94-003|103|10

VAXELN Runtime Facilities Guide

Order Number: AA–JM81E–TE

This manual is a guide to using the VAXELN runtime facilities.

Revision/Update Information:	This manual supersedes the *VAXELN Runtime Facilities Guide*, AA–JM81D–TE.
Operating System and Version:	VMS, Version 4.7 or higher
Software Version:	VAXELN, Version 4.1

digital equipment corporation
maynard, massachusetts

Revised, March 1990

Contents

FIGURES

Preface

The *VAXELN Runtime Facilities Guide* describes the VAXELN runtime software and explains how to use it to produce dedicated, realtime VAXELN systems.

The manual provides a language-independent discussion of the VAXELN Toolkit's runtime facilities. It explains VAXELN programming concepts and describes runtime features that you program and build into VAXELN systems. For information about developing and monitoring VAXELN systems, see the *VAXELN Development Utilities Guide*.

Intended Audience

This manual is for programmers and students who have a working knowledge of Pascal, C, or FORTRAN. Knowledge of the VMS operating system and a cursory understanding of the Digital command language (DCL) is recommended. Some information in this manual requires a more extensive understanding of the VMS operating system. In such cases, this manual directs you to appropriate documentation for additional information.

Document Structure

This manual consists of 14 chapters and 4 appendixes, organized as follows:

- Chapter 1, Runtime Facilities Overview, provides general information about the runtime facilities and their role in an executing application.
- Chapter 2, The VAXELN Kernel, introduces the VAXELN Kernel and describes the kernel data structures.
- Chapter 3, Job, Process, and Memory Management, explains how VAXELN application programs can manage processes, jobs, and memory. The kernel's roles in scheduling and memory allocation are discussed in this chapter.
- Chapter 4, Synchronization, explains how to use kernel objects, optimized structures, and related procedures to synchronize processes.
- Chapter 5, Communication, explains how to use kernel objects and related procedures to program interprocess and interjob communication.
- Chapter 6, Device Handling, explains how to use the kernel DEVICE object, related kernel procedures, and interrupt service routines in programs that handle device interrupts. This chapter also discusses recovery from power failure and direct memory access UNIBUS and Q-bus device handling.
- Chapter 7, Exception Handling, explains how to handle exceptions from your VAXELN programs. This chapter also discusses status codes and message-processing features that handle the conversion of status codes into message text.
- Chapter 8, Ethernet/IEEE 802 Datalink Drivers, describes the Ethernet/IEEE 802 datalink drivers and Datagram Service and explains how to use the Datagram Service.
- Chapter 9, DECnet Network Services, describes the VAXELN Network Service.

- Chapter 10, Internet Services, explains how to use the VAXELN Internet Services.
- Chapter 11, LAT Host Services, explains how to establish virtual circuits for local area transport communication, manage VAXELN service nodes, and set up dedicated service and application device environments.
- Chapter 12, System Security, explains how to include security features in your VAXELN systems for protecting resources and data.
- Chapter 13, File Service, describes the VAXELN File Service and explains how to use file, disk, and tape utility procedures in your application programs.
- Chapter 14, VAXELN Device Drivers, describes the disk, virtual memory, tape, printer, terminal, and realtime device drivers that VAXELN supplies.
- Appendix A, Status Values/Exception Names, lists the status values/exception names that VAXELN defines.
- Appendix B, Machine-Check Stack Frames, explains how to manually obtain and interpret a machine-check stack frame, in case a machine check occurs on a VAXELN target processor in an application that does not include the error-logging service.
- Appendix C, VMS Emulation Routines, identifies the VMS runtime library and system service emulation routines that the VAXELN Toolkit supports.
- Appendix D, SCSI Port Driver Interface Routines, describes the VAXELN SCSI port driver interface routines that you can use to program user-written SCSI class drivers for third-party SCSI devices.

Conventions

The following conventions are used in this manual:

Convention	Meaning
UPPERCASE characters	VMS, VAXELN, and language-specific reserved words and identifiers are printed in uppercase characters.

