNS name given in DNS$_FROMSTRINGNAME.

You must specify the following item codes:

DNS$_FROMSTRINGNAME (Simple_Name_Str)
DNS$_TOFULLNAME (Opaque_Simple_Name)

You can specify the following input item code:

DNS$_NEXTCHAR_PTR

$DNS can return the following:

SS$_NORMAL
DNS$_INVALIDNAME

You do not receive qualifying status.

DNS$_READ_ATTRIBUTE

This request returns (in DNS$_OUTVALSET) a set whose members are the values of the specified attribute. When the request completes successfully, the qualifying status indicates whether soft links were followed in resolving the name.

This function requires read access to the object whose attribute is to be read.

To manipulate the values returned by this call, use the DNS$REMOVE_FIRST_SET_VALUE run-time routine. The contents of DNS$_OUTVALSET are passed to DNS$REMOVE_FIRST_SET_VALUE, and the routine returns the value of the attribute.

The attribute values are returned in the order they were received. If the call returns DNS$_MOREDATA, not all values have been returned. The client application can make further calls, setting DNS$_CONTEXTVARTIME to the time-stamping of the last attribute in the set returned, until the procedure returns SS$_NORMAL. If the client sets the DNS$_MAYBEMORE argument to 1, the name service attempts to make subsequent DNS$_READ_ATTRIBUTE calls for the same entry more efficient. The client may set this argument to true on any call, but performance improves only if the client accesses no other entry before making a read attribute call for the previous entry.

You must include the following input item codes:

DNS$_ENTRY (Opaque_Full_Name)
DNS$_LOOKINGFOR (Entry_Type)
DNS$_ATTRIBUTENAME (Opaque_Simple_Name)

You must include the following output item code:

DNS$_OUTVALSET (set of values)

You can include the following input item codes:

DNS$_MAYBEMORE (Boolean)

DNS$_CONTEXTVARTIME (UID)
DNS$_CONF
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_MOREDATA
DNS$_INVALID_ENTRYNAME
DNS$_INVALID_ATTRIBUTENAME

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_RESOLVE_NAME

This request follows a chain of soft links to its destination, returning the full name of that entry so that future calls by the client application can use the entry name without incurring the overhead of following the link.

This function requires read access to each of the soft links in the chain.

Applications that maintain their own databases of opaque DNS names should use DNS$_RESOLVE_NAME any time they receive the qualifying status DNS$V_DNSB_OUTLINKED. This status indicates a need to update the current name, using the soft link facility of DNS. Use the original name with DNS$_RESOLVE_NAME and store the result in the application database.

If the application provides a name that does not contain any soft links, DNS$_NOTLINKED status is returned. If the target of any of the chain of soft links followed does not exist, the DNS$_DANGLINGLINK status is returned. To obtain the target of any particular soft link, use the DNS$_READ_ATTRIBUTE function with DNS$_LOOKINGFOR set to DNS$K_SOFTLINK and request the attribute DNS$LINKTARGET. This can be useful in discovering which link in a chain is "broken." If the DNS clerk detects a loop, it returns DNS$_POSSIBLECYCLE status.

You must specify the following input item code:

DNS$_LINKNAME (Opaque_Full_Name)

You must specify the following output item code:

DNS$_OUTNAME (Opaque_Full_Name)

You can specify the following input item codes:

DNS$_CONF
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_INVALID_LINKNAME
DNS$_NOTLINKED

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_SIMPLE_OPAQUE_TO_STRING

This request takes a simple name in opaque format and converts it to its equivalent in string format, as described in Section A.2.2.4.

You must specify the following item codes:

DNS$_FROMSIMPLENAME (Opaque_Simple_Name)
DNS$_TOSTRINGNAME (Simple_Name_Str)

$DNS returns the following:

SS$_NORMAL
DNS$_INVALIDNAME

You do not receive qualifying status.

DNS$_TEST_ATTRIBUTE

This request returns DNS$_TRUE if the specified attribute has one of the following characteristics:

- It is a single-value attribute and its value matches the client-specified value.
- It is a set-valued attribute and the attribute contains the client-specified value as one of its members.

On successful completion of the function, DNS$V_DNSB_OUTLINKED indicates whether soft links were followed in resolving the name.

This function requires test or read access to the entry whose attribute is to be tested.

If the attribute is not present in the entry or if the requested attribute does not exist, the function returns
DNS$_FALSE.

You must specify the following item codes:

DNS$_ENTRY (Opaque_Full_Name)
DNS$_LOOKINGFOR (Entry_Type)
DNS$_ATTRIBUTENAME (Opaque_Simple_Name)
DNS$_VALUE (value)

You can specify the following input item codes:

DNS$_CONF
DNS$_WAIT

$DNS returns the following when the call is successful:

DNS$_TRUE
DNS$_FALSE

$DNS returns the following when the call is unsuccessful:

DNS$_INVALID_ENTRYNAME
DNS$_INVALID_ATTRIBUTENAME

You might receive the following qualifying status:

DNS$V_DNSB_OUTLINKED

DNS$_TEST_GROUP

This request tests for group membership. It returns DNS$_TRUE if the specified member is a member of the specified group (or a subgroup thereof), and DNS$_FALSE otherwise. If a recursive search is required and one or more of the subgroups is unavailable, the status encountered in trying to access that group is returned.

The DNS$_INOUTDIRECT argument, on input, controls the scope of the search. If set to true, the only group considered is the top level group specified by the group argument. If set to false, recursive evaluation is performed. On output, the DNS$_INOUTDIRECT argument is set to 1 if the member was found in the top level group; otherwise it is set to 0.

You must specify the following item codes:

DNS$_GROUP (Opaque_Full_Name)
DNS$_MEMBER (Opaque_Full_Name)

You can specify the following input item codes:

DNS$_CONF
DNS$_INOUTDIRECT (Boolean)
DNS$_WAIT

$DNS returns the following:

SS$_NORMAL
DNS$_NOTAGROUP
DNS$_INVALID_GROUPNAME
DNS$_INVALID_MEMBERNAME

You might receive the following qualifying status:

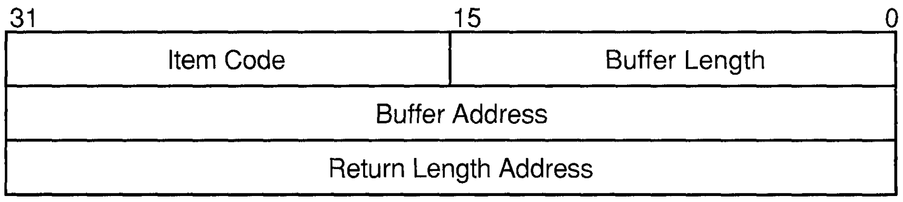
DNS$V_DNSB_INOUTDIRECT

itmlst

VMS usage:	**item_list_3**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Item list supplying information to be used in performing the function specified by the **func** argument. The **itmlst** argument is the address of the item list. The item list consists of one or more item descriptors, each of which is three longwords. The descriptors can be in any order in the item list. Each item descriptor specifies an item code. Each item code either describes the specific information to be returned by $DNS or otherwise affects the action designated by the function code. The item list is terminated by a longword of zero.

The item list is a standard VMS format item list. The following figure depicts the general structure of an item descriptor:



ZK–1705–GE

$DNS Item Descriptor Fields

item code

A word containing a symbolic code describing the nature of the information currently in the buffer or to be returned in the buffer. The location of the buffer is pointed to by the buffer address field. Each item code has a symbolic name; these symbolic names are defined by the $DNS macro and have the format DNS$_code.

buffer length

A word specifying the length of the buffer; the buffer either supplies information to be used by $DNS or receives information from $DNS. The required length of the buffer varies depending on the item code specified; each item code description specifies the required length.

buffer address

A longword containing the address of the buffer that specifies or receives the information.

return length address

A longword containing the address of a word specifying the actual length in bytes of the information returned by $DNS. The information resides in a buffer identified by the buffer address field. The field applies to output item list entries only and must be zero for input entries. If the return length address is 0, it is ignored.

$DNS Item Codes

DNS$_ATTRIBUTETYPE

The DNS$_ATTRIBUTETYPE item code specifies whether an attribute is set valued (DNS$K_SET) with a value of 3 or single valued (DNS$K_ SINGLE) with a value of 2.

DNS$_ATTRIBUTENAME

The DNS$_ATTRIBUTENAME item code specifies the opaque simple name of an attribute. An attribute name cannot be longer than 31 characters.

DNS$_CLASS

The DNS$_CLASS item code specifies the class of an object for the $DNS function DNS$_CREATE_OBJECT. DNS$_CLASS is an opaque simple name.

DNS$_CLASSFILTER

DNS$_CLASSFILTER is used by the $DNS function DNS$_ENUMERATE_OBJECTS to limit the scope of the enumeration to those objects belonging to a certain class (or, if a wildcard name is used, a group of classes). DNS$_CLASSFILTER is an opaque simple name, which can use a wildcard.

DNS$_CLASSFILTER is optional. A wildcard simple name of * is used by default, meaning that objects of all classes will be enumerated.

DNS$_CONF

DNS$_CONF specifies for $DNS the level of importance in returning up-to-date information. DNS$_CONF is 1 byte long and can take one of the following values:

Confidence Level	Value	Description
DNS$K_LOW	1	Service the DNS clerk request at the lowest cost, usually from cached information.
DNS$K_MEDIUM	2	Bypass any cached information and obtain the data directly from a name server.
DNS$K_HIGH	3	Service the request from a master directory.

The entry is optional; if it is not specified, the DNS clerk assumes a value of DNS$K_LOW.

DNS$_CONTEXTVARNAME

DNS$_CONTEXTVARNAME is used by the enumeration functions of $DNS to specify a context from which the enumeration is to begin. The item is an opaque simple name.

DNS$_CONTEXTVARNAME is optional. If not given, the enumeration begins with the first element.

DNS$_DIRECTORY

DNS$_DIRECTORY is used by most of the enumeration functions of $DNS to specify the namespace directory in which the elements of the enumeration are to be found. DNS$_DIRECTORY is an opaque full name.

DNS$_ENTRY

DNS$_ENTRY specifies for $DNS the opaque full name of a namespace entry (object, soft link, directory, clearinghouse).

DNS$_FROMFULLNAME

DNS$_FROMFULLNAME specifies for the DNS$_FULL_OPAQUE_TO_STRING function the opaque full name that is to be converted into string format.

DNS$_FROMSIMPLENAME

DNS$_FROMSIMPLENAME specifies for the DNS$_SIMPLE_OPAQUE_TO_STRING function the opaque simple name that is to be converted into string format.

DNS$_FROMSTRINGNAME

DNS$_FROMSTRINGNAME specifies a name in string format for the parse functions DNS$_PARSE_FULLNAME_STRING and DNS$_PARSE_SIMPLENAME_STRING that is to be converted to opaque format.

DNS$_GROUP

DNS$_GROUP specifies for the DNS$_TEST_GROUP function the opaque full name of the group that is to be tested. DNS$_GROUP must be the name of a group object.

DNS$_INOUTDIRECT

DNS$_INOUTDIRECT is a Boolean value that serves two different purposes for the DNS$_TEST_GROUP function. On input, DNS$_INOUTDIRECT controls the scope of the search for the test, as follows:

Value	Definition
1	The only group to be tested is the top level group specified by the DNS$_GROUP item.
0	All subgroups of the group named in DNS$_GROUP are tested for inclusion. A subgroup is any member that is a group in itself.

On output, DNS$_INOUTDIRECT is set to indicate whether the members were found in the top level group or were found as members of one of the subgroups, as follows:

Value	Definition
1	The member was found in the top level group.
0	The member was found in one of the subgroups of the top level group.

DNS$_INOUTDIRECT is a single-byte value.

DNS$_LINKNAME

DNS$_LINKNAME specifies the opaque full name of a soft link.

DNS$_LOOKINGFOR

DNS$_LOOKINGFOR specifies the type of entry on which the call is to operate. DNS$_LOOKINGFOR, which is encoded as a byte, can take one of the following values:

Type of Entry	Value
DNS$K_DIRECTORY	1
DNS$K_OBJECT	2
DNS$K_CHILDDIRECTORY	3
DNS$K_SOFTLINK	4
DNS$K_CLEARINGHOUSE	5

DNS$_MAYBEMORE

DNS$_MAYBEMORE is used with the DNS$_READ_ATTRIBUTE function to indicate that the results of the read operation are to be cached. This is a single-byte item.

When this item is set to 1, the name service returns as much information about the attributes for the entry as it is able to fit in the return buffer. All of this information is cached to make later lookups of attribute information for the entry quicker and more efficient.

If this item is not supplied, then only the requested information for the entry is returned.

DNS$_MEMBER

DNS$_MEMBER specifies for the DNS$_TEST_GROUP function of $DNS the opaque full name of a member that is to be tested for inclusion within a given group.

DNS$_MODOPERATION

DNS$_MODOPERATION specifies for the DNS$_MODIFY_ATTRIBUTE function the type of operation that is to take place. There are two types of modifications: adding an attribute (DNS$K_PRESENT), which has a value of 1, or deleting an attribute (DNS$K_ABSENT), which has a value of 0.

The name service adds an attribute in the following way:

- For an existing attribute where an attribute value is given, the value is added to a set-valued attribute and all other values for the set are unaffected. The value replaces any previous value in a single-value attribute.
- For an existing attribute where an attribute value is not given, all previous values for the attribute are unaffected.
- For a new attribute
  - Where an attribute is given, the attribute is created and given the attribute type of DNS$K_SET or DNS$K_SINGLE as supplied with the DNS$K_ATTRIBUTETYPE item. The value is assigned to the attribute.
  - Where an attribute value is not given, a set-valued attribute is created without a value assignment, but a single-value attribute is *not* created.

The name service deletes an attribute in the following way:

- If the attribute exists and an attribute value is given, the attribute value is removed from a set-valued attribute. All other values are unaffected. For a single-value attribute, the attribute (along with its value) is removed from the entry.
- If an attribute value is not given, then the attribute and all values of the attribute are removed. This is true for both set-valued attributes and single-value attributes.

DNS$_MODVALUE

DNS$_MODVALUE specifies for the DNS$_MODIFY_ATTRIBUTE function the value that is to be added to or deleted from an attribute. The structure of this value is dependent on the application.

DNS$_MODVALUE is an optional argument that affects the overall operation of the DNS$_MODIFY_ATTRIBUTE function. (See the DNS$_MODOPERATION item code description for more information.)

DNS$_NEXTCHAR_PTR

DNS$_NEXTCHAR_PTR is an optional item code that can be used with the parse functions DNS$_PARSE_FULLNAME_STRING and DNS$_PARSE_SIMPLENAME_STRING to return the address of the character that immediately follows a valid DNS name. This option is most useful when applications are parsing command line strings.

Without this item code, the parse functions return an error if any portion of the name string is invalid.

DNS$_OBJECTNAME

DNS$_OBJECTNAME specifies the opaque full name of an object.

DNS$_OUTATTRIBUTESET

DNS$_OUTATTRIBUTESET specifies to the DNS$_ENUMERATE_ATTRIBUTES function the address of a buffer that is to contain the set of enumerated attribute names.

The names returned in this set can be extracted from the buffer with the DNS$REMOVE_FIRST_SET_VALUE routine. The resulting values are contained in the $DNSATTRSPECDEF structure, a byte indicating whether an attribute is set-value or single-value followed by an opaque simple name.

DNS$_OUTNAME

DNS$_OUTNAME specifies for the DNS$_RESOLVE_NAME function the address of a buffer that is to contain the opaque full name of the namespace entry that is pointed to by a soft link.

DNS$_OUTOBJECTS

DNS$_OUTOBJECTS specifies for the DNS$_ENUMERATE_OBJECTS function the address of a buffer that is to contain the set of opaque simple names returned by the enumeration.

The values resulting from the enumeration can be extracted using the DNS$REMOVE_FIRST_SET_VALUE routine. The resulting values are the opaque simple names of the objects found in the directory.

DNS$_OUTCHILDREN

DNS$_OUTCHILDREN specifies for the DNS$_ENUMERATE_CHILDREN function the address of a buffer that is to contain the set of opaque simple names returned by the enumeration.

The values resulting from the enumeration can be extracted using the DNS$REMOVE_FIRST_SET_VALUE routine. These values are the opaque simple names of the child directories found in the parent directory.

DNS$_OUTSOFTLINKS

DNS$_OUTSOFTLINKS specifies for the DNS$_ENUMERATE_SOFTLINKS function the address of a buffer that is to contain the set of opaque simple names returned by the enumeration.

The values resulting from the enumeration can be extracted using the DNS$REMOVE_FIRST_SET_VALUE routine. The resulting values are the opaque simple names of the soft links found in the directory.

DNS$_OUTVALSET
DNS$_OUTVALSET specifies for the DNS$_READ_ATTRIBUTE function the address of a buffer that is to contain the set of values for the given attribute.

The values of the set placed in this buffer can be extracted using the DNS$REMOVE_FIRST_SET_VALUE routine. The extracted values are the values of the attribute.

DNS$_OUTUID
DNS$_OUTUID is an optional item code that contains the address of a buffer used by the create functions of $DNS to return the unique identifier (UID). The UID is the time-stamping the entry received at creation.

DNS$_SUPPRESS_NSNAME
DNS$_SUPPRESS_NSNAME is an optional item for the DNS$_FULL_OPAQUE_TO_STRING function that is used to indicate that the leading namespace name should not be returned in the converted full name string. This is a single-byte value.

A value of 1 suppresses the leading namespace name in the resulting full name string.

DNS$_TOFULLNAME
DNS$_TOFULLNAME specifies for the DNS$_PARSE_FULLNAME_STRING function the address of a buffer that will contain the resulting opaque full name.

DNS$_TOSIMPLENAME
DNS$_TOSIMPLENAME specifies for the DNS$_PARSE_SIMPLENAME_STRING function the address of a buffer that will contain the resulting opaque simple name.

DNS$_TOSTRINGNAME
DNS$_TOSTRINGNAME specifies the address of a buffer that is to contain the string name resulting from one of the conversion functions: DNS$_FULL_OPAQUE_TO_STRING or DNS$_SIMPLE_OPAQUE_TO_STRING.

DNS$_VALUE
DNS$_VALUE specifies for the DNS$_TEST_ATTRIBUTE function the value that is to be tested. This item contains the address of a buffer holding the value.

DNS$_VERSION
DNS$_VERSION specifies for the DNS$_CREATE_OBJECT function the version associated with an object. This item contains the address of a $DNSCVERSDEF (CLASSVERSION) structure. This is a 2-byte structure: the first byte contains the major version number; the second contains the minor version number.

DNS$_WAIT
DNS$_WAIT enables the client to specify a timeout value to wait for a call to complete. If the timeout expires, the call returns either DNS$K_TIMEOUTNOTDONE or DNS$K_TIMEOUTMAYBEDONE, depending on whether the name space was updated by the incomplete operation.

The $BINTIM service converts an ASCII string time value to the quadword time value required by $DNS.

The parameter is optional; if it is not specified, a system-defined default timeout value of 10 minutes is assumed.

DNS$_WILDCARD
DNS$_WILDCARD is an optional item code that specifies to the enumeration functions of $DNS the opaque simple name used to limit the scope of the enumeration. (The simple name does not have to use a wildcard.) Only those simple names that match the wildcard are returned by the enumeration.

Item Code Identifiers

The identifiers shown in Table A–1 are data structures that are used in item code arguments. Each data structure defines the encoding of an item list element.

Table A–1 DNS Item Code Arguments

Item Code Identifier	Description
Attribute_Name	The structure of an opaque simple name, limited to 31 ISO Latin 1 characters.
Attribute_Name_Str	An attribute name string with the structure of a simple name string but limited to 31 ISO Latin 1 characters.
Boolean	A 1-byte field with the value 0 if false and 1 if true.
Class_Name	An opaque simple name, limited to 31 ISO Latin 1 characters.
Class_Name_Str	A simple name string, limited to 31 ISO Latin 1 characters.
Class_Version	A 2-byte field specifying major and minor version numbers associated with the object class.
Confidence	A 1-byte field with the value: DNSK_LOW, DNSK_MEDIUM, or DNS$K_HIGH.
Entry_Type	A 1-byte field with the value DNSK_OBJECT, DNSK_SOFTLINK, DNS$K_DIRECTORY, or DNS$K_CLEARINGHOUSE.
Enum_Att_Name	A structure consisting of a single byte, indicating whether the attribute is a set (DNSK$_SET) or a single value (DNS$K_SINGLE), followed by an opaque simple name.