Convention	Meaning
italic characters	The following items are printed in italic characters: • Elements for which you supply a value. For example: *nodename*::"TASK=*portname*" • User-defined elements in code examples when these elements are used in text. For example: The *get_attributes* argument . . . • System Builder menu entry values when they appear in text. For example: Select *Yes* for the **Console** entry on the System Characteristics Menu. • First occurrence of a new term.
bold characters	The following items are printed in bold characters: • System Builder menu entries when they appear in text. For example: Select *Yes* for the **Console** entry on the System Characteristics Menu. • Case-sensitive C language elements, such as keywords, macros, modules, and procedures, when they appear in text. For example: The definition module **$vaxelnc** . . .
red characters	In interactive examples, elements for which you must supply input. For example: $ SHOW NETWORK
[ ]	Square brackets enclose optional items. For example: SHOW NODE *node-id* [SUMMARY] [COUNTERS] Square brackets are also used in the syntax of a directory name in a VMS file specification and in user identification code (UIC) specifications.
. . .	When an item is followed by horizontal ellipsis points, you can repeat the item one or more times.
. . .	Vertical ellipsis points in a figure or example indicate that not all the information the system displays is shown or that not all the information a user is to supply is shown.

Convention	Meaning
Ctrl/*x*	Ctrl/*x* indicates a control key sequence. Press the key labeled Ctrl while you simultaneously press another key. For example: Ctrl/C
n and *x*	When used in items such as names, the variables *n* and *x* represent numeric and nonnumeric characters, respectively. For example: VAX 6000–2*nn* series systems

Associated Documents

The following documents are relevant to programming VAXELN applications using the VAXELN runtime facilities:

VAXELN Documents:

- *VAXELN Release Notes*
- *VAXELN Installation Guide*
- *Introduction to VAXELN*
- *VAXELN Development Utilities Guide*
- *VAXELN Runtime Facilities Guide*
- *VAXELN Application Design Guide*
- *VAXELN Pascal Language Reference Manual*
- *VAXELN Pascal Runtime Library Reference Manual*
- *VAXELN C Reference Manual*
- *VAXELN C Runtime Library Reference Manual*
- *VAXELN FORTRAN Runtime Library Reference Manual*
- *VAXELN Pocket Reference*
- *VAXELN Messages Manual*
- *VAXELN Guide to DECwindows*
- *VAXELN Master Index and Glossary*

VAX Documents:

- *VAX Architecture Reference Manual*
- *VAX Hardware Handbook*
- *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer*

VMS Documents:

- *Guide to Creating VMS Modular Procedures*
- *Guide to Maintaining a VMS System*
- *Introduction to VMS*
- *Introduction to the VMS Run-Time Library*
- *Introduction to VMS System Management*
- *Introduction to VMS System Services*
- *VMS Authorize Utility Manual*
- *VMS DCL Dictionary*
- *VMS Error Log Utility Manual*
- *VMS I/O User's Reference Volume*
- *VMS Librarian Utility Manual*
- *VMS License Management Utility Reference Manual*
- *VMS Linker Utility Manual*
- *VMS Message Utility Manual*
- *VMS Network Control Program Reference Manual*
- *VMS Networking Manual*
- *VMS RTL Library (LIB$) Manual*
- *VMS RTL String Manipulation (STR$) Manual*
- *VMS Run-Time Library Routines Volume*
- *VMS System Services Reference Manual*

DECnet Documents:

- *DECnet DIGITAL Network Architecture General Description*
- *DECnet–VAX System Manager's Guide*
- *DECnet–VAX User's Guide*
- *Guide to DECnet–VAX Networking*

Hardware Documents:

- *ADQ32 A/D Converter Module User's Guide*
- *DLV11–J User's Guide*
- *DRB32 Hardware Manual*
- *DRB32 Technical Manual*
- *DRQ3B Parallel DMA I/O Module User's Guide*
- *DR11–W Direct Memory Access Interface User's Guide*
- *IEU11–A/IEQ11–A User's Guide*
- *IEX11–A IEC/IEEE Bus Interface*
- *KFQSA Installation Guide*
- *MicroVAX I Owner's Manual*
- *MicroVAX II Owner's Manual*
- *LSI–11 Analog System User's Guide*
- *Q-bus DMA Analog System User's Guide*
- *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)*
- *Small Computer System Interface: An Overview*
- *Small Computer System Interface: A Developer's Guide*
- *VAX 8800 Console Manual*
- *VAX 8nnn Console Manual*

VAX RTA Documents:

- *VAX Real-Time Accelerator Hardware/Software Installation Guide*
- *VAX Real-Time Accelerator Software User's Guide*

The *VAXELN Internals and Data Structures* manual is also available as a separate document. This manual describes the internal data structures and operations of the VAXELN Kernel and its associated subsystems.