(continued on next page)

Table A–1 (Cont.) DNS Item Code Arguments

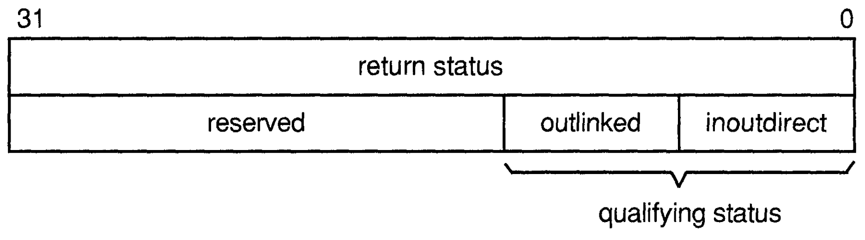
Item Code Identifier	Description
Full_Name_String	A full name string with the following structure: [NS_name:] [.] Namestring [.Namestring] NS_name:, if present, is a local system representation of the NSUID, the unique identifier of the name server. The DNS clerk supplies a namespace name (*node-name*_NS) if the value is omitted. Namestring represents a simple name component. Multiple simple names are separated by periods. You can include the asterisk wildcard (*) and simple name strings within quotation marks.
Group_Member	A structure consisting of a single byte, indicating whether the entry is a principal (DNS$K_GRPMEM_NOT_GROUP) or another group (DNS$K_GRPMEM_IS_GROUP), followed by the opaque full name of the member.
Opaque_Full_Name	The internal format of the complete name identifier for an object. The maximum output of DNS$PARSE_FULLNAME_STRING is 402 bytes.
Opaque_Simple_Name	A simple name specifies the internal format of one component of an Opaque_Full_Name. The Opaque_Simple_Name is the output of the DNS$PARSE_SIMPLENAME_STRING routine. It can be no longer than 257 bytes.
Simple_Name_Str	One term consisting of a string of ASCII characters with its length stored separately in an item list.

dnsb

VMS usage:	**dns_status_block**
type:	**quadword (unsigned)**
access:	**write only**
mechanism:	**by reference**

Status block to receive the final completion status of the $DNS operation. The **dnsb** argument is the address of the quadword $DNS status block.

The following figure depicts the structure of a $DNS status block:



ZK-1080A-GE

Status Block Fields

return status
Set on completion of a DNS clerk request to indicate the success or failure of the operation. Check the qualifying status word for additional information about a request marked as successful. Wherever possible, each function code description includes return status values.

qualifying status
This field consists of a set of flags that provide additional information about a successful name service operation. Wherever possible, each function code description includes qualifying status values.

The qualifying status values are defined as follows:

- DNS$V_DNSB_INOUTDIRECT—If true, indicates only the top level group was seached for a member.
- DNS$V_DNSB_OUTLINKED—If set, indicates that one or more soft links were encountered while resolving the object of the call.

astadr

VMS usage:	**ast_procedure**
type:	**procedure entry mask**
access:	**call without stack unwinding**
mechanism:	**by reference**

Asynchronous system trap (AST) routine to be executed when I/O completes. The **astadr** argument, which is the address of a longword value, is the entry mask to the AST routine.

The AST routine executes in the access mode of the caller of $DNS.

astprm

VMS usage:	**user_arg**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by value**

Asynchronous system trap (AST) parameter passed to the AST service routine. The **astprm** argument is a longword value containing the AST parameter.

DESCRIPTION

The VMS Distributed Name Service Clerk system service provides a low-level interface between an application (client) and the VAX Distributed Name Service. The DNS clerk interface is used to create, delete, modify, and retrieve objects or soft links in a namespace.

A single system service call supports the DNS clerk. It 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 use of this item list is similar to that of other system services that use a single item list for both input and output operations.

Item list entries must be specified in opaque format. You can convert any one of the name strings to opaque format with a conversion function. If applications need to store names, they must store them in opaque format. The opaque format guarantees the uniqueness of a name over time, whereas a string format does not.

Many of the functions return results as a set. In some cases, the specified output buffer might not be large enough to contain the complete set. In this case, the return status indicates this condition with the success status $DNS_MOREDATA. To obtain the remaining data from the set, the client should make repeated calls, each time specifying the last attribute received in the context variable item until the call returns SS$_NORMAL.

The context variable item can take one of two forms depending on the function:

- DNS$CONTEXTVARNAME—If the returned data is a set of names, then the item is a simple name.
- DNS$CONTEXTVARTIME—If the returned data is a set of values, then the item is a time-stamping.

If the context variable item is not specified or is null, then the results are returned from the beginning of the set.

All functions return the SS$_NORMAL status for success except DNS$_TEST_ATTRIBUTE, which returns DNS$_TRUE or DNS$_FALSE. The functions return linked information in the $DNS status block. The DNS$V_DNSB_OUTLINKED bit in the status block indicates whether any soft links are encountered in an information search.

CONDITION VALUES RETURNED

SS$_BADPARAM	Bad parameter value.
SS$_NORMAL	Normal completion of the request.
DNS$_ACCESSDENIED	Caller does not have required access to the entry in question. This error is returned only if the client has some access to the entry. Otherwise, the unknown entry status is returned.

DNS$_BADCLOCK	The clock at the name server has a value outside the permissible range.
DNS$_BADEPOCH	Copies of directories are not synchronized.
DNS$_BADITEMBUFFER	Invalid output item buffer detected. (This normally indicates that the buffer has been modified during the call.)
DNS$_CACHELOCKED	Global client cache locked.
DNS$_CLEARINGHOUSEDOWN	Clearinghouse is not available.
DNS$_CLERKBUG	Internal clerk error detected.
DNS$_CONFLICTINGARGUMENTS	Two or more optional arguments conflict; they cannot be specified in the same function call.
DNS$_DANGLINGLINK	Soft link points to nonexistent entry.
DNS$_DATACORRUPTION	An error occurred in accessing the data stored at a clearinghouse. The clearinghouse may be corrupted.
DNS$_ENTRYEXISTS	An entry with the same full name already exists in the namespace.
DNS$_FALSE	Unsuccessful test operation.
DNS$_INVALIDARGUMENT	A syntactically incorrect, out of range, or otherwise inappropriate argument was specified in the call.
DNS$_INVALID_ATTRIBUTENAME	The name given for function is not a valid DNS attribute name.
DNS$_INVALID_CLASSNAME	The name given for function is not a valid DNS class name.
DNS$_INVALID_ CLEARINGHOUSENAME	The name given for function is not a valid DNS clearinghouse name.
DNS$_INVALID_CONTEXTNAME	The name given for function is not a valid DNS name.
DNS$_INVALID_DIRECTORYNAME	The name given for function is not a valid DNS directory name.
DNS$_INVALID_ENTRYNAME	The name given for function is not a valid DNS entry name.
DNS$_INVALIDFUNCTION	Invalid function specified.
DNS$_INVALID_GROUPNAME	The name given for function is not a valid DNS group name.
DNS$_INVALIDITEM	Invalid item list entry specified.
DNS$_INVALID_LINKNAME	The name given for function is not a valid DNS link name.
DNS$_INVALID_MEMBERNAME	The name given for function is not a valid DNS name.
DNS$_INVALIDNAME	A badly formed name was supplied to the call.
DNS$_INVALID_NSNAME	Namespace name given in name string is not a valid DNS name.

DNS$_INVALID_OBJECTNAME	The name given for function is not a valid DNS object name.
DNS$_INVALID_TARGETNAME	The name given for function is not a valid DNS name.
DNS$_INVALIDUPDATE	An update was attempted to an attribute that cannot be directly modified by the client.
DNS$_INVALID_WILDCARDNAME	The name given for function is not a valid DNS name.
DNS$_LOGICAL_ERROR	Error translating logical name in given string.
DNS$_MISSINGITEM	Required item list entry is missing.
DNS$_MOREDATA	More output data to be returned.
DNS$_NAMESERVERBUG	A name server encountered an implementation bug. Please submit an SPR.
DNS$_NOCACHE	Client cache file not initialized.
DNS$_NOCOMMUNICATION	No communication was possible with any name server capable of processing the request. Check NCP event 353.5 for the DECnet error.
DNS$_NONSRESOURCES	The call could not be performed due to lack of memory or communication resources at the local node to process the request.
DNS$_NONSNAME	Unknown namespace name specified.
DNS$_NOTAGROUP	The full name given is not the name of a group.
DNS$_NOTIMPLEMENTED	This function is defined by the architecture as optional and is not available in this implementation.
DNS$_NOTLINKED	A link is not contained in the name.
DNS$_NOTNAMESERVER	The node contacted by the clerk does not have a DNS server running. This can happen when the application supplies the clerk with inaccurate replica information.
DNS$_NOTSUPPORTED	This version of the architecture does not support the requested function.
DNS$_POSSIBLECYCLE	Loop detected in link or group entry.
DNS$_RESOURCEERROR	Failure to obtain system resource.
DNS$_TIMEOUTNOTDONE	The operation did not complete in the time allotted. No modifications have been performed even if the operation requested them.
DNS$_TIMEOUTMAYBEDONE	The operation did not complete in the time allotted. Modifications may or may not have been made to the namespace.
DNS$_TRUE	Successful test operation.

DNS$_UNKNOWNCLEARINGHOUSE	The clearinghouse does not exist.
DNS$_UNKNOWNENTRY	Either the requested entry does not exist or the client does not have access to the entry.
DNS$_UNTRUSTEDCH	A DNS server is not included in the object's access control set.
DNS$_WRONGATTRIBUTETYPE	The caller specified an attribute type that did not match the actual type of the attribute.

$DNSW Distributed Name Service Clerk and Wait

The Distributed Name Service Clerk and Wait service registers a resource in a distributed database; same as $DNS. However, the $DNSW service completes synchronously; that is, it returns to the caller after the operation completes.

For asynchronous completion, use the $DNS service, which returns to the caller immediately after making a name service call. The return status to the client call indicates whether a request was successfully queued to the name service.

In all other respects, $DNSW is identical to $DNS. Refer to the $DNS description for complete information about the $DNSW service.

FORMAT **SYS$DNSW** *[efn] ,func ,itmlst ,[dnsb] ,[astadr] ,[astprm]*

A.2.9 DNS Run-Time Routines

All applications designed to take advantage of the Distributed Name Service (DNS) use the DNS clerk system services and the DNS run-time routines to register a resource in the DNS namespace and to modify and find information within the namespace. This section describes the run-time routines.

DNS$APPEND_SIMPLENAME_TO_RIGHT Append a Simple Name to the End of a Full Name

The Append a Simple Name to the End of a Full Name routine adds an opaque simple name to the end of an opaque full name to create a new full name.

FORMAT

DNS$APPEND_SIMPLENAME_TO_RIGHT
fullname ,simplename ,resulting-fullname ,resulting-length

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque full name gaining a new simple name. The **fullname** argument is the address of a descriptor pointing to the opaque full name that is to be extended.

simplename

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque simple name that is appended. The **simplename** argument is the address of a descriptor pointing to an opaque simple name that is to be appended to the full name, thus creating a new full name.

resulting-fullname

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The new full name. The **resulting-fullname** argument is the address of a descriptor that points to the buffer that receives the new full name.

This buffer can be the same as the buffer referred to by the **fullname** argument; however, the descriptors must be separate.

resulting-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the new full name. The **resulting-length** argument is the address of a word that receives the length of the new full name found in **resulting-fullname**.

DESCRIPTION

DNS$APPEND_SIMPLENAME_TO_RIGHT adds an opaque simple name to the end of an opaque full name to create a new full name.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.
0	Error appending name.

DNS$COMPARE_FULLNAME Compare Full Names

The Compare Full Names routine compares two opaque full names and returns the result.

FORMAT

DNS$COMPARE_FULLNAME *fullname1 ,fullname2*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname1

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

One opaque full name. The **fullname1** argument is the address of a descriptor pointing to an opaque full name.

fullname2

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

One opaque full name. The **fullname2** argument is the address of a descriptor pointing to an opaque full name.

DESCRIPTION

DNS$COMPARE_FULLNAME compares two opaque full names and returns the result. First, the procedure checks the namespace UIDs of the full names as numbers. If they are unequal, the routine returns the result. If they are equal, it compares each simple name in the full name until it finds an inequality or determines that both names are the same.

CONDITION VALUES RETURNED

–1	**fullname1** is less than **fullname2**.
0	**fullname1** equals **fullname2**.
1	**fullname1** is greater than **fullname2**.

DNS$COMPARE_SIMPLENAME Compare Two Simple Names

The Compare Two Simple Names routine compares two simple names, without considering case.

FORMAT

DNS$COMPARE_SIMPLENAME *simplename1 ,simplename2*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

simplename1

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

An opaque simple name. The **simplenamel** argument is the address of a descriptor pointing to the first simple name.

simplename2

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

An opaque simple name. The **simplename2** argument is the address of a descriptor pointing to the second simple name.

DESCRIPTION

DNS$COMPARE_SIMPLENAME compares two simple names, without considering case. The routine determines the relationship between two opaque simple names to see if they are equal.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
–1	**simplename1** is less than **simplename2**.
0	**simplename1** equals **simplename2**.
1	**simplename1** is greater than **simplename2**.

DNS$CONCATENATE_NAME Join Two Names

The Join Two Names routine joins two opaque full names to form a new full name.

FORMAT

DNS$CONCATENATE_NAME
fullname1 ,fullname2 ,resulting-fullname ,resulting-length

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname1

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque full name to be joined. The **fullname1** argument is the address of a descriptor pointing to the opaque full name.

fullname2

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque full name appended to **fullname1**. The **fullname2** argument is the address of a descriptor pointing to the full name to be appended.

resulting-fullname

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The buffer where the new full name will be written. The **resulting-fullname** argument is the address of a descriptor pointing to the buffer. This buffer can be the same as the buffer referred to by the **fullname1** argument; however, the descriptors must be separate.

resulting-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the new full name. The **resulting-length** argument is the address of a word that receives the length of the new full name found in **resulting-fullname**.

DESCRIPTION

DNS$CONCATENATE_NAME joins two opaque full names to form a new opaque full name, which is placed in the buffer named by the **resulting-fullname** argument. The new full name receives the namespace name of the first opaque full name. For example, appending the full name TEST:.POP.DIR1 (**fullname2**) to DEC:.ENG.NAC (**fullname1**) results in a full name of DEC:.ENG.NAC.POP.DIR1.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.
0	Error performing concatenation.

DNS$COUNT_SIMPLENAMES Count the Simple Names in a Full Name

The Count the Simple Names in a Full Name routine counts the number of simple names contained in an opaque full name.

FORMAT

DNS$COUNT_SIMPLENAMES *fullname ,count*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The full name to be counted. The **fullname** argument is the address of a descriptor pointing to the full name that is to be examined for the simple names it contains.

count

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The number of simple names found in the full name. The **count** argument is the address of a word that receives the number of simple names.

DESCRIPTION

DNS$COUNT_SIMPLENAMES counts the number of simple names—but not the namespace name—found in an opaque full name. The number of simple names counted is returned in the word referenced by the **count** argument. The routine is meant to be used with DNS$REMOVE_RIGHT_SIMPLENAME and DNS$REMOVE_LEFT_SIMPLENAME.

CONDITION VALUES RETURNED

SS$_NORMAL	*Normal successful completion.*
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.

DNS$CVT_DNSADDRESS_TO_BINARY Convert a DNS Address to a Phase IV Binary Address

The Convert a DNS Address to a Phase IV Binary Address routine takes a DNS address and returns the DECnet Phase IV node address.

FORMAT

DNS$CVT_DNSADDRESS_TO_BINARY *dnsaddress ,binary*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

dnsaddress

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The DNS address. The **dnsaddress** argument is the address of a descriptor pointing to the DNS address.

binary

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The DECnet Phase IV address found in the DNS address structure. The **binary** argument is the address of a word containing the 16-bit Phase IV address of the node.

DESCRIPTION

DNS$CVT_DNSADDRESS_TO_BINARY takes a DNS address and returns the DECnet Phase IV node address. The Phase IV address is returned in a word. If no Phase IV address is found in the DNS address, then the value 0 is returned as an error.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
0	No DECnet Phase IV address found.

DNS$CVT_DNSADDRESS_TO_NODENAME Convert a DNS Address to a Node Name

The Convert a DNS Address to a Node Name routine takes a DNS address and returns a DECnet Phase IV node name.

FORMAT

DNS$CVT_DNSADDRESS_TO_NODENAME
dnsaddress ,nodename ,resulting-length

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

dnsaddress

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The DNS address. The **dnsaddress** argument is the address of a descriptor pointing to the DNS address.

nodename

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The DECnet Phase IV node name. The **nodename** argument is the address of a descriptor pointing to the Phase IV node name. The memory buffer referenced by the DSC$A_POINTER portion of this descriptor must be large enough to contain the entire Phase IV node name string, which can be up to six bytes long.

resulting-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the node name (in bytes) after conversion. The **resulting-length** argument is a word containing the length of the node name (in bytes) after conversion.

DESCRIPTION

DNS$CVT_DNSADDRESS_TO_NODENAME takes a DNS address and returns a DECnet Phase IV node name. If no Phase IV address is found, then the value 0 is returned.

Because DNS$CVT_DNSADDRESS_TO_NODENAME calls both $ASSIGN and $QIOW, it can return condition values from either of these system services. The routine also returns errors detected through NETACP.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
0	No DECnet Phase IV address found.

DNS$CVT_NODENAME_TO_DNSADDRESS Convert a Node Name to an Address

The Convert a Node Name to a DNS Address routine takes a DECnet Phase IV node name and returns a DNS address.

FORMAT

DNS$CVT_NODENAME_TO_DNSADDRESS
nodename ,dnsaddress ,resulting-length

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

nodename

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The DECnet Phase IV node name. The **nodename** argument is the address of a descriptor pointing to the node name. This routine creates a DNS address containing the node address of the given Phase IV node name.

dnsaddress

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The address of a buffer containing the DNS address. The **dnsaddress** argument is the address of a descriptor pointing to the buffer address.

resulting-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the DNS address after conversion. The **resulting-length** argument is a word containing the length of the address.

DESCRIPTION

DNS$CVT_NODENAME_TO_DNSADDRESS takes a DECnet Phase IV node name and returns a DNS address. The routine creates the DNS address for a given Phase IV node name.

DNS$CVT_NODENAME_TO_DNSADDRESS calls $ASSIGN and $QIOW so it can return condition values from either of these system services. The routine also returns errors detected through NETACP.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.

DNS$CVT_TO_USERNAME_STRING Convert an Opaque User Name to a String

The Convert an Opaque User Name to a String routine converts an opaque DECnet Phase IV user name into a username string.

FORMAT

DNS$CVT_TO_USERNAME_STRING
fullname ,username ,resulting-length

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque full name for the DECnet Phase IV user name. The **fullname** argument is the address of a descriptor pointing to the name.

username

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The name converted to DECnet Phase IV format (node::user). The **username** argument is the address of a descriptor pointing to a buffer containing the converted name.

resulting-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the converted user name. The **resulting-length** argument is the address of a word containing the length.

DESCRIPTION

DNS$CVT_TO_USERNAME_STRING converts a DNS representation of a Phase IV user name into a Phase IV username string.

If any full name other than a DNS representation of a Phase IV user name is given, the routine returns a DNS$_INVALIDNAME error.

CONDITION VALUES RETURNED

SS$_NORMAL	Procedure successfully completed.
DNS$_ACCESSVIOLATION	Memory or other resource access violation.
DNS$_CACHELOCKED	Global client cache locked by another process.
DNS$_INVALIDARGUMENT	One of the arguments was incorrect, out of range, or otherwise inappropriate.
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.
DNS$_NOCACHE	Client cache file not initialized.
DNS$_RESOURCEERROR	Insufficient resources on local system to process request.

DNS$PARSE_USERNAME_STRING Convert a User Name from String to Opaque

The Convert a User Name from String to Opaque routine converts a DECnet Phase IV user name to an opaque full name.