Chapter 1

Runtime Facilities Overview

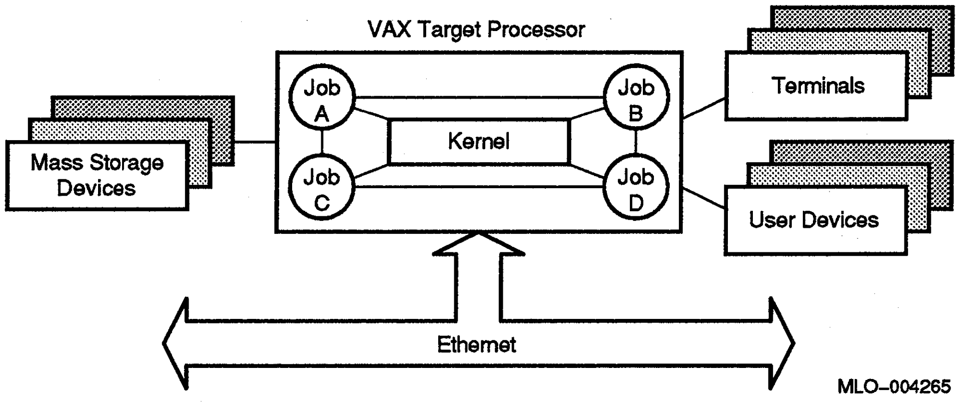
The VAXELN Toolkit is a VMS layered product that provides software for developing dedicated, realtime software applications that run on VAX processors. A *dedicated application* uses a computer to solve a specific problem or set of related problems. A typical dedicated application takes advantage of VAXELN realtime capabilities, which give prompt, predictable responses to time-critical events. The VAXELN Toolkit's low-overhead design caters to these application needs by applying the VAX processor's speed and responsiveness. Typical examples of dedicated, realtime applications include the following:

- Computer integrated manufacturing
- Process control
- Simulations
- Data acquisition and analysis
- File and print servers
- Communication switching systems
- Multifaceted professional workstations

VAXELN systems are only as complex as they need to be; they are statically defined and include only those services necessary to support the functions required by your application.

You develop a VAXELN application on a VAX *host processor* using VMS software and VAXELN development tools. The resulting *VAXELN system* includes user and Digital program images that reside in the memory of and run independently on a supported VAX *target processor*. Figure 1–1 shows a typical VAXELN application.

Figure 1–1: A VAXELN Application



This chapter provides information about the following:

- The VAXELN runtime environment, Section 1.1
- Basic VAXELN programming concepts, Section 1.2
- Facilities provided by VAXELN runtime software, Section 1.3

For information about the VAXELN development and utility features, see the *VAXELN Development Utilities Guide*.

1.1 VAXELN Runtime Environment

A *VAXELN runtime environment* consists of one or more VAX target processors running a VAXELN system image. The system image executes on the target processor as a dedicated application under the control of the kernel (see Section 1.3.1) and supplied services.

The runtime hardware requirements include the following:

- At least one of the target processors that the VAXELN Toolkit supports. For a list of supported target processors, see the help text for the System Builder's Target Processor Menu, or the latest VAXELN Toolkit *System Support Addendum* (SSA) or *Software Product Description* (SPD).

- Ethernet hardware if an application requires down-line loading, remote debugging, remote error logging, VAXELN Performance Utility, or remote VAXELN Command Language Utility support.
- Application-specific peripheral devices, such as disks, terminals, communications hardware, and special interfaces that Digital, a third party, or the programmer supplies.