FORMAT

DNS$PARSE_USERNAME_STRING
user-string ,phase4-name ,resulting-length [,next-character-pointer]

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

user-string

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The name string to convert. The **user-string** argument is the address of a descriptor pointing to the DECnet Phase IV username string, which is in the format *node::user*.

phase4-name

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The opaque full name resulting from conversion. The **phase4-name** argument is the address of a descriptor pointing to the buffer that is to contain an opaque full name representing a user name on a Phase IV node.

resulting-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the opaque full name. The **resulting-length** argument is the address of a word holding the length of the name returned in **phase4-name**.

next-character-pointer

VMS usage:	**address**
type:	**address**
access:	**write only**
mechanism:	**by reference**

The character following the DNS name extracted from **user-string**. The **next-character-pointer** argument is the address of the character following the DNS name. When you use this argument, DNS$PARSE_USERNAME_STRING returns DNS$_INVALIDNAME when it encounters an invalid name. In such a case, **next-character-pointer** points to the first character in the name that is invalid.

DESCRIPTION

DNS$PARSE_USERNAME_STRING converts a DECnet Phase IV user name to an opaque full name that represents the user name.

The **next-character-pointer** argument affects how the routine parses the string:

- When **next-character-pointer** is zero or absent, the full name string given in **user-string** must contain valid DNS characters with DNS naming syntax. If any part of the string violates this rule, the routine returns DNS$_INVALIDNAME and the output should not be used.
- When the **next-character-pointer** argument has a nonzero value, the parsing begins at the first character referenced by **user-string** and parsing continues until one of the following occurs:
  - An invalid DNS character is found.
  - An exception to DNS syntax rules occurs.
  - All characters have been parsed.

  Then the address given by **next-character-pointer** is set to the address of the character or group of characters that is invalid. It returns DNS$_INVALIDNAME if the colons (::) separating the node name from the user name of the Phase IV name are missing.

If any part of the node portion of the DECnet Phase IV username string is not a proper DNS name, the routine returns DNS$_INVALIDNAME regardless of the value and whether or not the **next-character-pointer** argument is supplied.

Error conditions can result from the parse routine. You can test for error conditions in any of the following ways:

- When all parts of the name are invalid, test whether **next-character-pointer** contains the same address as **user-string**. Alternatively, test whether the resulting length is zero.
- When **user-string** contains a valid DNS name, test whether **next-character-pointer** contains the address immediately following the given buffer. Alternatively, test whether the address in **next-character-pointer** minus the address of **user-string** is equal to or larger than the size of the given buffer.

- When parsing a user name that has been extracted from a command line, test whether the character given at the address of **next-character-pointer** is a valid separator for the command line, for example, a space. If you find an invalid character, then part of the DNS name is invalid.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
DNS$_ACCESSVIOLATION	Memory or other resource access violation.
DNS$_CACHELOCKED	Global client cache locked by another process.
DNS$_INVALIDARGUMENT	One of the arguments was incorrect, out of range, or otherwise inappropriate.
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.
DNS$_NOCACHE	Client cache file not initialized.
DNS$_RESOURCEERROR	Insufficient resources on local system to process request.
0	Error creating opaque name.

DNS$REMOVE_FIRST_SET_VALUE Remove a Value from a Set

The Remove a Value from a Set routine extracts the first value from a set and returns the value with its creation time-stamping UID.

FORMAT

DNS$REMOVE_FIRST_SET_VALUE
set [,value] [,value-length] [,uid] [,uid-length] [,newset] [,newset-length]

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

set

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The set from which the value is extracted. The **set** argument is the address of a descriptor pointing to the set.

value

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The value extracted from the set. The **value** argument is the address of a descriptor pointing to a buffer containing the value.

value-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the value. The **value-length** argument is the address of a word holding the length of the value placed in **value**.

uid

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The buffer holding the creation time-stamping UID of the extracted value. The **uid** argument is the address of a descriptor pointing to the buffer.

uid-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the UID placed in **uid**. The **uid-length** argument is the address of a word holding the length.

newset

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The opaque **set** without the first value. The **newset** argument is the address of a descriptor pointing to a buffer containing that set. The buffer can be the same as the one given in the **set** argument.

newset-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the new set copied to the **newset** buffer. The **newset-length** argument is the address of a word that receives the length.

DESCRIPTION

DNS$REMOVE_FIRST_SET_VALUE extracts a value from a set and returns the value with its creation time-stamping UID. Use the routine to extract values from the sets returned by $DNS and $DNSW.

A set can contain any number of values that are relevant to a given call. The routine extracts values one at a time from the opaque set for use by a program. In order to extract all values from the set, you must use an execution loop.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
DNS$_INVALIDARGUMENT	The set argument was incorrect, out of range, or otherwise inappropriate.
0	Set buffer is empty.
–1	Length of **value**, **uid**, or **newset** buffers too small.

DNS$REMOVE_LEFT_SIMPLENAME Strip the Simple Name on the Left from the Full Name

The Remove the Simple Name on the Left from the Full Name routine removes the leftmost simple name from an opaque full name. It returns both the simple name stripped and the new full name that results from the operation.

FORMAT

DNS$REMOVE_LEFT_SIMPLENAME
fullname [,resulting-fullname] [,resulting-fullname-length] [,resulting-simplename] [,resulting-simplename-length]

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque full name to strip. The **fullname** argument is the address of a descriptor pointing to the opaque full name to strip. If the full name does not contain any simple names, the routine returns a value of 0 in **cond_value**.

resulting-fullname

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The opaque full name resulting from the operation. The **resulting-fullname** argument is the address of a descriptor pointing to the buffer containing the resulting opaque full name. This buffer can be the same as the buffer referred to by the **fullname** argument; however, the descriptors must be separate.

resulting-fullname-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the full name that is returned. The **resulting-fullname-length** argument is the address of a word receiving the length of the full name returned in **resulting-fullname**.

resulting-simplename

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The simple name stripped from **fullname**. The **resulting-simplename** argument is the address of a descriptor pointing to the buffer containing the opaque simple name that was stripped.

resulting-simplename-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the simple name. The **resulting-simplename-length** argument is the address of a word that receives the length of the simple name returned in **resulting-simplename**.

DESCRIPTION

DNS$REMOVE_LEFT_SIMPLENAME removes the leftmost simple name from an opaque full name. When **resulting-fullname** is nonzero, the full name resulting from the removal of the leftmost simple name is returned in that buffer. When **resulting-simplename** is nonzero, the simple name that was stripped from **fullname** is returned in that buffer. The namespace name is not stripped from the full name; only simple names are affected.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.
–1	Error stripping name.
0	No simple name.

DNS$REMOVE_RIGHT_SIMPLENAME Strip the Simple Name on the Right from the Full Name

The Remove the Simple Name on the Right from the Full Name routine removes the rightmost simple name from an opaque full name. It returns both the simple name stripped and the new full name that results from the operation.

FORMAT

DNS$REMOVE_RIGHT_SIMPLENAME
fullname [,resulting-fullname] [,resulting-fullname-length] [,resulting-simplename] [,resulting-simplename-length]

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

ARGUMENTS

fullname

VMS usage:	**char_string**
type:	**character string**
access:	**read only**
mechanism:	**by descriptor**

The opaque full name to strip. The **fullname** argument is the address of a descriptor pointing to the opaque full name to strip. When the opaque full name does not contain any simple names, the routine returns a value of 0 in **cond_value**.

resulting-fullname

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

The opaque full name resulting from the operation. The **resulting-fullname** argument is the address of a descriptor pointing to a buffer containing the resulting opaque full name. This buffer can be the same as the buffer referred to by the **fullname** argument; however, the descriptors must be separate.

resulting-fullname-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the full name returned in **resulting-fullname**. The **resulting-fullname-length** argument is the address of a word that receives the length of the full name returned in **resulting-fullname**.

resulting-simplename

VMS usage:	**char_string**
type:	**character string**
access:	**write only**
mechanism:	**by descriptor**

A buffer containing the opaque simple name stripped from **fullname**. The **resulting-simplename** argument is the address of a descriptor pointing to the buffer.

resulting-simplename-length

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

The length of the simple name. The **resulting-simplename-length** argument is the address of a word receiving the length of the simple name returned in **resulting-simplename**.

DESCRIPTION

DNS$REMOVE_RIGHT_SIMPLENAME removes the rightmost simple name from an opaque full name. When **resulting-fullname** is nonzero, the full name resulting from the removal of the rightmost simple name is returned in that buffer. When **resulting-simplename** is nonzero, the simple name that was stripped from **fullname** is returned in that buffer. The namespace name is not stripped from the full name; only simple names are affected.

CONDITION VALUES RETURNED

SS$_NORMAL	Normal successful completion.
DNS$_INVALIDNAME	The name to be converted is not a valid DNS name.
–1	Error stripping name.

A.2.10 Starting the DNS Clerk

The VAX Distributed Name Service (DNS) is a product consisting of two modules: a clerk and a server. The DNS clerk is an integral part of the VMS operating system. The server is a layered product. As long as a DNS server is installed in your network, you can start the DNS clerk on your local VMS system. Then, applications can take advantage of the DNS name service.

You start the DNS clerk once DECnet is running. The DNS startup procedure defines the default DNS server, installs necessary libraries, and creates an advertiser process. Startup involves two steps:

1. Obtain the name of the default DNS server from your network administrator.
2. Execute the command procedure DNS$CHANGE_DEF_FILE. It runs the command procedure DNS$CLERK_STARTUP, which installs the shareable libraries and creates the advertiser process DNS$ADVER.

   To run the command procedure, enter the following command:

   ```
   $  @SYS$STARTUP:DNS$CHANGE_DEF_FILE.COM
   ```

   When executed, SYS$STARTUP:DNS$CHANGE_DEF_FILE.COM prompts for the name of the node where the DNS server is located.

   ```
   Name of DNS server node?
   ```

   Enter a node name, identifying the node that has the DNS server installed.

Once you have run DNS$CHANGE_DEF_FILE.COM, you do not need to run it again unless you want to change the default DNS server or the default namespace. DNS$CHANGE_DEF_FILE.COM copies a configuration file to SYS$SYSTEM that is called DNS$DEFAULT_FILE.DAT. It lists the name of the namespace currently being used as a default.

You must add the following line to SYS$MANAGER:SYSTARTUP.COM after the line that starts DECNET: @SYS$STARTUP:DNS$CLERK_STARTUP.COM. When the system boots, this line installs the DNS clerk images and starts the advertiser.

A.2.11 DECnet Event Messages

The following are DECnet event messages sent by the Distributed Name Service clerk. For a complete list of DECnet event messages, see the *VMS Network Control Program Manual.*

353.5 DNS Clerk Unable to Communicate with Server

The DNS clerk was unable to communicate with a DNS server. This message displays the name of the clearinghouse where the communication failed, the node on which the DNS server servicing the clearinghouse exists, and the reason why the communication failed, which might be any of the following:

- Unknown clearinghouse

  The requested clearinghouse is not serviced by the DNS server that was contacted. This can happen when the cache maintained by the local DNS clerk contains outdated information for a directory.

- Clearinghouse down

  A DNS server is unable to service a request because the clearinghouse is not operational (stopped state).

- Wrong state

  A DNS server is unable to service a request because the clearinghouse is currently starting up or shutting down.

- Data corruption

  A DNS server is unable to service the request because the clearinghouse file has been corrupted.

- No communication

  A network error occurred on the local system or on the system containing the DNS server. The local VMS error is displayed as a part of this message.

353.20 Local DNS Advertiser Error

This event communicates errors that are local to the DNS Advertiser (DNS$ADVER). All these errors have the prefix ADV and are generated when the DNS Advertiser has encounters an error that prevents proper operation of the process. Exact errors are listed in the *VMS System Messages and Recovery Procedures Reference Manual.*

B VMS Version 5.2 Features

This appendix describes features that were new to Version 5.2 of the VMS operating system but are not yet documented in other printed manuals.

B.1 VMS Version 5.2 System Management Features

The following sections describe enhancements to these components of the VMS operating system:

- System Generation Utility (SYSGEN)
- NETCONFIG.COM
- Backup Utility (BACKUP)

B.1.1 System Generation Utility (SYSGEN)

The VMS Version 5.2 System Generation Utility (SYSGEN) contains the following new command and parameter:

- DEINSTALL command
- ERLBUFFERPAGES parameter

B.1.1.1 DEINSTALL Command Description

DEINSTALL removes or "deinstalls" system page files and system swap files. Any file that is installed with the INSTALL command can be removed with the DEINSTALL command.

Use of the DEINSTALL command requires the CMKRNL privilege.

Format

DEINSTALL *filespec*

Parameter

filespec
Specifies the name of the page or swap file. The default file type is SYS.

Qualifiers

/ALL
Deinstalls all page and swap files currently installed on the system. This command is most useful during an orderly system shutdown procedure where all disk volumes are being dismounted.

/INDEX=n
Deinstalls a page or swap file specified by the page file index. The page file index is presented in the SHOW MEMORY/FILES/FULL display as "Paging File Number."

/PAGEFILE
Specifies that the file to be deinstalled is a page file.

/SWAPFILE
Specifies that the file to be deinstalled is a swap file.

Example

```
SYSGEN>  DEINSTALL SYS$SYSTEM:PAGEFILE.SYS/PAGEFILE
```

The command in this example deinstalls the system page file.

B.1.1.2 ERLBUFFERPAGES Parameter

The ERLBUFFERPAGES parameter specifies the number of pages of memory to allocate for each buffer requested by the ERRORLOGBUFFERS parameter. The ERLBUFFERPAGES parameter has a default value of 2 pages and a maximum value of 32 pages. The default value of 2 pages consists of one page for each buffer greater than the previous buffer size.

B.1.2 NETCONFIG.COM Security Enhancements

In VMS Version 5.2, the DECnet network configuration command procedure NETCONFIG.COM has been enhanced to provide several options for restricting default access. A new command procedure for existing networked systems, NETCONFIG_UPDATE.COM, described in Section B.1.3, has been created for the same purpose.

You can plan the appropriate level of default access for your system and implement that plan by responding to a few questions posed by NETCONFIG.COM. NETCONFIG.COM then automatically records your choices in the UAF (user authorization file) and in the network configuration database.

Previously, NETCONFIG.COM created one default account named DECNET. That account provided default access to all network objects and applications that were not restricted by other forms of access control (for example, proxy accounts and access control lists). If you chose to limit default access, it was necessary to manually enter all the appropriate commands in the UAF, using the Authorize Utility, and in the network configuration database, using NCP commands.

B.1.2.1 Default Access Options

NETCONFIG.COM provides two different ways to limit default access. The most restrictive form is to not create the default DECnet account but to grant default access for certain system objects by creating a default account for each one that you want to use. Using NETCONFIG.COM, you can create an account for one or more of the following network objects:

- MAIL
- File access listener (FAL)
- PHONE
- Network management listener (NML)
- Loopback mirror (MIRROR)

- VMS Performance Monitor (VPM)

The second, less restrictive form of default access is to create a default DECnet account but to disable default access to user-written programs and procedures (also known as TASK objects). Default access for system objects is still enabled.

You can still create an unrestricted default DECnet account that includes default access to TASK objects. This type of access is suitable for systems with low security requirements. To do so, you must override the defaults provided by NETCONFIG.COM.

Note: **If you do not create the default DECnet account, you must create a default account for MAIL and VPM, if you want to use them. The same is true for the MIRROR object if you want to use the User Environment Test Package (UETP) to test the network connection.**

FAL, if enabled by the default DECnet account or a separate default account, makes a system vulnerable to unauthorized access. Digital advises against creating a default account for FAL. Note, however, that when you do *not* use FAL with a default account, remote file requests must include explicit file access control information, or the local system manager must set up proxy access for remote users. Consider an example with a local node ETHQKE, and a remote node MISHA with no default account. Entering the command $DIR MISHA:: from node ETHQKE produces the following messages:

```
%DIRECT-E-OPENIN, error opening MISHA::*.*;* as input
-RMS-E-FND, ACP file or directory lookup failed
-SYSTEM-F-INVLOGIN, login information invalid at remote node
```

However, you can access node MISHA by entering the command $DIR MISHA"Username Password":: from node ETHQKE.

The system manager could also, by using AUTHORIZE, enable proxy access for node ETHQKE by adding REMOTE_USER_FOO, as shown in the following example:

```
$  SET DEF SYS$SYSTEM
$  RUN AUTHORIZE
UAF>  ADD/PROXY/DEFAULT ETHQKE::REMOTE_USER_FOO LOCAL_USER
UAF>  EXIT
```

Entering the command $ DIR MISHA:: from node ETHQKE would then give user ETHQKE::REMOTE_USER_FOO access to remote node MISHA by proxy; MISHA then associates this account with the account LOCAL_USER on node MISHA.

The MIRROR object is used for loopback testing. To test your network connection with VAX UETP you must create a default account for the MIRROR object, if you did not create the default DECnet account.

The VPM object is used by the Monitor Utility in VAXcluster configurations to obtain performance information about VAXcluster members. If your system is a member of a VAXcluster and the cluster manager wants to use the Monitor Utility to collect such information, you must create a default account for the VPM object, if you did not create the default DECnet account.

B.1.2.2 Security Benefits

The DECnet account provides default access for all incoming links (unless this access is overridden by other forms of access control). However, default accounts for any of the system objects named in the NETCONFIG.COM procedure limit access to these objects. Default accounts for selected objects, when used with other system security facilities, enable a system or network manager to monitor these accounts and to detect unauthorized access.

For each default account that you create, NETCONFIG.COM generates a password and registers it in your network configuration database. Such system-generated passwords are more secure than the passwords that users typically create.

B.1.2.3 Questions Posed by NETCONFIG.COM

NETCONFIG.COM poses the following questions (the responses in brackets are the default values):

```
Do you want a default DECnet account?                     [NO]:
```

(The following question is asked only if you said YES to a default DECnet account.)

```
Do you want default access to the TASK object disabled? [YES]:
```

(The following questions are asked regardless of whether you said YES or NO to a default DECnet account.)

```
Do you want a default account for the MAIL object?      [YES]:

Do you want a default account for the FAL object?        [NO]:

Do you want a default account for the PHONE object?     [YES]:

Do you want a default account for the NML object?       [YES]:
```

(The following questions are asked only if you said NO to a default DECnet account.)

```
Do you want a default account for the MIRROR object?    [YES]:

Do you want a default account for the VPM object?       [YES]:
```

B.1.3 New NETCONFIG_UPDATE.COM for Existing Networks

NETCONFIG_UPDATE.COM is a new command procedure for existing networks that provides the same security enhancements for default access that are provided by NETCONFIG.COM (see Section B.1.2). It also provides a secondary procedure for modifying members of a VAXcluster. Both procedures are described in the following sections.

B.1.3.1 Benefits of NETCONFIG_UPDATE.COM

NETCONFIG_UPDATE.COM, unlike NETCONFIG.COM, configures default access only. It performs no other network configuration. Therefore, when you use NETCONFIG_UPDATE.COM to specify changes to default access, everything else in the configuration database remains unchanged.

NETCONFIG_UPDATE.COM, like NETCONFIG.COM, generates passwords for each account that you create for default access and for any existing default accounts that you decide to keep in your configuration database. For example, if you currently have a default account for MAIL and decide to keep it, NETCONFIG.COM_UPDATE generates a new password for it and replaces the existing password with the new one.

B.1.3.2 Using NETCONFIG_UPDATE.COM in a VAXcluster

NETCONFIG_UPDATE.COM provides a secondary procedure that updates the default access of VAXcluster members. After you run NETCONFIG_UPDATE.COM on one member of a VAXcluster, the procedure detects that it is a VAXcluster member and instructs you to run SYS$COMMON:[SYSMGR]UPDATE_CLUSTER_MEMBERS.COM on the other VAXcluster members. This secondary procedure will modify the default access of each VAXcluster member exactly as you modified that of the first member.

With the SYSMAN Utility (see the *VMS SYSMAN Utility Manual*), you can use the SET ENVIRONMENT/CLUSTER command to execute this secondary procedure only once. The default access of all the remaining VAXcluster members will be updated automatically.

B.1.4 Backup Utility (BACKUP)

This section describes the following new Backup Utility (BACKUP) features:

- Performance enhancements that cause BACKUP save and copy operations to complete more quickly on systems that are configured correctly
- Faster cyclic redundancy checking (CRC) emulation for processors that emulate CRC in software, resulting in a significant performance enhancement for BACKUP on these processors
- Support for the control character Ctrl/T, which returns information about the online or standalone BACKUP operation in progress

B.1.4.1 Performance Enhancements

Version 5.2 of the Backup Utility includes a new method of scanning files on the input disk. This new file-scanning method results in faster save and copy operations on systems that are configured correctly. (It does not improve BACKUP's performance during restore, compare, verify, or list operations, however.) Prior to Version 5.2, disk head movement on the input disk constrained the speed at which BACKUP could save or copy files.

To take full advantage of the new BACKUP file-scanning method, you must change the values of certain user authorization file (UAF) and System Generation Utility (SYSGEN) parameters. Sections B.1.4.2 and B.1.4.3 specify which parameters you need to change.

B.1.4.2 Setting Up the BACKUP Account

BACKUP's new file-scanning method depends on the values of some user authorization file (UAF) parameters of the account from which you perform BACKUP operations. For example, if you perform BACKUP operations from the SYSTEM account, the UAF parameters for the SYSTEM account affect the way BACKUP performs. These UAF parameters define process quotas, which are the amounts of system resources available to a process created by the account. Digital recommends that you change the values of these UAF parameters for the account you use to perform BACKUP operations. See the *VMS Authorize Utility Manual* for more information about modifying the values of UAF parameters.

Table B–1 describes the UAF parameters that should be modified and supplies values that provide the maximum amount of resources to BACKUP. These values may not provide the best performance in all cases, however. They are intended to be general guidelines.

Note: **BACKUP bases its memory consumption on the WSQUOTA value, not WSEXTENT.**

Table B–1 UAF Process Quotas for the BACKUP Account

UAF Parameter	Meaning	Recommended Value
WSQUOTA	The number of pages of memory the working set of the process can consume.	Equal to SYSGEN parameter WSMAX
WSEXTENT	The absolute limit of physical memory allowed to the process.	Equal to WSQUOTA
PGFLQUO	The number of pages of memory your process is allowed in the page file.	Greater than or equal to WSEXTENT
FILLM	The number of files that can be open simultaneously. BACKUP scans this number of files at one time.	Equal to the SYSGEN parameter CHANNELCNT
DIOLM	The number of direct I/O operations (usually disk operations) that can be outstanding simultaneously.	Maximum of either (3 x FILLM) or 4096
ASTLM	The number of asynchronous system traps that can be queued to the process simultaneously.	Maximum of either (3 x FILLM) or 4096
BIOLM	The maximum number of buffered I/O operations that can be outstanding simultaneously.	Less than or equal to FILLM
BYTLM	The total number of bytes of memory that can be outstanding for buffered I/O operations.	Greater than or equal to the following value: (256 x FILLM) + (6 x DIOLM)
ENQLM	The maximum number of locks that can be queued simultaneously.	Greater than FILLM

Table B–2 lists a set of UAF parameter values that may be useful for your configuration. You can choose to set the values for WSQUOTA and FILLM lower than these values under the following circumstances:

- If your disks are highly fragmented, lower values prevent BACKUP from becoming highly CPU-intensive.
- If you use BACKUP during periods of heavy system use, lower values prevent BACKUP from consuming too many system resources.

Note: If you decrease the values of UAF parameters other than WSQUOTA and FILLM, use the ratios in Table B–1 to determine appropriate values.

Alternatively, you can choose to set the values higher than these suggested values if files are stored contiguously on your disks and if you perform BACKUP operations during periods of light system use.

Table B–2 Suggested Values for UAF Process Quotas

UAF Parameter	Value
WSQUOTA	16,384
WSEXTENT	Greater than or equal to WSQUOTA
PGFLQUO	32,768
FILLM	128
DIOLM	4096
ASTLM	4096
BIOLM	128
BYTLM	65,536
ENQLM	256

After changing UAF parameters, log out of the BACKUP account and log back in, allowing the new values of the UAF parameters to be used.

B.1.4.3 Setting System Generation Utility (SYSGEN) Parameters

For the new BACKUP file-scanning method to work efficiently, the System Generation Utility (SYSGEN) parameters CHANNELCNT and WSMAX must be set to appropriate values. If the account you use to perform BACKUP operations has a FILLM value greater than the value of the SYSGEN parameter CHANNELCNT, CHANNELCNT constrains the number of files that can be opened at any one time. If the WSQUOTA value of the account is greater than the value of the SYSGEN parameter WSMAX, WSMAX constrains the number of pages of memory that the working set of the process can consume. See the *VMS System Generation Utility Manual* for more information about changing the values of SYSGEN parameters.

After changing SYSGEN parameters, shut down and reboot the system, allowing the new values of the parameters to be used.

B.1.4.4 Understanding Why the Output Device Seems Idle

Because BACKUP can scan many files at a time, it is possible that no data will be sent to the output device for up to several minutes after the save or copy operation begins. This does not indicate that BACKUP is performing slowly or that your output device is not working correctly. Depending on the values of the UAF parameters and the SYSGEN parameters, BACKUP's new file-scanning method requires a certain amount of time to become established. When the file-scanning is completed, BACKUP sends the data to the output device more efficiently than it did before VMS Version 5.2.

B.1.4.5 /BUFFER_COUNT Command Qualifier Is Now Obsolete

The new file-scanning method used by BACKUP makes the command qualifier /BUFFER_COUNT obsolete. Previously, this command qualifier specified the number of buffers used in a save, compare, or restore operation to or from a tape. BACKUP now determines how many buffers to use, depending on the amount of memory available to the account performing the BACKUP operation and the number of files that account can open simultaneously.

You can still specify the /BUFFER_COUNT qualifier, however, although it has no effect. This ensures that command procedures written before VMS Version 5.2 will still operate correctly. Digital recommends that you remove the /BUFFER_COUNT qualifier from command procedures.

B.1.4.6 Cyclic Redundancy Checking Emulation Improvements

The method for performing cyclic redundancy checking (CRC) emulation is now approximately 40% faster than the method used before VMS Version 5.2. This is not a BACKUP-specific improvement, but it does improve BACKUP performance on processors that emulate CRC in software. BACKUP operations that use cyclic redundancy checking (CRC is applied by default) now require significantly less time to complete on the following processors, all of which emulate CRC in software:

- MicroVAX II/VAXstation II
- MicroVAX 2000/VAXstation 2000
- MicroVAX 3200/VAXstation 3200
- MicroVAX 3500/VAXstation 3500
- MicroVAX 3600
- VAX 6200

B.1.4.7 Pressing Ctrl/T to Obtain Information About BACKUP Operations

Version 5.2 of the VMS operating system supplies an additional two lines of information when you press Ctrl/T during an online or standalone BACKUP operation. Ctrl/T interrupts execution of the BACKUP command, and displays three lines of information. The first line displays information about the current process (node name, process name, system time, currently running image, elapsed CPU time, page faults, direct and buffered I/O operations, and pages in physical memory). The second line displays information about BACKUP input. The third line displays information about BACKUP output. For example, if you press Ctrl/T during a save operation, the second line displays the name of the last file scanned by BACKUP and the third line displays the save-set volume number, save-set block number, and the number of bytes in a block.

In order to use Ctrl/T, the command SET CONTROL=T must appear either in the system login command procedure or in your personal login command procedure. You can also enable Ctrl/T interactively by entering the DCL command SET CONTROL=T.

The following example shows what happens when you press Ctrl/T during a BACKUP save operation:

```
$  BACKUP/LOG DUA0:[MISHA]*.COM;* MUA0:COMPROCS.BCK/REWIND/LABEL=COMP
BACKUP-S-COPIED, copied DUA0:[MISHA]A.COM;32
BACKUP-S-COPIED, copied DUA0:[MISHA]B.COM;30
BACKUP-S-COPIED, copied DUA0:[MISHA]C.COM;16
Ctrl/T
SQUASH::MISHA 14:02:12 BACKUP   CPU=00:00.18.44 PF=2101 IO=827 MEM=534
Last file scanned: DUA0:[NATASHA]D.DAT
Saveset volume: 1, saveset block: 35, (32256 byte blocks)
BACKUP-S-COPIED, copied DUA0:[MISHA]D.COM;2
BACKUP-S-COPIED, copied DUA0:[MISHA]E.COM;22
    .
    .
    .
$
```

B.2 VMS Version 5.2 System Services Features

The VMS Version 5.2 operating system includes the following new system services:

New Service	Function
$DEVICE_SCAN	Scan across the system for devices
$PROCESS_SCAN	Scan across the system or cluster for processes

The Device Scan system service, described in Section B.2.30, lets you find the names of all devices that match a specified set of search criteria. $DEVICE_SCAN can be used to produce a list of all disks, printers, or terminals on the local node. After $DEVICE_SCAN has located device information, you can use $GETDVI to further select the information, for instance, to find the names of all mounted disks or all terminals running at the same baud rate.

The Process Scan system service, described in Section B.2.30, lets you scan for processes across the cluster. With the VMS Version 5.2 operating system, any VMS process can now be seen or modified from any node in a VAXcluster environment. Any system service that examines or modifies a process is now capable of examining or controlling a process located on a different node in the VAXcluster environment.

For Version 5.2 of the VMS operating system, processes are visible clusterwide. As a result, significant changes have been made to the following system services:

- $CANWAK
- $DELPRC
- $FORCEX
- $GETJPI
- $RESUME
- $SCHDWK
- $SETPRI
- $SUSPEND
- $WAKE

Changes have also been made to the $GETUAI, $SETUAI, $MOUNT, $DISMOUNT, and $MOD_IDENT system services. The following sections describe the new and changed system services in more detail.

B.2.1 Modifications to $SETUAI and $GETUAI

VMS Version 5.2 includes the following changes to the $SETUAI and $GETUAI system services:

- New $SETUAI item codes

  UAI$_PASSWORD
  UAI$_PASSWORD2
  UAI$_USER_DATA

- New $SETUAI authorization flags

  UAI$V_DISIMAGE
  UAI$V_RESTRICTED

- New $GETUAI item code

  UAI$_USER_DATA

- New $GETUAI authorization flags

  UAI$V_DISIMAGE
  UAI$V_RESTRICTED

B.2.2 New Item Codes for $SETUAI and $GETUAI

The following new item codes have been added for the $SETUAI system service:

New Item Code	Description
UAI$_PASSWORD	When you specify UAI$_PASSWORD, $SETUAI sets the specified plaintext string as the primary password of the user and updates the password change date.
UAI$_PASSWORD2	When you specify UAI$_PASSWORD2, $SETUAI sets the specified plaintext string as the secondary password of the user and updates the password change date.
UAI$_USER_DATA	When you specify UAI$_USER_DATA, $SETUAI sets up to 255 bytes of information in the user data area of the system user authorization file (SYSUAF). This is the supported method for modifying the user data area of the SYSUAF. Digital no longer supports direct user modification of the SYSUAF. To clear all information in the user data area of the SYSUAF, specify $SETUAI with a buffer length of zero.

The SYSPRV privilege is required to set any passwords (including the password of the calling process) or to modify the user data with $SETUAI.

The UAI$_PASSWORD and UAI$_PASSWORD2 item codes provide the building blocks for designing a site-specific SET PASSWORD utility. If you create such a utility, you should set the LOCKPWD bit in the user authorization file (UAF) to prevent users from using the SET PASSWORD command and to prevent the LOGINOUT process from forcing password changes. If you create a site-specific SET PASSWORD utility, install the utility with SYSPRV privilege.

When specifying a password with UAI$_PASSWORD or UAI$_PASSWORD2, adhere to the following guidelines:

- The password must meet the minimum password length defined on the system.
- The password cannot exceed 32 characters in length.
- The password must be different from the previous password.

To clear the primary or secondary password, specify UAI$_PASSWORD or UAI$_PASSWORD2 with a buffer length of zero.

VMS Version 5.2 includes a new item code—UAI$_USER_DATA—for the $GETUAI system service.

New Item Code	Description
UAI$_USER_DATA	When you specify UAI$_USER_DATA, $GETUAI reads up to 255 bytes of information from the user data area of the system user authorization file (SYSUAF). You can read information written to the user data area from previous versions of the VMS operating system as long as the information adheres to the guidelines described in the *Guide to VMS System Security*.

B.2.3 New Authorization Flags for $SETUAI and $GETUAI

Two new authorization flags, UAI$V_DISIMAGE and UAI$V_RESTRICTED, are used in the creation of captive and restricted user accounts. (See the *Guide to VMS System Security* for a complete description of these flags.) Use the $SETUAI system service to set the flags for the specified user. Use the $GETUAI system service to determine whether the specified flag is set. The new flags are represented as bits in the UAI$_FLAGS item code with the following symbolic names:

New Authorization Flags	Description
UAI$V_DISIMAGE	When you specify UAI$V_DISIMAGE, the user cannot issue the RUN or MCR commands or use the foreign command mechanism in DCL.
UAI$V_RESTRICTED	Set the RESTRICTED flag (UAI$V_RESTRICTED) to return the account to the level of security previously specified by the CAPTIVE authorization flag in earlier versions of the VMS operating system. (Under VMS Version 5.2, the security of accounts with the CAPTIVE flag set (UAI$_V_CAPTIVE) has been increased by disallowing any access to the DCL command level and disallowing use of the INQUIRE verb in captive command procedures.)

B.2.4 Modifications to $MOUNT

The following flags have been added to the $MOUNT system service:

New Flag	Description
MNT$M_NOLABEL	The volume is to be mounted as a foreign volume; a foreign volume is not Files–11 structured. If you specify MNT$M_NOLABEL, the following item codes can each appear in the item list only once: MNT$_DEVNAM, MNT$_VOLNAM, and MNT$_LOGNAM. To specify MNT$M_NOLABEL, the caller must either own the volume or have VOLPRO privilege.

New Flag	Description
MNT$M_NOREBUILD	The volume to be mounted should be returned to active use immediately, without performing a rebuild operation. If a disk volume is improperly dismounted (such as during a system failure), you must rebuild it to recover any caching limits that were enabled on the volume at the time of the dismount. By default, MOUNT attempts the rebuild. For a successful rebuild operation that includes reclaiming all the available free space, you must mount all of the volume set members. Since the rebuild operation can take a significant amount of time, specifying MNT$M_NOREBUILD is recommended. The volume should be rebuilt at a convenient time using the DCL SET VOLUME/REBUILD command to recover the free space.
MNT$M_NOUNLOAD	The volume to be mounted is not to be unloaded when it is dismounted. Specifying MNT$M_NOUNLOAD causes the volume to remain loaded when it is dismounted unless the dismount explicitly requests that the volume be unloaded.

B.2.5 Modifications to $DISMOUNT

VMS Version 5.2 includes the following changes to the $DISMOUNT system service:

New Flag	Description
DMT$M_NOUNLOAD	Specifies that the volume is not to be physically unloaded after the dismount. If both the DMT$M_UNLOAD and DMT$M_NOUNLOAD flags are specified, the DMT$M_NOUNLOAD flag is ignored. If neither flag is specified, the volume is physically unloaded, unless the DMT$M_NOUNLOAD flag was specified on the $MOUNT system service or the /NOUNLOAD qualifier was specified on the MOUNT command when the volume was mounted.
DMT$M_OVR_CHECKS	Specifies that the volume should be dismounted without checking for open files, spooled devices, installed images, or installed swap and page files.

New Flag	Description
DMT$M_UNIT	The specified device, rather than the entire volume set, is dismounted.
DMT$M_UNLOAD	Specifies that the volume is to be physically unloaded after the dismount. If both the DMT$M_UNLOAD and DMT$M_NOUNLOAD flags are specified, the DMT$M_NOUNLOAD flag is ignored. If neither flag is specified, the volume is physically unloaded, unless the DMT$M_NOUNLOAD flag was specified on the $MOUNT system service or the /NOUNLOAD qualifier was specified on the MOUNT command when the volume was mounted.

B.2.6 Modification to $MOD_IDENT

For Version 5.2 of the VMS operating system, the following status value has been added to the list of Condition Values Returned:

SS$_DUPNAME	The specified identifier name already exists in the rights database.

B.2.7 Modifications to Existing System Services for Clusterwide Process Accessibility

The following system services have been modified because a VMS process is now visible clusterwide:

Modified Service	Function
$CANWAK	Cancel Wakeup
$DELPRC	Delete Process
$FORCEX	Force Image Exit
$GETJPI	Get Job/Process Information
$RESUME	Resume Process
$SCHDWK	Schedule Wakeup for Process
$SETPRI	Set Priority
$SUSPEND	Suspend Process
$WAKE	Wake Process

The descriptions of the **pidadr** and **prcnam** arguments have been changed for these system services. The **pidadr** argument can now refer to a process running on another node in the cluster. The process name can now specify a node name as well as the process name. This full process name can contain up to 23 characters.

In addition, these services now return the following status codes:

Status	Explanation
SS$_INCOMPAT	The remote node is running a version of VMS prior to Version 5.2 and is unable to handle the request.
SS$_NOSUCHNODE	The specified node is not currently a member of the cluster.
SS$_REMRSRC	The remote node has insufficient resources to respond to the request. (Bring this error to the attention of your system manager.)
SS$_UNREACHABLE	The remote node is a member of the cluster but is not accepting requests. (This is normal for a brief period early in the system boot process.)

B.2.8 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 these 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, see the *VMS System Services Reference Manual.* For detailed information about $PROCESS_SCAN, see Section B.2.30.

B.2.9 Overview of $GETJPI and $GETJPI with $PROCESS_SCAN

$GETJPI was previously included with the process control system services. However, because $GETJPI is used to obtain information about processes rather than control processes, $GETJPI is now included with the process information services.

$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 by $GETJPI 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 information is returned only about the processes you have selected. 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 information only about processes running as batch jobs. You can specify a search for all batch processes owned by one user name and $GETJPI returns information only about processes owned by that user name that are running as batch jobs.

B.2.10 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.

It might be preferable to use the **pidadr** argument instead of the **prcnam** argument when specifying a process to $GETJPI, for the following reasons:

- The **pidadr** argument can be used to identify any process in the system, whereas the **prcnam** argument can be used only to identify processes that have the same UIC group number as the caller of $GETJPI.
- $GETJPI executes faster when you use **pidadr** rather than **prcnam**. When you specify **prcnam**, $GETJPI must search a table of process names and UICs for an entry that contains the specified process name and the UIC group number of the calling process; this search is unnecessary when you use **pidadr**.

Table B–3 shows how $GETJPI operates given various values for the **prcnam** and **pidadr** arguments.

Table B–3 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.
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.

B.2.11 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 $GETJPI. See the *Introduction to VMS System Services* for more information about process identification.

B.2.12 Modifications to $GETJPI

You can still use $GETJPI as you did before VMS Version 5.2. However, the $GETJPI system service has been modified to work with $PROCESS_SCAN. If you use $PROCESS_SCAN with $GETJPI, the process context (**pidctx**) is used by $GETJPI as the **pidadr**.

$GETJPI has also been modified to include new status codes and new item codes. The new status codes are described in Section B.2.7. The new item codes are described in the following table:

Item Code	Function
JPI$_CPU_ID	ID of the CPU on which the process is running or on which it last ran, returned as –1 if the system is not a multiprocessor.
JPI$_GETJPI_CONTROL_ FLAGS	Flags for options that control what actions $GETJPI takes to retrieve the information (described in detail in the *Introduction to VMS System Services*).
JPI$_NODENAME	Name of the VAXcluster node on which the process is running.
JPI$_NODE_CSID	Cluster ID of the VAXcluster node on which the process is running.
JPI$_NODE_VERSION	VMS version number of the VAXcluster node on which the process is running.
JPI$_STS2	Second longword of process status flags.
JPI$_TERMINAL	Now returns up to eight bytes of information, including the colon (:) after the device name.
JPI$_TT_ACCPORNAM	Access port name for the terminal associated with the process. (The terminal name is returned by JPI$_ TERMINAL.) If the terminal is on a terminal server, this item returns the terminal server name and the name of the line port on the server. If the terminal is a DECnet remote terminal, this item returns the source system node name and the user name on the source system. Otherwise, it returns a null string.
JPI$_TT_PHYDEVNAM	Physical device name of the terminal associated with the process. This name is the same as JPI$_ TERMINAL unless virtual terminals are enabled, in which case JPI$_TERMINAL returns the name of the virtual terminal and JPI$_TT_PHYDEVNAM returns the name of the physical terminal. If JPI$_TERMINAL is null, or if the virtual terminal is disconnected from the physical terminal, JPI$_TT_PHYDEVNAM returns a null string.

B.2.13 Using $GETJPI Alone

Using $GETJPI without $PROCESS_SCAN 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.

B.2.14 Requesting Information About a Single Process

Example B–1 is a FORTRAN program that displays the process name and the PID of the calling program.

Example B–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
```

(continued on next page)

Example B–1 (Cont.) Using $GETJPI to Obtain Information About the Calling Process

```
        ! 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
        ! 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 B–2 demonstrates (in FORTRAN) how to use the process name to obtain information about a process.

Example B–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 B-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
```

B.2.15 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 for each 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 B–3 is a MACRO program that demonstrates how to use the $GETJPI –1 wildcard to search for all processes on the local system.

Example B–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<>
```

(continued on next page)

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

```
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

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
```

B.2.16 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.

B.2.17 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$M_OR flag is used to connect identical item codes in an item list. For example, in a program that searches for processes owned by several specified users, each user name must be specified in a separate item list entry. The item list entries are connected with the PSCAN$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*.

The PSCAN$M_PREFIX_MATCH flag also allows either the PSCAN$M_EQL or the PSCAN$M_NEQ flag to be specified. If you specify PSCAN$M_NEQ, the service matches those names that do *not* begin with the specified character string.

B.2.18 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 B–4 demonstrates (in FORTRAN) how to perform a $PROCESS_SCAN search on the local node to select all processes that are owned by user SMITH.

Example B–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
        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
```

(continued on next page)

Example B–4 (Cont.) Using $GETJPI and $PROCESS_SCAN to Select Process Information by User Name

```
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

        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
```

(continued on next page)

Example B–4 (Cont.) Using $GETJPI and $PROCESS_SCAN to Select Process Information by User Name

```
        ! 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
```

B.2.19 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 FORTRAN code such as that shown in Example B–5 to initialize the item list in Example B–4.

Example B–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      *
!********************************************
```

B.2.20 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 B–6 demonstrates (in FORTRAN) how to find any batch process owned by either SMITH or JONES. The program uses syntax similar to the following logical expression to initialize the item list:

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

Example B–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
PSCANLIST(3).ITMFLAGS = 0
PSCANLIST(4).END_LIST = 0

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

See Section B.2.30 for more information about $PROCESS_SCAN item codes and flags.

B.2.21 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.

B.2.22 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 FORTRAN code like that in Example B–7 to initialize the item list to find all processes with a nonzero cluster system identifier (CSID) and a user name of SMITH.

Example B–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      *
!********************************************
```

B.2.23 Scanning Specific Nodes on the Cluster for Processes

You can specify a list of nodes as well. Example B–8 demonstrates (in FORTRAN) how to design an item list to search for batch processes on the nodes TIGNES, VALTHO, or 2ALPES.

Example B–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      *
!********************************************
```

B.2.24 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. See the section on asynchronous service completion in the *Introduction to VMS***

***System Services*. 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.

B.2.25 Programming Considerations for GETJPI$

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

B.2.26 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.

B.2.27 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 B–9 demonstrates (in FORTRAN) how to use a $GETJPI buffer to improve performance.

Example B–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      *
!********************************************
```

B.2.28 Meeting Remote $GETJPI Quota Requirements

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

1. Add:
   - 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.

B.2.29 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, $GETJPI might be unable to retrieve all the data requested in an item list because JPI$_GETJPI_CONTROL_FLAGS requests that $GETJPI not perform certain actions that may be necessary to collect the data. For example, a $GETJPI control flag might 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 zero. To verify that $GETJPI received a data item, examine the data length to be sure that it is not zero. 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 adding 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 might 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 might 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 bring a process into memory only 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, are being deleted, or are in miscellaneous wait states.

Example B–10 demonstrates (in FORTRAN) how to use $GETJPI control flags to avoid swapping processes during a $GETJPI call.

Example B–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
```

(continued on next page)

Example B–10 (Cont.) Using $GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
        RECORD /PSCANITMLST/          ! Declare the item list for
        2         PSCANLIST(5)        !  $PROCESS_SCAN

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

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

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

        INTEGER*2 IOSB(4)             ! I/O status block for $GETJPI

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

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

        LOGICAL*4 DONE                ! Done with data loop

        !*******************************************
        !*  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
```

(continued on next page)

Example B–10 (Cont.) Using $GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

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

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

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

        ! Initialize $GETJPI item list

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

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

             ! 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
```

(continued on next page)

Example B–10 (Cont.) Using $GETJPI Control Flags to Avoid Swapping a Process into the Balance Set

```
                ! 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
```

B.2.30 Descriptions of New VMS Version 5.2 System Services

This section contains reference information for two system services, $DEVICE_SCAN, an input/output system service, and $PROCESS_SCAN, a process information system service.

$DEVICE_SCAN Scan for Devices

The Device Scan system service returns the names of all devices that match a specified set of search criteria. The names returned by $DEVICE_SCAN can then be passed to another service, for example, $GETDVI or $MOUNT.

FORMAT

SYS$DEVICE_SCAN *return_devnam ,retlen ,[search_devnam] ,[itmlst] ,[contxt]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed under Condition Values Returned.

ARGUMENTS

return_devnam

VMS usage:	**char_string**
type:	**character-coded text string**
access:	**write only**
mechanism:	**by descriptor–fixed length string descriptor**

Buffer to receive the device name. The **return_devnam** argument is the address of a character string descriptor pointing to a buffer into which $DEVICE_SCAN writes the name of the first or next device that matches the specified search criteria. The maximum size of any device name is 64 bytes.

retlen

VMS usage:	**word_unsigned**
type:	**word (unsigned)**
access:	**write only**
mechanism:	**by reference**

Length of the device name string returned by $DEVICE_SCAN. The **retlen** argument is the address of a word into which $DEVICE_SCAN writes the length of the device name string.

search_devnam

VMS usage:	**device_name**
type:	**character-coded text string**
access:	**read only**
mechanism:	**by descriptor–fixed length string descriptor**

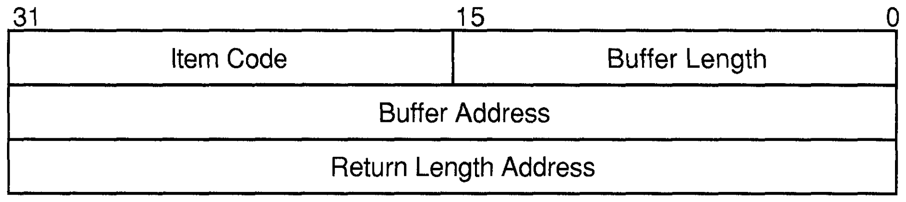
Name of the device for which $DEVICE_SCAN is to search. The **search_devnam** argument accepts the standard wildcard characters, the asterisk (*), which matches any sequence of characters, and the percent sign (%), which matches any one character. For example, to match all unit 0 DU devices on any controller, specify **DU%0*. This string is compared to the most complete device name (DVI$_ALLDEVNAM).

itmlst

VMS usage: **item_list_3**
type: **longword_unsigned**
access: **read only**
mechanism: **by reference**

Item list specifying search criteria used to identify the device names for return by $DEVICE_SCAN. The **itmlst** argument is the address of a list of item descriptors, each of which describes one search criterion. The list of item descriptors is terminated by a longword of 0.

The following figure depicts the format of a single item descriptor:



ZK-1705-GE

$DEVICE_SCAN Item Descriptor Fields

buffer length
A word containing a user-supplied integer specifying the length (in bytes) of the buffer from which $DEVICE_SCAN is to read the information. The length of the buffer needed depends upon the item code specified in the item code field of the item descriptor.

item code
A word containing a user-supplied symbolic code specifying the item of information that $DEVICE_SCAN is to return. The $DVSDEF macro defines these codes. Each item code is described following this list of item descriptor fields.

buffer address
A longword containing the user-supplied address of the buffer from which $DEVICE_SCAN is to read the information.

return length address
This field is not currently used.

$DEVICE_SCAN Item Codes

DVS$_DEVCLASS
An input value item code that specifies, as an unsigned longword, the device class being searched. The $DCDEF macro defines these classes.

The DVS$_DEVCLASS argument is a longword containing this number; however, DVS$_DEVCLASS uses only the low-order byte of the longword.

DVS$_DEVTYPE
An input value item code that specifies, as an unsigned longword, the device type for which $DEVICE_SCAN is going to search. The $DCDEF macro defines these types.

The DVS$_DEVTYPE argument is a longword containing this number; however, DVS$_DEVTYPE uses only the low-order byte of the longword. DVS$_DEVTYPE should be used with $DVS_DEVCLASS to specify the device type being searched for.

contxt

VMS usage:	**quadword_unsigned**
type:	**quadword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Value used to indicate the current position of a $DEVICE_SCAN search. The **contxt** argument is the address of the quadword that receives this information. On the initial call, the quadword should contain 0.

DESCRIPTION

The Device Scan service returns all device names that match a specified set of search criteria. The device names are returned for one process per call. A context value is used to continue multiple calls to $DEVICE_SCAN.

$DEVICE_SCAN allows wildcard searches based on device names, device classes, and device types. It also provides the ability to perform a wildcard search on other device-related services.

$DEVICE_SCAN makes it possible to combine search criteria. For example, to find only RA82 devices use the following selection criteria:

```
DVS$_DEVCLASS = DC$_DISK and DVS$_DEVTYPE = DT$_RA82
```

To find all mailboxes with *MB* as part of the device name (excluding mailboxes such as NLA0), use the following selection criteria:

```
DVS$_DEVCLASS = DC$_MAILBOX and DEVNAM = *MB*
```

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The **search_devnam**, **itmlst**, or **contxt** argument cannot be read by the caller, or the **retlen**, **return_devnam**, or **contxt** argument cannot be written by the caller.
SS$_BADPARAM	The **contxt** argument contains an invalid value or the item list contains an invalid item code.
SS$_NOSUCHDEV	The specified device does not exist on the host system.
SS$_NOMOREDEV	No more devices match the specified search criteria.

$PROCESS_SCAN Process Scan

The Process Scan system service creates and initializes a process context that is used by $GETJPI to scan processes on the local system or across the nodes in a VAXcluster system. An item list is used to specify selection criteria to obtain information about specific processes, for example, all processes owned by one user or all batch processes.

FORMAT

SYS$PROCESS_SCAN *pidctx [,itmlst]*

RETURNS

VMS usage:	**cond_value**
type:	**longword (unsigned)**
access:	**write only**
mechanism:	**by value**

Longword condition value. All system services (except $EXIT) return by immediate value a condition value in R0. Condition values returned by this service are listed under Condition Values Returned.

ARGUMENTS

pidctx

VMS usage:	**process_id**
type:	**longword (unsigned)**
access:	**modify**
mechanism:	**by reference**

Context value supplied by $PROCESS_SCAN to be used as the **pidadr** argument of $GETJPI. The **pidctx** argument is the address of a longword that is to receive the process context longword. This longword normally contains zero or a previous context. If it contains a previous context, the old context is deleted. If it contains a value other than zero or a previous context, the old value is ignored.

itmlst

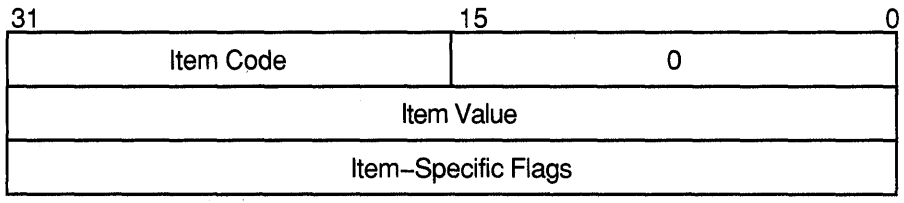
VMS usage:	**item_list_3**
type:	**longword (unsigned)**
access:	**read only**
mechanism:	**by reference**

Item list specifying selection criteria to be used by the scan or to control the scan.

The **itmlst** argument is the address of a list of item descriptors, each of which describes one selection criterion or control option. Within each selection criterion you can include several item entries. The list of item descriptors is terminated by a longword of zero.

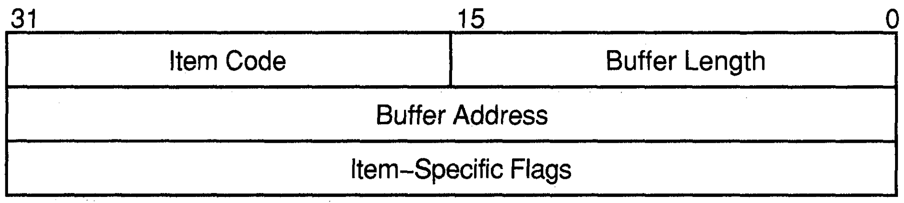
The information in the item list is passed to the item descriptor in one of two ways. If the item descriptor can always hold the actual value of the selection criterion, the value is placed in the second longword of the item descriptor and the buffer length is specified as zero. If the item descriptor points to the actual value of the selection criterion, the address of the value is placed in the second longword of the item descriptor and you must specify the buffer length for the selection criterion. Each item code description specifies whether the information is passed by value or by reference.

The following figure depicts the format of an item descriptor that passes the selection criterion as a value:



ZK–0949A–GE

The following figure depicts the format of an item descriptor that passes the selection criterion by reference:



ZK–0948A–GE

$PROCESS_SCAN Item Descriptor Fields

buffer address

A longword containing the user-supplied address of the buffer from which $PROCESS_SCAN retrieves information needed by the scan. When you specify an item code that is passed by reference, $PROCESS_SCAN uses the address as a pointer to the actual value. See the description of the **item value** field for information about item codes that are passed by value.

buffer length

Buffer length is specified in a different way for the two types of item descriptors:

- Character string or reference descriptors

  A word containing a user-supplied integer specifying the length (in bytes) of the buffer from which $PROCESS_SCAN retrieves a selection criterion. The length of the buffer needed depends upon the item code specified in the item descriptor.

- Immediate value descriptors

  The length of the buffer is always specified as zero.

item code
A word containing the selection criterion. These codes are defined by the $PSCANDEF macro. Each item code is described after this list of descriptor fields.

item value
A longword containing the actual value of the selection criterion. When you specify an item code that is passed by value, $PROCESS_SCAN searches for the actual value contained in the item list. See the description of the **buffer address** field for information about item codes that are passed by reference.

item-specific flags
A longword that contains flags to help control selection information. Item-specific flags, for example EQL or NEQ, are used to specify how the value specified in the item descriptor is compared to the process value.

These flags are defined by the $PSCANDEF macro. Some flags are common to multiple item codes; other flags are specific to an individual item code. See the description of each item code to determine which flags are used.

For item codes that describe bit masks or character strings, these flags control how the bit mask or character string is compared with that in the process. By default, they are compared for equality.

For item codes that describe integers, these flags specify an arithmetic comparison of an integer item with the process attribute. For example, a PSCAN$M_GTR selection specifying the value 4 for the item code PSCAN$_PRIB finds only the processes with a base priority above 4. Without one of these flags, the comparison is for equality.

$PROCESS_SCAN Item Codes

PSCAN$_ACCOUNT

When you specify PSCAN$_ACCOUNT, $GETJPI returns information about processes that match the account field.

If the string supplied in the item descriptor is shorter than the account field, the string is padded with blanks for the comparison unless the item-specific flag PSCAN$M_PREFIX_MATCH is present.

Because the information is a character string, the selection value is passed by reference. The length of the buffer is placed in the first word of the item descriptor and the address of the buffer is placed in the second longword.

Although the current length of the account field is eight bytes, the PSCAN$_ACCOUNT buffer can be up to 64 bytes in length. If the buffer length is zero or greater than 64, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

Item-Specific Flag	Description
PSCAN$M_OR	Match this value or the next value
PSCAN$M_EQL	Match value exactly (the default)

Item-Specific Flag	Description
PSCAN$M_NEQ	Match if value is not equal
PSCAN$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

PSCAN$_AUTHPRI

When you specify PSCAN$_AUTHPRI, $GETJPI returns information about processes that match the authorized base priority field.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_CURPRIV

When you specify PSCAN$_CURPRIV, $GETJPI returns information about processes that match the current privilege field. Privilege bits are defined by the $PRVDEF macro.

Because the bit mask information is too long to be passed by value, the information is passed by reference. The privilege buffer must be exactly eight bytes; otherwise, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

Item-Specific Flag	Description
PSCAN$M_OR	Match this value or the next value
PSCAN$M_EQ	Match value exactly (the default)
PSCAN$M_NEQ	Match if value is not equal
PSCAN$M_BIT_ALL	All bits set in pattern set in target
PSCAN$M_BIT_ANY	Any bit set in pattern set in target

PSCAN$_GETJPI_BUFFER_SIZE

When you specify PSCAN$_GETJPI_BUFFER_SIZE, you determine the size of a buffer to be used by $GETJPI to process multiple requests in a single message. Using this item code can greatly improve the performance of scans on remote nodes because fewer messages are needed. This item code is ignored during scans on the local node.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero. The buffer is allocated by $PROCESS_SCAN; you do not have to allocate a buffer.

If you use PSCAN$_GETJPI_BUFFER_SIZE with $PROCESS_SCAN, all calls to $GETJPI using the context established by $PROCESS_SCAN must request the same item code information. Because $GETJPI locates information for more than one process at a time, it is not possible to change the item codes or the length of the buffers used in the $GETJPI item list. $GETJPI checks each call and returns the error SS$_BADPARAM if an attempt is made to change the item list during a buffered process scan. However, the buffer addresses can be changed between $GETJPI calls.

Because the locating and buffering of information by $GETJPI is transparent to a calling program, you are not required to change the way $GETJPI is called when you use this item code.

The $GETJPI buffer uses the process quota BYTLM. If the buffer is too large for the process quota, $GETJPI (not $PROCESS_SCAN) returns the error SS$_EXBYTLM. If the buffer specified is not large enough to contain the data for at least one process, $GETJPI returns the error SS$_BADPARAM.

No item-specific flags are used with PSCAN$_GETJPI_BUFFER_SIZE.

PSCAN$_GRP

When you specify PSCAN$_GRP, $GETJPI returns information about processes that match the UIC group number.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. Because the value of the group number is a word, the high-order word of the value is ignored. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_HW_MODEL

When you specify PSCAN$_HW_MODEL, $GETJPI returns information about processes that match the specified CPU hardware model number.

The hardware model number is an integer, such as VAX$K_V8840. The VAX$ symbols are defined by the $VAXDEF macro.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_HW_NAME
When you specify PSCAN$_HW_NAME, $GETJPI returns information about processes that match the specified CPU hardware name, such as VAX 11/780, VAX 8800, or VAXstation II/GPX.

Because the information is a character string, the selection value is passed by reference. The length of the selection value is placed in the first word of the item descriptor and the address of the buffer is placed in the second longword.

The PSCAN$_HW_NAME buffer can be up to 128 bytes in length. If the buffer length is zero or greater than 128, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

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_CASE_BLIND	Match without regard to case of letters
PSCAN$M_PREFIX_MATCH	Match on the leading substring
PSCAN$M_WILDCARD	Match a wildcard pattern

PSCAN$_JOBPRCCNT
When you specify PSCAN$_JOBPRCCNT, $GETJPI returns information about processes that match the subprocess count for the job (the count of all subprocesses in the job tree).

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

Item-Specific Flag	Description
PSCAN$M_OR	Match this value or the next value
PSCAN$M_EQL	Match value exactly (the default)

Item-Specific Flag	Description
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$_JOBTYPE
When you specify PSCAN$_JOBTYPE, $GETJPI returns information about processes that match the job type. The job type values include the following:

Value	Description
JPI$K_LOCAL	Local interactive process
JPI$K_DIALUP	Interactive process accessed by a modem line
JPI$K_REMOTE	Interactive process accessed by using SET HOST
JPI$K_BATCH	Batch process
JPI$K_NETWORK	Noninteractive network process
JPI$K_DETACHED	Detached process

These values are defined by the $JPIDEF macro. Note that job type values are similar to mode values. See PSCAN$_MODE.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_MASTER_PID
When you specify PSCAN$_MASTER_PID, $GETJPI returns information about processes that are descendants of the specified parent process. The master process is the first process created in the job tree. The PSCAN$_OWNER item is similar, but the owner process is the process that created the target process (the owner process might itself be a subprocess). Although all jobs in a job tree must have the same master, they can have different owners.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_MEM

When you specify PSCAN$_MEM, $GETJPI returns information about processes that match the UIC member number.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. Because the value of the member number is a word, the high-order word of the value is ignored. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_MODE

When you specify PSCAN$_MODE, $GETJPI returns information about processes that match the specified mode. Mode values include the following:

Value	Description
JPI$K_INTERACTIVE	Interactive process
JPI$K_BATCH	Batch job
JPI$K_NETWORK	Noninteractive network job
JPI$K_OTHER	Detached and other process

These values are defined by the $JPIDEF macro. Note that values checked by PSCAN$_MODE are similar to PSCAN$_JOBTYPE values.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_NODE_CSID

When you specify PSCAN$_NODE_CSID, $GETJPI returns information about processes on the specified nodes. To scan all nodes in a VAXcluster system, you specify a CSID of zero and the item-specific flag PSCAN$M_NEQ.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_NODENAME

When you specify PSCAN$_NODENAME, $GETJPI returns information about processes that match the specified node names. To scan all of the nodes in a VAXcluster system, specify the node name using an asterisk wildcard (*) and the PSCAN$M_WILDCARD item-specific flag.

Because the information is a character string, the selection value is passed by reference. The length of the selection value is placed in the first word of the item descriptor and the address of the buffer is placed in the second longword.

Although the current length of the node name is 6 bytes, the PSCAN$_NODENAME buffer can be up to 64 bytes in length. If the buffer length is zero or greater than 64, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

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_CASE_BLIND	Match without regard to case of letters
PSCAN$M_PREFIX_MATCH	Match on leading substring
PSCAN$M_WILDCARD	Match a wildcard pattern

PSCAN$_OWNER

When you specify PSCAN$_OWNER, $GETJPI returns information about processes that are immediate descendants of the specified process. The PSCAN$_MASTER_PID item is similar, but the owner process is the process that created the target process (the owner process might itself be a subprocess). Although all jobs in a job tree must have the same master, they can have different owners.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_PRCCNT

When you specify PSCAN$_PRCCNT, $GETJPI returns information about processes that match the subprocess count (the count of all immediate descendants of a given process). The PSCAN$_JOBPRCCNT item code is similar, except that JOBPRCCNT is the count of all subprocesses in a job.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_PRCNAM

When you specify PSCAN$_PRCNAM, $GETJPI returns information about processes that match the specified process names.

The process name string is padded with blanks for the comparison unless the item-specific flag PSCAN$M_PREFIX_MATCH is present.

Because the information is a character string, the selection value is passed by reference. The length of the selection value is placed in the first word of the item descriptor and the address of the buffer is placed in the second longword.

Although the current length of the process name field is 15 bytes, the PSCAN$_PRCNAM buffer can be up to 64 bytes in length. If the buffer length is zero or greater than 64, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

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_CASE_BLIND	Match without regard to case of letters
PSCAN$M_PREFIX_MATCH	Match on leading substring
PSCAN$M_WILDCARD	Match a wildcard pattern

PSCAN$_PRI

When you specify PSCAN$_PRI, $GETJPI returns information about processes that match current priority. Note that the current priority of a process can be temporarily increased as a result of system events such as the completion of I/O.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_PRIB

When you specify PSCAN$_PRIB, $GETJPI returns information about processes that match base priority.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

Item-Specific Flag	Description
PSCAN$M_OR	Match this value or the next value

Item-Specific Flag	Description
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$_STATE
When you specify PSCAN$_STATE, $GETJPI returns information about processes that match the specified process state. State values, for example SCH$C_COM and SCH$C_PFW, are defined by the $STATEDEF macro.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_STS
When you specify PSCAN$_STS, $GETJPI returns information that matches the current status mask. Without any item-specific flags, the match is for a process mask that is equal to the pattern. Status bits, for example PCB$V_ASTPEN or PCB$V_PSWAPM, are defined by the $PCBDEF macro.

This bit mask item code uses an immediate value descriptor; the selection value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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_BIT_ALL	All bits set in pattern set in target
PSCAN$M_BIT_ANY	Any bit set in pattern set in target

PSCAN$_TERMINAL
When you specify PSCAN$_TERMINAL, $GETJPI returns information that matches the specified terminal names. The terminal name string is padded with blanks for the comparison unless the item-specific flag PSCAN$M_PREFIX_MATCH is present.

Because the information is a character string, the selection value is passed by reference. The length of the selection value is placed in the first word of the item descriptor and the address of the buffer is placed in the second longword.

Although the current length of the terminal name field is 8 bytes, the PSCAN$_TERMINAL buffer can be up to 64 bytes in length. If the buffer length is zero or greater than 64, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

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_CASE_BLIND	Match without regard to case of letters
PSCAN$M_PREFIX_MATCH	Match on leading substring
PSCAN$M_WILDCARD	Match a wildcard pattern

PSCAN$_UIC

When you specify PSCAN$_UIC, $GETJPI returns information about processes that match the UIC identifier. To convert an alphanumeric identifier name to the internal identifier, use the $ASCTOID system service before calling $PROCESS_SCAN.

This integer item code is passed by value; the value is placed in the second longword of the item descriptor. The buffer length must be specified as zero.

The following flags can be used with this item code:

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$_USERNAME

When you specify PSCAN$_USERNAME, $GETJPI returns information about processes that match the specified user name.

The user name string is padded with blanks for the comparison unless the item-specific flag PSCAN$M_PREFIX_MATCH is present.

Because the information is a character string, the selection value is passed by reference. The length of the selection value is placed in the first word of the item descriptor and the address of the buffer is placed in the second longword.

Although the current length of the username field is 12 bytes, the PSCAN$_USERNAME buffer can be up to 64 bytes in length. If the buffer length is zero or greater than 64, the SS$_IVBUFLEN error is returned.

The following flags can be used with this item code:

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_CASE_BLIND	Match without regard to case of letters
PSCAN$M_PREFIX_MATCH	Match on leading substring
PSCAN$M_WILDCARD	Match a wildcard pattern

The following flags can be furnished in the item-specific flag field of the item descriptor.

$PROCESS_SCAN Item-Specific Flags

PSCAN$M_BIT_ALL

If the PSCAN$M_BIT_ALL flag is used, all bits set in the pattern mask specified by the item descriptor must also be set in the process mask. Other bits in the process mask may also be set.

For item codes that describe bit masks, such as privilege masks and status words, this flag controls how the pattern bit mask specified by the item descriptor is compared with that in the process. By default, the bit masks are compared for equality.

The PSCAN$M_BIT_ALL flag is used only with bit masks.

PSCAN$M_BIT_ANY

If the PSCAN$M_BIT_ANY flag is used, a match occurs if any bit in the pattern mask is also set in the process mask.

For item codes that describe bit masks, such as privilege masks and status words, this flag controls how the pattern bit mask specified by the item descriptor is compared with that in the process. By default, the bit masks are compared for equality.

The PSCAN$M_BIT_ANY flag is used only with bit masks.

PSCAN$M_CASE_BLIND

When you specify PSCAN$M_CASE_BLIND to compare the character string specified by the item descriptor with the character string value from the process, $PROCESS_SCAN does not distinguish between uppercase and lowercase letters.

The PSCAN$M_CASE_BLIND flag is used only with character-string item codes. The PSCAN$M_CASE_BLIND flag can be specified with either the PSCAN$M_PREFIX_MATCH flag or the PSCAN$M_WILDCARD flag.

PSCAN$M_EQL

When you specify PSCAN$M_EQL, $PROCESS_SCAN compares the value specified by the item descriptor with the value from the process to see if there is an exact match.

PSCAN$M_EQL and PSCAN$M_NEQ are used with bit masks, character strings, and integers to control how the item is interpreted. Only one of the flags can be specified; if more than one of these flags is used the SS$_IVSSRQ error is returned. If you want to specify that bits not set in the pattern mask must not be set in the process mask, use PSCAN$M_EQL.

PSCAN$M_GEQ

When you specify PSCAN$M_GEQ, $PROCESS_SCAN selects a process if the value from the process is greater than or equal to the value specified by the item descriptor.

PSCAN$M_GEQ, PSCAN$M_GTR, PSCAN$M_LEQ, and PSCAN$M_LSS are used with integer item codes only. Only one of these four flags can be specified; if more than one of these flags is used the SS$_IVSSRQ error is returned.

PSCAN$M_GTR

When you specify PSCAN$M_GTR, $PROCESS_SCAN selects a process if the value from the process is greater than the value specified by the item descriptor.

PSCAN$M_GEQ, PSCAN$M_GTR, PSCAN$M_LEQ, and PSCAN$M_LSS are used with integer item codes only. Only one of these four flags can be specified; if more than one of these flags is used the SS$_IVSSRQ error is returned.

PSCAN$M_LEQ

When you specify PSCAN$M_LEQ, $PROCESS_SCAN selects a process if the value from the process is less than or equal to the value specified by the item descriptor.

PSCAN$M_GEQ, PSCAN$M_GTR, PSCAN$M_LEQ, and PSCAN$M_LSS are used with integer item codes only. Only one of these four flags can be specified; if more than one of these flags is used the SS$_IVSSRQ error is returned.

PSCAN$M_LSS

When you specify PSCAN$M_LSS, $PROCESS_SCAN selects a process if the value from the process is less than the value specified by the item descriptor.

PSCAN$M_GEQ, PSCAN$M_GTR, PSCAN$M_LEQ, and PSCAN$M_LSS are used with integer item codes only. Only one of these four flags can be specified; if more than one of these flags is used the SS$_IVSSRQ error is returned.

PSCAN$M_NEQ

When you specify PSCAN$M_NEQ, $PROCESS_SCAN selects a process if the value from the process is not equal to the value specified by the item descriptor.

PSCAN$M_EQL and PSCAN$M_NEQ are used with bit masks, character strings, and integers to control how the item is interpreted. Only one of the flags can be specified; if more than one of these flags is used the SS$_IVSSRQ error is returned.

PSCAN$M_OR

When you specify PSCAN$M_OR, $PROCESS_SCAN selects processes whose values match the current item descriptor or the next item descriptor. The next item descriptor must have the same item code as the item descriptor with the PSCAN$M_OR flag. Multiple items are chained together; all except the last item descriptor must have the PSCAN$M_OR flag.

The PSCAN$M_OR flag can be specified with any other flag and can be used with bit masks, character strings, and integers. If the PSCAN$M_OR flag is used between different item codes or if it is missing between identical item codes, the SS$_IVSSRQ error is returned.

PSCAN$M_PREFIX_MATCH

When you specify PSCAN$M_PREFIX_MATCH, $PROCESS_SCAN compares the character string specified in the item descriptor to the leading characters of the requested process value.

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.

The PSCAN$M_PREFIX_MATCH flag also allows either the PSCAN$M_EQL or the PSCAN$M_NEQ flag to be specified. If you specify PSCAN$M_NEQ, the service matches those names that do *not* begin with the specified character string.

The PSCAN$M_PREFIX_MATCH is used only with character-string item codes. The PSCAN$M_PREFIX_MATCH flag cannot be specified with the PSCAN$M_WILDCARD flag; if both of these flags are used the SS$_IVSSRQ error is returned.

PSCAN$M_WILDCARD

When you specify PSCAN$M_WILDCARD, the character string specified by the item descriptor is assumed to be a wildcard pattern. Acceptable wildcard characters are the asterisk (*), which allows the match to substitute any number of characters in place of the asterisk and the percent sign (%), which allows the match to substitute any one character in place of the percent sign. For example, if you want to search for all process names that begin with the letter *A* and end with the string *ER*, use the string A*ER with the PSCAN$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$M_WILDCARD is used only with character-string item codes. The PSCAN$M_WILDCARD flag cannot be specified with the PSCAN$M_PREFIX_MATCH flag; if both of these flags are used the SS$_IVSSRQ error is returned. The PSCAN$M_NEQ flag can be used with PSCANM$_WILDCARD to exclude values during a wildcard search.

DESCRIPTION

The Process Scan system service creates and initializes a process context that is used by $GETJPI to scan processes on the local system or across the nodes in a VAXcluster system. An item list is used to specify selection criteria to obtain information about specific processes, for example, all processes owned by one user or all batch processes.

The output of the $PROCESS_SCAN service is a process context longword named **pidctx**. This process context is then provided to $GETJPI as the **pidadr** argument. The process context provided by $PROCESS_SCAN enables $GETJPI to search for processes across the nodes in a VAXcluster system and to select processes that match certain selection criteria.

The process context consumes process dynamic memory. This memory is deallocated when the end of the context is reached. For example, when the $GETJPI service returns SS$_NOMOREPROC or when $PROCESS_SCAN is called again with the same **pidctx** longword, the dynamic memory is deallocated. If you anticipate that a scan might be interrupted before it runs out of processes, $PROCESS_SCAN should be called a second time (without an **itmlst** argument) to release the memory. Dynamic memory is automatically released when the current image terminates.

$PROCESS_SCAN copies the item list and user buffers to the allocated dynamic memory. This means that the item lists and user buffers can be deallocated or reused immediately; they are not referenced during the calls to $GETJPI.

The item codes referenced by $PROCESS_SCAN are found in data structures that are always resident in the system, primarily the process control block (PCB) and the job information block (JIB). A scan of processes never forces a process that is swapped out of memory to be brought into memory to read nonresident information.

CONDITION VALUES RETURNED

SS$_NORMAL	The service completed successfully.
SS$_ACCVIO	The **pidctx** argument cannot be written by the caller, the item list cannot be read by the caller, or a buffer for a reference descriptor cannot be read.
SS$_BADPARAM	The item list contains an invalid item identifier or an invalid combination of item-specific flags is present.
SS$_IVBUFLEN	The buffer length field is invalid. For immediate value descriptors, the buffer length must be zero. For reference descriptors, the buffer length cannot be zero or longer than the maximum for the specified item code. This error is also returned if the total length of the item list plus the length of all of the buffer fields is too large to process.
SS$_IVSSRQ	The **pidctx** argument was not supplied, or the item list is improperly formed (for example, multiple occurrences of a given item code were interspersed with other item codes).

C VMS Version 5.1 Features

This appendix describes features that were new to Version 5.1 of the VMS operating system but are not yet documented in other printed manuals.

C.1 VMS Version 5.1 Support for Compound Documents

The term **compound documents** refers to files that can contain a number of integrated components including text, graphics, and scanned images. This appendix specifically describes VMS support for using the text from DECwindows compound documents that are structured according to the DIGITAL Document Interchange Format (DDIF) specification. Refer to the *Introduction to the CDA Services* and *CDA Reference Manual* for more information about compound documents.

VMS commands and utilities, as well as existing application programs that accept text input, can now use the text content of DECwindows compound documents.

To support the use of DDIF text, VMS RMS has implemented a new RMS file attribute, **stored semantics,** and a DDIF-to-text **RMS extension**. The value of the stored semantics attribute is called the file **tag** and it specifies how file data is to be interpreted. When file data is to be interpreted in accordance with the DDIF specification, the appropriate file tag is DDIF. The use of file tags is limited to disk files on VMS Version 5.1 and later systems.

The DDIF-to-text RMS extension transparently extracts text from DDIF files as variable-length text records that can be accessed through the VMS RMS interface.

The enhancements made to support the reading of text from DDIF files are transparent to the user and to the application programmer. This support requires that all DDIF files in a VMS Version 5.1 environment be tagged with the DDIF file tag. DDIF files created by VMS and VMS layered products are tagged appropriately.

Section C.1.1 describes various VMS_file management commands and utilities that display, create, and preserve file tags where appropriate. Section C.1.1 also describes the way various VMS commands and utilities respond to DDIF file input. Section C.1.2 describes VMS support for DDIF files in heterogeneous computing environments. Section C.1.3 describes the changes made to the VMS RMS program interface to support the stored semantics attribute and to control access to the content of DDIF files.

C.1.1 VMS Commands and Utilities

This section describes the VMS commands and utilities that support tag maintenance by displaying, creating, and preserving the RMS file tags used with DDIF files. It also provides additional information that is relevant to the way selected VMS commands and utilities respond to DDIF file input.

The following table lists the VMS commands and utilities that support tag maintenance:

Command/Utility	Tag Maintenance Function
DIRECTORY/FULL	Displays file tag
ANALYZE/RMS_FILE	Displays file tag
SET FILE/SEMANTICS	Creates file tag
VMS MAIL	Preserves file tag†
COPY	Preserves file tag†
BACKUP	Preserves file tag

†See text for exceptions.

Tags are made up of binary values that can be up to 64 bytes long and can be expressed using hexadecimal notation. The hexadecimal value of the DDIF tag, for example, is 2B0C8773010301. VMS permits you to assign mnemonics to tag values so that DCL commands, such as DIRECTORY/FULL, and VMS utilities, such as FDL and ANALYZE/RMS_FILE, display a mnemonic for the DDIF tag instead of the hexadecimal value. The following DCL commands have been included in the system startup command file to assign the mnemonic DDIF to the hexadecimal value for a DDIF tag:

```
$ DEFINE/TABLE=RMS$SEMANTIC_TAGS DDIF 2B0C8773010301
$ DEFINE/TABLE=RMS$SEMANTIC_OBJECTS 2B0C8773010301 DDIF
```

Using the appropriate DEFINE commands, you can assign mnemonics for other tags, including tags used with international program applications.

C.1.1.1 Displaying RMS File Tags

The DIRECTORY/FULL command and the Analyze/RMS_File Utility now display the RMS file tag for DDIF files.

C.1.1.1.1 DIRECTORY/FULL

Where applicable, the DIRECTORY/FULL command now provides the value of the stored semantics tag as part of the file information returned to the user. This is the recommended method for quickly determining whether or not a file is tagged. The following display illustrates how the DIRECTORY/FULL command returns the RMS attributes for a DDIF file named X.DDIF:

```
X.DDIF;1                         File ID:  (767,20658,0)
.
.
.
RMS attributes:     Stored semantics: DDIF
.
.
.
```

C.1.1.1.2 ANALYZE/RMS_FILE

When you use the ANALYZE/RMS_FILE command to analyze a DDIF file, the utility returns the file tag as an RMS file attribute.

```
        FILE HEADER
        File Spec: USERD$:[TEST]X.DDIF;1
    .
    .
    .
      Stored semantics: DDIF
    .
    .
    .
```

One ANALYZE/RMS_FILE command option is to create an output FDL file that reflects the results of the analysis, using the following format:

ANALYZE/RMS_FILE/FDL filespec

When you use this option for analyzing a tagged file, the output FDL file includes the file tag as a secondary attribute to the FILE primary attribute. This is illlustrated in the following FDL file excerpt:

```
IDENT    " 9-JUN-1989 13:27:30  VAX/VMS ANALYZE/RMS_FILE Utility"
    .
    .
    .
SYSTEM
        SOURCE                  VMS
FILE
        ALLOCATION              3
    .
    .
    .
        STORED_SEMANTICS        %X'2B0C8773010301' ! DDIF
    .
    .
    .
```

C.1.1.2 Creating RMS File Tags

The CDA$CREATE_FILE routine in the Compound Document Architecture toolkit creates and tags DDIF files. However, you might encounter a DDIF file that was created without a file tag or a DDIF file whose file tag was not preserved during file processing.

The DCL command SET FILE provides a qualifier, /[NO]SEMANTICS, that permits you to tag a DDIF file through the DCL interface for VMS Version 5.1 or later systems. You can also use the qualifier to change a tag or to remove a tag from a file.

The following command line tags the file X.DDIF as a DDIF file by assigning the appropriate value to the /SEMANTICS qualifier:

```
$ SET FILE X.DDIF/SEMANTICS=DDIF
```

See Section C.1.1 for information about how to use logical name tables to assign a mnemonic to a tag.

A subsequent DIRECTORY/FULL command displays the following line as part of the file header:

```
   .
   .
   .
 RMS attributes:     Stored semantics: DDIF
   .
   .
   .
```

The next example illustrates how to use the SET FILE command to delete an RMS file tag:

```
$ SET FILE X.DDIF/NOSEMANTICS
```

C.1.1.3 Preserving RMS File Tags and DDIF Semantics

The COPY command and the VMS Mail Utility preserve RMS file tags and DDIF semantics when you copy or mail a DDIF file on a VMS Version 5.1 or later system, except for conditions described in Sections C.1.2.2, C.1.2.3, and C.1.2.4.

The Backup Utility always preserves file tags and semantics when you back up a DDIF file to magnetic tape.

C.1.1.3.1 COPY Command

This section describes the results of using the COPY command with DDIF files for various operations.

When you use the COPY command to copy a DDIF file to a disk on a VMS Version 5.1 or later system, VMS RMS preserves the DDIF tag and the DDIF semantics of the input file in the output file.

When you use the COPY command to copy a DDIF file to a nondisk device on a VMS Version 5.1 or later system, VMS RMS does *not* preserve the DDIF tag or the DDIF semantics of the input file in the output file. Instead, VMS RMS writes the text from the input file to the output file as variable-length records.

When you copy two or more DDIF and text files in any combination to a single output file, the output file takes the characteristics of the first input file, as shown in the following examples:

1 In this example, the first input file is a text file, so the output file (FOO.TXT) contains variable-length text records from X.TXT, Y.DDIF, and Z.TXT but does not include the DDIF tag from Y.DDIF.

   ```
   $ COPY X.TXT,Y.DDIF,Z.TXT FOO.TXT
   ```

2 In this example, the first input file (A.DDIF) is a DDIF file, so the output file (FOO.DDIF) includes the DDIF tag as well as the DDIF semantics from A.DDIF. The attempt to copy the text input file (Z.TXT) fails because there is no text-to-DDIF RMS extension, but the contents of B.DDIF and C.DDIF are copied to the output file. However, the output file has no practical use because, as a result of the way DDIF files are structured, only the data from the first input file (A.DDIF) is accessible in the output file.

```
$ COPY A.DDIF,B.DDIF,Z.TXT,C.DDIF FOO.DDIF
```

3 In this example, the first input file (A.DDIF) is a DDIF file, so the output file (FOO.DDIF) includes the DDIF tag as well as the contents of A.DDIF. FOO.DDIF also includes the contents of B.DDIF and C.DDIF. Again, however, the output file has no practical use because, as a result of the way DDIF files are structured, only the data from the first input file (A.DDIF) is accessible in the output file.

```
$ COPY A.DDIF,B.DDIF,C.DDIF FOO.DDIF
```

C.1.1.3.2 VMS Mail Utility

The VMS Mail Utility preserves the DDIF file tag when DDIF files are mailed between VMS Version 5.1 or later systems. The VMS Mail Utility also preserves the DDIF file tag when you create an output file on a VMS Version 5.1 or later system using the EXTRACT command.

When you read a mail message that is a DDIF file, the VMS Mail Utility outputs only the text portion of the file. Similarly, if you edit a DDIF mail file, you can access only the file text; the output file is a text file that can no longer be used as a DDIF file. However, if you forward a message that consists of a DDIF file, the VMS Mail Utility sends the entire DDIF file, including DDIF semantics and the DDIF tag, to the addressee.

C.1.1.4 APPEND Command

This section describes what happens when you attempt to use the APPEND command with DDIF and text files.

In the first example, the APPEND command appends a DDIF file to a text file:

```
$ APPEND X.DDIF Y.TXT
```

The output file, Y.TXT, contains its original text records as well as text from the input file, X.DDIF, reformatted as variable-length text records.

In the next example, the APPEND command appends a DDIF file to another DDIF file:

```
$ APPEND X.DDIF Y.DDIF
```

The output file, Y.DDIF, contains the DDIF tag, the original contents of Y.DDIF, and the contents of X.DDIF. However, the portion of the file that contains X.DDIF is not accessible because of the way DDIF files are structured.

In the final example, the APPEND command attempts to append a text file to a DDIF file:

```
$ APPEND X.TXT Y.DDIF
```

This append operation fails because there is no text-to-DDIF RMS extension.

C.1.2 DDIF Support in a Heterogeneous Environment

This section describes the implementation of DDIF support in two heterogeneous environments. The first heterogeneous environment includes VMS Version 5.1 or later systems and non-VMS systems. The second heterogeneous environment includes VMS Version 5.1 or earlier systems.

C.1.2.1 EXCHANGE/NETWORK Command

A new DCL command, EXCHANGE/NETWORK, has been created to support the transfer of files between VMS systems and non-VMS systems that do not support VMS file types. The EXCHANGE/NETWORK command transfers files in either record mode or block mode but can be used only when both systems support DECnet file transfers.

To interactively tag a DDIF file and transfer the file between a non-VMS operating system and a VMS Version 5.1 or later system, do the following:

1 Create the following file, assigning it the name DDIF.FDL:

```
FILE
        ORGANIZATION            sequential
        STORED_SEMANTICS        DDIF
RECORD
        CARRIAGE_CONTROL        none
        FORMAT                  fixed
        SIZE                    512
```

2 To transfer the desired file, enter the EXCHANGE/NETWORK command, using the following format:

```
EXCHANGE/NETWORK/FDL=DDIF.FDL input_filespec output_filespec
```

See Section C.2 for more information about the EXCHANGE/NETWORK command.

C.1.2.2 COPY Command

If you use the COPY command to copy tagged DDIF files to systems other than VMS Version 5.1 systems from a VMS Version 5.1 system, the results will vary depending on the target system:

- If the target system is a non-VMS system, the file is copied, but the DDIF tag is not preserved.
- If the target system is a VMS Version 5.1 or earlier system, the copy operation fails with the VMS RMS error message RMS$_SUPPORT, network operation not supported, and a secondary error message of RMS$_SEMANTICS, inconsistent usage of RMS Semantics. Error messages similiar to the following will appear:

```
%COPY-E-OPENOUT, error opening PWEDGE::[]TRY.DDIF;1 as output
-RMS-F-SUPPORT, network operation not supported
-RMS-E-SEMANTICS, inconsistent usage of RMS Semantics
%COPY-W-NOTCOPIED, ABCD4:[DAVIDS]TRY.DDIF;1 not copied
```

- If the target system is a cluster alias for a mixed version cluster containing Version 5.1 or earlier systems, the result of the copy operation depends on whether the cluster node that actually handles the request is a Version 5.1 or earlier system.
- If you use the COPY command to copy tagged DDIF files from Version 5.1 or later systems to earlier systems while on an earlier system, the copy operation will fail with the error message RMS$_NET, network operation failed at remote node, and with a DAP status code of 16F, inconsistent usage of RMS Semantics. Error messages similiar to the following will appear:

```
%COPY-E-OPENIN, error opening ARC"davids password"::ABCD4:[DAVIDS]TRY.DDIF;1 as
input
-RMS-F-NET, network operation failed at remote node; DAP code = 01F7516F
%COPY-W-NOTCOPIED, ARC"davids password"::ABCD4:[DAVIDS]TRY.DDIF;1 not copied
PWEDGE$
```

C.1.2.3 VMS Mail Utility

If you try to send mail messages containing DDIF files to non-VMS systems that do not support tagged files, the VMS Mail Utility returns the NOACCEPTMSG error message, indicating that the remote node cannot accept the message format.

Similarly, the VMS Mail Utility does not support the mailing of DDIF files to systems earlier than Version 5.1. As with non-VMS systems, the VMS Mail Utility returns the NOACCEPTMSG error message for systems earlier than Version 5.1, indicating that the remote node cannot accept the message format.

C.1.2.4 DDIF File Access Within a Mixed Version Cluster

In a cluster that contains both Version 5.1 or earlier systems, operations on DDIF files from systems earlier than Version 5.1 will cause inconsistent behavior. Records read from DDIF files on systems earlier than Version 5.1 will be fixed-length 512-byte records, which contain DDIF control information in addition to the text context. Thus, typing a DDIF file on a system earlier than Version 5.1 does not produce readable text.

Copying a DDIF file using a system earlier than Version 5.1 will not preserve the DDIF tag on the output file, which will cause problems in later access to the new file from a Version 5.1 or later system.

However, using the Backup Utility from systems earlier than Version 5.1 will create a correct backup of DDIF files, and will properly restore DDIF files from BACKUP save sets.

C.1.3 VMS RMS Interface Changes

This section provides details about the changes made to the VMS RMS interface that support access to text in VMS DECwindows DDIF files. It includes information related to tagging files and accessing tagged files through the VMS RMS interface. The section also describes how tags are preserved at the VMS RMS interface.

C.1.3.1 Programming Interface for File Tagging

This section focuses on the use of the DDIF tag for supporting VMS DECwindows files, although VMS RMS also supports file tagging for other compound document data formats.

You can tag a file from the VMS RMS interface by using the $CREATE service in conjunction with a new extended attribute block (XAB) called the item XAB ($XABITM). The $XABITM macro is a general-purpose macro that was added to the RMS interface to support several Version 5.0 features. Tagged file support involves the use of the two item codes shown in Table C–1.

Table C–1 Tag Support Item Codes

Item	Buffer Size	Function
XAB$_STORED_SEMANTICS	64 bytes maximum	Defines the file semantics established when the file is created
XAB$_ACCESS_SEMANTICS	64 bytes maximum	Defines the file semantics desired by the accessing program

The entries XAB$_STORED_SEMANTICS and XAB$_ACCESS_SEMANTICS in the item list can represent either a control (set) function or a monitor (sense) function that can be passed to VMS RMS from the application program by way of the RMS interface.

The symbolic value XAB$K_SEMANTICS_MAX_LEN represents the tag length. This value can be used to allocate buffer space for sensing and setting stored semantics for the DDIF file.

Within any one $XABITM, you can activate either the set function or the sense function for the XAB$_STORED_SEMANTICS and XAB$_ACCESS_SEMANTICS items, because a common field (XAB$B_MODE) determines which function is active. If you want to activate both the set function and the sense function for either or both items, you must use two $XABITM control blocks, one for setting the functions and one for sensing the functions.

Each entry in the item list addressed by the $XABITM is made up of three longwords and a longword of zero terminates the list. You can locate the item list anywhere within the readable address space for a process, but any buffers required by the related function must be located in read/write memory. If the item list is invalid, RMS returns a status of RMS$_XAB in the RAB$L_STS field and the address of the XAB in RAB$L_STV.

The format and arguments of the $XABITM macro are as follows. Note that the block length field and the type code field are statically initialized by the $XABITM macro or may be explicitly initialized using a high-level language.

$XABITM

FORMAT

$XABITM *ITEMLIST=item-list-address,*
MODE= $\left\{ \begin{array}{l} sensemode \\ setmode \end{array} \right\}$,
NXT=next-xab-address

ARGUMENTS

The ITEMLIST argument defaults to zero, but a valid pointer must be specified when you use a XABITM. MODE defaults to *sensemode.* The symbolic offset, size, and a brief description of each XABITM field are described in the following list:

- The block length field (XAB$B_BLN) is a 1-byte static field that defines the length of the XABITM, in bytes. This field is initialized to the value XAB$C_ITMLEN.
- The type code (XAB$B_COD) field is a 1-byte static field that identifies this control block as a XABITM. This field is initialized to the value XAB$C_ITM.
- The XAB$L_ITEMLIST field is a longword field that contains the symbolic address of the item list.
- The XAB$B_MODE field is a 1-byte field that specifies whether the items can be set by the program. It contains either the symbolic value XAB$K_SETMODE or the symbolic value XAB$K_SENSEMODE (default).
- The XAB$L_NXT field is a longword field that contains the symbolic address of the next XAB in the XAB chain. A value of zero (the default) indicates that the current XAB is the last (or only) XAB in the chain.

Example C–1 illustrates a BLISS–32 program that tags a file through the RMS interface. The tag value shown is a 6-byte hexadecimal number representing the code for the DDIF tag. The VMS RMS program interface accepts only hexadecimal tag values.

To write to a tagged file without using an RMS extension, the application program must specify access semantics that match the file's stored semantics. As shown in the example, $CREATE tags the file and $CONNECT specifies the appropriate access semantics.

Example C–1 Tagging a File

```
MODULE TYPE$MAIN (
        IDENT = 'X-1',
        MAIN = MAIN,
        ADDRESSING_MODE (EXTERNAL=GENERAL)
        ) =
BEGIN
!
FORWARD ROUTINE
    MAIN : NOVALUE;                         ! Main routine
!
! INCLUDE FILES:
!
LIBRARY 'SYS$LIBRARY:LIB';
OWN
    NAM             : $NAM(),
    RETLEN,
    DDIF_TAG         : BLOCK[ 7, BYTE]
                INITIAL( BYTE(%X'2B',%X'0C',%X'87',%X'73',%X'01',%X'03',%X'01')),
    FAB_XABITM          :
                $xabitm
                  ( itemlist=
                        $ITMLST_UPLIT
                          (
                            (ITMCOD=XAB$_STORED_SEMANTICS,
                             BUFADR=DDIF_TAG,
                             BUFSIZ=%ALLOCATION(DDIF_TAG))
                          ),
                    mode = SETMODE),
    RAB_XABITM          :
                $xabitm
                  ( itemlist=
                        $ITMLST_UPLIT
                          (
                            (ITMCOD=XAB$_ACCESS_SEMANTICS,
                             BUFADR=DDIF_TAG,
                             BUFSIZ=%ALLOCATION(DDIF_TAG))
                          ),
                    mode = SETMODE),
    FAB         : $FAB( fnm = 'TAGGED-FILE.TEST',
                        nam = NAM,
                        mrs = 512,
                        rfm = FIX,
                        fac = <GET,PUT,UPD>,
                        xab = FAB_XABITM),
    REC              : BLOCK[512,BYTE],
    STATUS,
    RAB         : $RAB( xab = RAB_XABITM,
```

(continued on next page)

Example C–1 (Cont.) Tagging a File

```
                        fab = FAB,
                        rsz = 512,
                        rbf = REC,
                        usz = 512,
                        ubf = REC),
    DESC            : BLOCK[8,BYTE] INITIAL(0);
ROUTINE MAIN : NOVALUE =
BEGIN
STATUS = $CREATE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
STATUS = $CONNECT( RAB = RAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
STATUS = $CLOSE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
END;
END
ELUDOM
```

C.1.3.2 Accessing a Tagged File

This section provides details of how VMS RMS handles access to tagged files at the program level. When a program accesses a tagged file, VMS RMS must determine whether and when to associate an RMS extension with the access. This is important to the programmer because an RMS extension can change the attributes of the accessed file.

For example, a DDIF file is stored as a sequentially organized file having 512-byte, fixed-length records. If the DDIF-to-text RMS extension is used to extract text from a DDIF file, the accessed file appears as a sequentially organized file having variable-length records with a maximum record size of 2048 bytes and an implicit carriage return.

One consideration in determining whether an access requires the RMS extension is the type of access (FAB$B_FAC). When an application program opens a file through the VMS RMS program interface, it must specify if it will be doing record I/O (default), block I/O (BIO), or mixed I/O (BRO), where the program has the option of using either block I/O or record I/O for each access. For example, if block I/O operations are specified, VMS RMS does not associate the RMS extension with the file access.

Another consideration is whether the program senses the tag when it opens a file. If the program does not sense the tag when it opens a DDIF file for record access, VMS RMS associates the RMS extension during the $OPEN and returns the file attributes that have been modified by the extension.

The final consideration is the access semantics the program specifies and the file's stored semantics (tag). If the program specifies block I/O (FAB$V_BIO) operations, RMS does not associate the RMS extension and the $OPEN service returns the file's stored attributes to the accessing program regardless of whether the program senses tags.

C.1.3.2.1 File Accesses That Do Not Sense Tags

This section describes what happens when a program does not use the XABITM to sense a tag when it opens a file.

When a program opens a DDIF file for record operations and does not sense the tag, VMS RMS assumes that the program wants to access text in the file. In this case, VMS RMS associates the RMS extension, which provides file attributes that correspond to record-mode access.

When a program opens a DDIF file with the FAB$V_BRO option and does not sense the tag, any subsequent attempt to use block I/O fails. If the program specifies block I/O (FAB$V_BIO) when it invokes the $CONNECT service, the operation fails because the file attributes returned at $OPEN permit record access only. Similarly, if the program specifies the FAB$V_BRO option when it opens the file, and then specifies mixed mode (block /record) operations by not specifying RAB$V_BIO at $CONNECT time, block operations such as READ and WRITE are disallowed.

C.1.3.2.2 File Accesses That Sense Tags

VMS RMS does not associate the RMS extension as part of the $OPEN service if a program opens a DDIF file and senses the stored semantics. This allows the program to specify access semantics with the $CONNECT service. VMS RMS returns the file attributes, including the stored semantics attribute (tag value), to the program as part of the $OPEN service.

When the program subsequently invokes the $CONNECT service, VMS RMS uses the specified operations mode to determine its response. If the program specified FAB$V_BRO with the $OPEN service and then specifies block I/O (RAB$V_BIO) when it invokes the $CONNECT service, VMS RMS does not associate the RMS extension.

But if the program specifies record access or FAB$V_BRO when it opens the file and then decides to use record I/O when it invokes the $CONNECT service, VMS RMS compares the access semantics with the file's stored semantics to determine whether to associate the RMS extension. If the access semantics match the stored semantics, VMS RMS does not associate the RMS extension. If the access semantics do not match the stored semantics, VMS RMS associates the access with the RMS extension. In this case, the program must use the $DISPLAY service to obtain the modified file attributes. If VMS RMS cannot find the appropriate RMS extension, the operation fails and the $CONNECT service returns the EXTNOTFOU error message.

If the application program senses the file's stored semantics, VMS RMS allows mixed-mode operations. In this case, mixed block and record operations are permitted because the application gets record mode file attributes and data from the RMS extension and block mode file attributes and data from the file.

Example C–2 illustrates a BLISS–32 program that accesses a tagged file from an application program that does not use an RMS extension.

Example C–2 Accessing a Tagged File

```
MODULE TYPE$MAIN (
        IDENT = 'X-1',
        MAIN = MAIN,
        ADDRESSING_MODE (EXTERNAL=GENERAL)
        ) =
BEGIN
!
FORWARD ROUTINE
    MAIN : NOVALUE;                        ! Main routine
!
! INCLUDE FILES:
!
LIBRARY 'SYS$LIBRARY:STARLET';
OWN
    NAM            : $NAM(),
    ITEM_BUFF   : BLOCK[ XAB$K_SEMANTICS_MAX_LEN,BYTE ],
    RETLEN,
    FAB_XABITM        :
                $xabitm
                  ( itemlist=
                        $ITMLST_UPLIT
                          ((ITMCOD=XAB$_STORED_SEMANTICS,
                             BUFADR=ITEM_BUFF,
                             BUFSIZ=XAB$K_SEMANTICS_MAX_LEN,
                             RETLEN=RETLEN)),
                    mode = SENSEMODE),
    RAB_ITEMLIST  : BLOCK[ ITM$S_ITEM + 4, BYTE ],
    RAB_XABITM    : $XABITM
                  ( itemlist=RAB_ITEMLIST,
                    mode=SETMODE ),
    FAB         : $FAB( fnm = 'TAGGED-FILE.TEST',
                        nam = NAM,
                        fac = <GET,PUT,UPD>,
                        xab = FAB_XABITM),
    REC             : BLOCK[512,BYTE],
    STATUS,
    RAB         : $RAB( xab = RAB_XABITM,
                        fab = FAB,
                        rsz = 512,
                        rbf = REC,
                        usz = 512,
                        ubf = REC),
    DESC            : BLOCK[8,BYTE] INITIAL(0);
ROUTINE MAIN : NOVALUE =
BEGIN
STATUS = $OPEN( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
RAB_ITEMLIST[ ITM$W_BUFSIZ ] = .RETLEN;
RAB_ITEMLIST[ ITM$L_BUFADR ] = ITEM_BUFF;
RAB_ITEMLIST[ ITM$W_ITMCOD ] = XAB$_ACCESS_SEMANTICS;
STATUS = $CONNECT( RAB = RAB );
IF NOT .STATUS
THEN
```

(continued on next page)

Example C–2 (Cont.) Accessing a Tagged File

```
    SIGNAL (.STATUS);
STATUS = $CLOSE( FAB = FAB );
IF NOT .STATUS
THEN
    SIGNAL (.STATUS);
END;
END
ELUDOM
```

C.1.3.3 Preserving Tags

To preserve the integrity of a tagged file that is being copied or transmitted, the tag must be preserved in the destination (output) file. The most efficient way to use the RMS interface for propagating tags is to open the source file (input) and sense the tag using a $XABITM with the item code XAB$_STORED_SEMANTICS:

```
    .
    .
    .
ITEMLIST[ ITM$W_BUFSIZ ] = XAB$K_SEMANTICS_MAX_LEN;
ITEMLIST[ ITM$L_BUFADR ] = ITEM_BUFF;
ITEMLIST[ ITM$L_RETLEN ] = RETLEN;
ITEMLIST[ ITM$W_ITMCOD ] = XAB$_STORED_SEMANTICS;
    .
    .
    .
XABITM[ XAB$B_MODE ] = XAB$K_SENSEMODE;
STATUS = $OPEN( FAB = FAB );
    .
    .
    .
```

Then create the destination (output) file and set the tag using a $XABITM with the item code XAB$_STORED_SEMANTICS:

```
    .
    .
    .
IF .RETLEN GTR 0
THEN
    BEGIN
    ITEMLIST[ ITM$W_ITMCOD ] = XAB$_STORED_SEMANTICS;
    ITEMLIST[ ITM$L_SIZE   ] = .RETLEN;
    XABITM[ XAB$B_MODE ] = XAB$K_SETMODE;
    END;

STATUS = $CREATE( FAB = FAB );
    .
    .
    .
END;
END
ELUDOM
```

C.1.4 Distributed File System Support for DDIF Tagged Files

Version 1.1 of the Distributed File System (DFS) includes limited support for DDIF tagged files. You can create and read DDIF files on a DFS device when the DFS client node is running VMS Version 5.1 or later versions. You can also use the DIRECTORY/FULL command to determine whether a DDIF file on a DFS device is tagged.

You cannot use the SET FILE/[NO]SEMANTICS command to either tag DDIF files or remove the tags from DDIF files on a DFS device. Furthermore, the Backup Utility does not preserve the DDIF tag or the DDIF stored semantics for data files on a DFS device.

C.1.5 VMS RMS Errors

Four VMS RMS error messages signal the user when the corresponding error condition exists:

- RMS$_EXTNOTFOU
- RMS$_SEMANTICS
- RMS$_EXT_ERR
- RMS$_OPNOTSUP

The RMS$_EXTNOTFOU error message indicates that VMS RMS has not found the specified RMS extension. Verify that the file is correctly tagged, using the DIRECTORY/FULL command, and that the application program is specifying the appropriate access semantics.

VMS RMS returns the RMS$_SEMANTICS error message when you try to create a tagged file on a remote system earlier than VMS Version 5.1 from a Version 5.1 or later system.

VMS RMS returns the RMS$_EXT_ERR error when the DDIF RMS extension detects an inconsistency.

VMS RMS returns the RMS$_OPNOTSUP error when the RMS DDIF extension is invoked by an RMS operation. For example, if the extension does not support write access to a DDIF file, verify that the application program is not performing record operations that modify the file.

C.2 EXCHANGE/NETWORK Command

The EXCHANGE/NETWORK command allows the VMS operating system to transfer files to or from operating systems that do not support VMS file organizations. The transfer occurs over a DECnet network communications link that connects VMS and non-VMS operating system nodes.

Using DECnet services, the EXCHANGE/NETWORK command can perform the following operations:

- Transfer files between a VMS node and a non-VMS system node
- Transfer a group of input files to a group of output files
- Transfer files between two non-VMS nodes, provided those nodes share DECnet connections with the VMS node that issues the EXCHANGE /NETWORK command

The EXCHANGE/NETWORK command imposes the following restrictions:

- Transfers of files can occur only between disk devices. (If a disk device is not the desired permanent residence for the file, you must either move the file to a disk before issuing the command or retrieve the file from a disk after the command completes.)
- The remote system must have a block size of 512 bytes, where a byte is 8 bits long.
- The nodes transferring files must support the DECnet Data Access Protocol (DAP).

The VMS Record Management Services (RMS) facility provides VMS access to records in VMS RMS files. To transfer VMS RMS files between two nodes where both nodes are VMS nodes, use one of the other DCL commands (such as COPY, APPEND, or CONVERT), as appropriate. These commands recognize RMS file organizations and are designed to ensure that RMS record structures are preserved as your files are moved.

Use the EXCHANGE/NETWORK command to transfer files between VMS nodes and non-VMS nodes when the differences in the file organizations would otherwise prevent the transfer or could lead to undesirable results. While COPY ensures that both the contents and the attributes of a replicated file are preserved, EXCHANGE/NETWORK is more flexible. EXCHANGE /NETWORK offers you explicit control of your record attributes during file transfers, with the opportunity to make a file usable on several different operating systems.

FORMAT

EXCHANGE/NETWORK *input-file-spec[,...] output-file-spec*

PARAMETERS

input-file-spec[,...]
Specifies the name of an existing file to be transferred. Wildcard characters are allowed. Use a comma (,) to indicate multiple file specifications.

output-file-spec
Specifies the name of the output file into which the input is transferred.

You must specify at least one field in the output file specification. If you omit the device or directory, your current default device and directory are used. The EXCHANGE/NETWORK command replaces any other missing fields (file name, file type, version number) with the corresponding field of the input file specification.

EXCHANGE/NETWORK creates a new output file for every input file that you specify.

You can use the asterisk wildcard character in place of the following: file name, file type, or version number. The EXCHANGE/NETWORK command uses the corresponding field in the related input file to name the output file. You can also use the wildcard character in the output file specification to direct EXCHANGE/NETWORK to create more than one output file. For example:

```
$ EXCHANGE/NETWORK A.A,B.B  MYPC::*.C
```

This EXCHANGE/NETWORK command creates the files A.C and B.C at the non-VMS target node MYPC.

A more complete explanation of wildcard characters and version numbers follows in the "description" section.

DESCRIPTION

The EXCHANGE/NETWORK command transfers files between VMS nodes and non-VMS nodes connected to the same DECnet network. If the non-VMS system does not support VMS file organizations, EXCHANGE /NETWORK can modify or discard file and record attributes during the transfer. However, if the target system is a VMS node, you have the option of applying new file and record attributes to the output file by supplying a File Definition Language (FDL) file, as described later in this section. EXCHANGE/NETWORK provides a number of defaults to handle the majority of transfers properly. However, in some situations, you need to know your file or record format requirements at both nodes.

VMS File and Record Attributes

All RMS files in the VMS environment include stored information, known as the file and record attributes, to describe the file and record characteristics. File attributes consist of items such as file organization, file protection, and file allocation information. Record attributes consist of items such as the record format, record size, key definitions for indexed files, and carriage control information. These attributes define the data format and access methods for the VMS RMS facility.

Non-VMS operating systems that do not support VMS file organizations have no means of storing file and record attributes with their files. Transferring a VMS file to a non-VMS system that is unable to store and handle file and record attributes can result in most of this information being discarded. Removing these attributes from a file can render it useless if it must be returned to the VMS system.

Transferring Files to VMS Nodes

When you transfer files to a VMS system from a non-VMS system, the files typically assume default file and record attributes. However, you can specify the attributes that you want the file to acquire in a File Definition Language (FDL) file. If you specify an FDL file with the /FDL qualifier, the FDL file determines the characteristics of the output file. This feature is useful in establishing compatible file and record attributes when you transfer a file from a non-VMS system to a VMS system. However, when you use an FDL file, you also assume responsibility for determining the required characteristics.

See the *VMS File Definition Language Facility Manual* for more information about FDL files.

Transferring Files to Non-VMS Nodes

EXCHANGE/NETWORK discards file and record attributes associated with a VMS file during a transfer to a non-VMS system that does not support VMS file organizations. Be aware that the loss of file and record attributes in the transfer can render the output file useless for many applications.

Selecting Transfer Modes

The EXCHANGE/NETWORK command has four transfer mode options: AUTOMATIC, BLOCK, RECORD, and CONVERT. For most file transfers, AUTOMATIC is sufficient. The AUTOMATIC transfer mode option allows EXCHANGE/NETWORK to transfer files using either block or record I/O. The selection is based on the input file organization and the operating systems involved.

Selecting the BLOCK transfer mode option forces EXCHANGE/NETWORK to open both the input and output files for block I/O access. The input file is then transferred to the output file block by block. Use this transfer mode when you transfer executable images. It is also useful when you must preserve a file's content exactly, which is a common requirement when you store files temporarily on another system or when cooperating applications exist on the systems.

Selecting the RECORD transfer mode option forces EXCHANGE /NETWORK to open both the input file and output file for record I/O access. The input file is then transferred to the output file record by record. This transfer mode is primarily used for transferring text files.

Selecting the CONVERT transfer mode option forces EXCHANGE /NETWORK to open the input file for RECORD access and the output file for BLOCK access. Records are then read in from the input file, packed into blocks, and written to the output file. This transfer mode is primarily used for transferring files with no implied carriage control. For example, to transfer a file created with DIGITAL Standard Runoff (DSR) to a DECNET-DOS system, you must use the CONVERT transfer mode option. To transfer the resultant output file back to a VMS node, use the AUTOMATIC transfer mode option.

Wildcard Characters

Wildcard characters are permitted in the file specifications and follow the behavior typical of other VMS commands with respect to the VMS node.

When more than one input file is specified but wildcards are not specified in the output file specification, the first input file is copied to the output file, and each subsequent input file is transferred and given a higher version number of the same output file name. Note that the files are not concatenated into a single output file. Also note that when you transfer files to foreign systems that do not support version numbers, only one output file results, and it is the last input file.

To create multiple output files, specify multiple input files and use at least one of the following:

- An asterisk wildcard character in the output file name, file type, or version number field
- Only a node name, a device name, or a directory specification as the output file specification

When you create multiple output files, EXCHANGE/NETWORK uses the corresponding field from each input file in the output file name.

Use the /LOG qualifier when you specify multiple input and output files to verify that the files were copied as you intended.

Version Numbers

The following guidelines apply when the target node file formats accept version numbers.

If no version numbers are specified for input and output files, the EXCHANGE/NETWORK command (by default) assigns a version number to the output files that is either of the following:

- The version number of the input file
- A version number one greater than the highest version number of an existing file with the same file name and file type

When the output file version number is specified by an asterisk wildcard character, the EXCHANGE/NETWORK command uses the version numbers of the associated input files as the version numbers of the output files.

If the output file specification has an explicit version number, the EXCHANGE/NETWORK command normally uses that number for the output file specification. However, if an equal or higher version of the output file already exists, no warning message is issued, the file is copied, and the version number is set to a value one greater than the highest version number already existing.

File Protection and Creation/Revision Dates

The EXCHANGE/NETWORK command treats an output file as a new file when any portion of the output file name is explicitly specified. When the output node is a VMS system, the creation date for a new file is set to the current time and date. However, if the output file specification consists *only* of wildcard characters, the output file no longer qualifies as a new file, and, therefore, the creation date of the input file is used. That is, if the output file specification is one of the following, the creation date becomes that of the input file: *, *.*, or *.*;*.

The revision date of the output file is always set to the current time and date; the backup date is set to zero. The output file is assigned a new expiration date. (Expiration dates are set by the file system if retention is enabled; otherwise, they are set to zero.)

When the target node is a VMS node, the protection and access control list (ACL) of the output file is determined by the following parameters, in the following order:

1 Protection of previously existing versions of the output file
2 Default protection and ACL of the output directory
3 Process default file protection

For an introduction to access control lists, see the *VMS DCL Concepts Manual.*

On VMS systems, the owner of the output file usually is the same as the creator of the output file. However, if a user with extended privileges creates the output file, the owner is either the owner of the parent directory or the owner of a previous version of the output file, if one exists.

Extended privileges include any of the following:

- SYSPRV or BYPASS
- System UIC
- GRPPRV if the owner of the parent directory (or previous version of the output file) is in the same group as the creator of the new output file
- An identifier (with the resource attribute) representing the owner of the parent directory (or previous version of the output file)

QUALIFIERS

/BACKUP

Modifies the time value specified with the /BEFORE or /SINCE qualifier. /BACKUP selects files according to the dates of their most recent backup. This time qualifier is incompatible with the other time qualifiers that also allow you to select files according to time attributes: /CREATED, /EXPIRED, and /MODIFIED. If you do not specify any of these four time qualifiers, the default is /CREATED.

/BEFORE[=time]

Selects only those files dated prior to the specified time. You can specify time as an absolute time, as a combination of absolute and delta times, or as one of the following keywords: TODAY (default), TOMORROW, or YESTERDAY. Specify one of the following time qualifiers with /BEFORE to indicate the time attribute to be used as the basis for selection: /BACKUP, /CREATED (default), /EXPIRED, or /MODIFIED.

See the *VMS DCL Concepts Manual* for complete information about specifying time values.

/BY_OWNER[=uic]

Selects only those files whose owner user identification code (UIC) matches the specified owner UIC. The default UIC is that of the current process.

Specify the UIC using standard UIC format as described in the *VMS DCL Concepts Manual.*

/CONFIRM
/NOCONFIRM (default)

Controls whether a request is issued before each file transfer operation to confirm that the operation should be performed on that file. The following responses are valid:

YES	NO	QUIT
TRUE	FALSE	Ctrl/Z
1	0	ALL
	Return	

You can use any combination of uppercase and lowercase letters for word responses. Word responses can be abbreviated to one or more letters (for example, T, TR, or TRU for TRUE), but these abbreviations must be unique. Affirmative answers are YES, TRUE, and 1. Negative answers are NO, FALSE, 0, and the Return key. QUIT or Ctrl/Z indicates that you want to stop processing the command at that point. When you respond with ALL, the command continues to process, but no further prompts are given. If you type a response other than one of those in the list, DCL displays an error message and redisplays the prompt.

/CREATED (default)

Modifies the time value specified with the /BEFORE or /SINCE qualifier. The /CREATED qualifier selects files based on their date of creation. This time qualifier is incompatible with the other time qualifiers that also allow you to select files according to time attributes: /BACKUP, /EXPIRED, and /MODIFIED. If you do not specify any of these four time qualifiers, the default is /CREATED.

/EXCLUDE=(file-spec[,...])
Excludes the specified files from the file transfer operation. You can include a directory but not a device in the file specification. Wildcard characters are allowed in the file specification. However, you cannot use relative version numbers to exclude a specific version. If you provide only one file specification, you can omit the parentheses.

/EXPIRED
Modifies the time value specified with the /BEFORE or /SINCE qualifiers. /EXPIRED selects files according to their expiration date. (The expiration date is set with the SET FILE/EXPIRATION_DATE command.) This time qualifier is incompatible with the other time qualifiers that also allow you to select files according to time attributes: /BACKUP, /CREATED, and /MODIFIED. If you do not specify any of these four time qualifiers, the default is /CREATED.

/FDL=fdl-file-spec
Specifies that the output file characteristics are described in the File Definition Language (FDL) file. Use this qualifier when you require special output file characteristics. See the *VMS File Definition Language Facility Manual* for more information about FDL files.

Use of the /FDL qualifier implies that the transfer mode is block by block. However, the transfer mode you specify with the /TRANSFER_MODE qualifier prevails.

/LOG
/NOLOG (default)
Controls whether the EXCHANGE/NETWORK command displays the file specifications of each file copied.

When you use the /LOG qualifier, the EXCHANGE/NETWORK command displays the following for each copy operation: (1) the file specifications of the input and output files, and (2) the number of blocks or the number of records copied (depending on whether the file is copied on a block-by-block or record-by-record basis).

/MODIFIED
Modifies the time value specified with the /BEFORE or /SINCE qualifier. The /MODIFIED qualifier selects files according to the date on which they were last modified. This time qualifier is incompatible with the other time qualifiers that also allow you to select files according to time attributes: /BACKUP, /CREATED, and /EXPIRED. If you do not specify any of these four time qualifiers, the default is /CREATED.

/SINCE[=time]
Selects only those files dated after the specified time. You can specify time as an absolute time, a combination of absolute and delta times, or as one of the following keywords: TODAY (default), TOMORROW, or YESTERDAY. Specify one of the following time qualifiers with /SINCE to indicate the time attribute to be used as the basis for selection: /BACKUP, /CREATED (default), /EXPIRED, or /MODIFIED.

See the *VMS DCL Concepts Manual* for complete information about specifying time values.

/TRANSFER_MODE=option

Specifies the I/O method to be used in the transfer. This qualifier is useful for all file formats. You can specify any one of the following options:

Option	Function
AUTOMATIC	Allows EXCHANGE/NETWORK to determine the appropriate transfer mode.
BLOCK	Transfers block by block.
CONVERT[=option[,...]]	Reads records from the input file, packs them into blocks, and writes to the output file in block mode. The options determine what additional information is inserted during the transfer.
RECORD	Transfers record by record.

The AUTOMATIC transfer mode option allows EXCHANGE/NETWORK to determine the appropriate transfer mode. The default is the AUTOMATIC transfer mode.

If you explicitly select the BLOCK transfer mode option, EXCHANGE /NETWORK opens both the input and output files for block I/O. EXCHANGE /NETWORK then transfers the files block by block.

If you explicitly select the RECORD transfer mode option, EXCHANGE /NETWORK opens both the input and output files for record I/O. The target system must support record operations, and the input file must be record oriented.

If you select the CONVERT transfer mode option, EXCHANGE /NETWORK reads records in from the input file, packs them into blocks, and writes them to the output file in block mode. There are four options available with the CONVERT transfer mode to control the insertion of special characters in the records, as explained in the following paragraphs:

- CARRIAGE_CONTROL
- COUNTED
- FIXED_CONTROL
- RECORD_SEPARATOR=separator

If you specify CARRIAGE_CONTROL, any carriage control information in the input file is interpreted, expanded into actual characters, and included with each record.

If you specify COUNTED, the length of each record in bytes is included at the beginning of each record. The length includes all FIXED_CONTROL, CARRIAGE_CONTROL, and RECORD_SEPARATOR information in each record.

If you specify FIXED_CONTROL, all variable length with fixed control record (VFC) information is written to the output file as part of the data. This information follows the record length information if the COUNTED option was specified.

If you specify RECORD_SEPARATOR, a 1- or 2-byte record separator is inserted between each record. Record separator characters are the last characters in the record. The three choices for separator characters are CR for carriage return only, LF for line feed only, or CRLF for carriage return and line feed.

EXAMPLES

1
```
$ EXCHANGE/NETWORK VMS_FILE.DAT FOO::FOREIGN_SYS.DAT
```

In this example, the EXCHANGE/NETWORK command transfers the file VMS_FILE.DAT located in the current default device and directory to the file FOREIGN_SYS.DAT on the non-VMS node FOO. Because the /TRANSFER_MODE qualifier was not explicitly specified, EXCHANGE /NETWORK automatically determines whether the transfer method should be block or record I/O.

2
```
$ EXCHANGE/NETWORK/TRANSFER_MODE=BLOCK -
_$  FOO::FOREIGN_SYS.DAT VMS_FILE.DAT
```

In this example, the EXCHANGE/NETWORK command transfers the file FOREIGN_SYS.DAT from the non-VMS node FOO to the file VMS_FILE.DAT in the current default device and directory. Block I/O is specified for the transfer mode.

3
```
$ EXCHANGE/NETWORK/FDL=VMS_FILE_DEFINITION.FDL -
_$  FOO::REMOTE_FILE.TXT  VMS_FILE.DAT
```

In this example, the EXCHANGE/NETWORK command transfers the file REMOTE_FILE.TXT on node FOO to the file VMS_FILE.DAT. The file attributes for the output file VMS_FILE.DAT are obtained from the File Definition Language (FDL) source file VMS_FILE_DEFINITION.FDL. For more information about creating FDL files, see the *VMS File Definition Language Facility Manual.* Because the qualifier /FDL is specified and the /TRANSFER_MODE qualifier is omitted, the transfer mode uses block I/O, by default.

4
```
$ EXCHANGE/NETWORK -
_$  /TRANSFER_MODE=CONVERT=(CARRIAGE_CONTROL,COUNTED, -
_$  RECORD_SEPARATOR=CRLF,FIXED_CONTROL) -
_$  PRINT_FILE.TXT  FOO::*
```

In this example, the EXCHANGE/NETWORK command transfers the file PRINT_FILE.TXT from the current default device and directory to the file PRINT_FILE.TXT on the non-VMS node FOO. The use of the CONVERT option with the /TRANSFER_MODE qualifier forces the input file to be read in record by record, modified as specified by the convert options described below, and written to the output file block by block. As many records as will fit are packed into the output blocks.

The CONVERT option CARRIAGE_CONTROL specifies that carriage control information be converted to ASCII characters and inserted before the data or appended to the record, depending on whether prefix control or postfix control, or both, are used. The CONVERT option FIXED_CONTROL specifies that any fixed control information be translated to ASCII characters and inserted at the beginning of the record. The CONVERT option RECORD_SEPARATOR=CRLF appends the two specified characters, carriage return and line feed, to the end of the record. The CONVERT option COUNTED specifies that the total length of the record must be counted (once the impact of all the previous convert options have been added), and the result is to be inserted at the beginning of the record, in the first two bytes.

Index

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

VMS Version 5.4 New Features Manual
AA–LA97C–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 ADRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications — Spit Brook
ZKO1-3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line