The target processor can exist as a standalone system or can be distributed on a local area network.

A VAXELN system image includes user application program images and program images that Digital supplies. Typical user application programs, which you write in high-level languages, include data acquisition and reduction programs, process control supervisors, and user-written device drivers. The program images that Digital supplies include the VAXELN Toolkit's highly optimized kernel executives and images of the following:

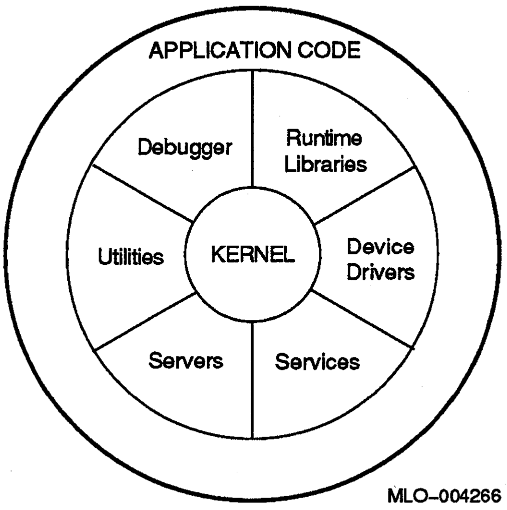
- Runtime libraries
- Device drivers
- Services
- A server
- Runtime utilities
- A local debugger component

Using the toolkit's System Builder, you combine the application program images and the program images that Digital supplies into a VAXELN system. When building the system, you can specify the programs that are to start executing as soon as you load and boot the system.

Figure 1–2 presents the system image components in a hierarchical diagram.

The diagram shows that the kernel executive is the heart of a VAXELN system; it schedules and controls an application's execution and access to system resources. The second tier of the diagram represents optional user and Digital software that provides kernel extensions. You tailor your VAXELN system by including only those services and utilities that

Figure 1–2: VAXELN System Software

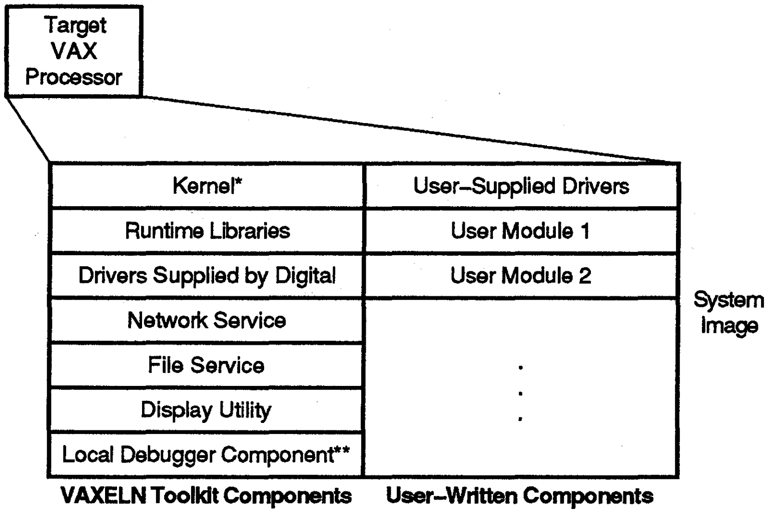


your application requires. The outermost tier represents a VAXELN system's highest level of code: your application program images.

After building your VAXELN system image, you can load and boot it onto the target processor from disk, tape, or read-only memory (ROM). If you have an optional DECnet–VAX license and the appropriate Ethernet hardware, you can down-line load the system image from the host processor to the target processor. The system image executes on the target processor independently under the control of the VAXELN Kernel and runtime services.

Figure 1–3 shows a typical VAXELN runtime environment, and Table 1–1 briefly describes the system components that Digital supplies.

Figure 1–3: Runtime Environment



* Required component
**Usually not included in final system

MLO–004267

Table 1–1: Runtime System Components

Component	Description
Kernel[1]	Controls the sharing of the target processor's resources. The System Builder includes the appropriate kernel for your target processor.
Runtime Libraries	Contain object modules and shareable images that support realtime, I/O, math, DECwindows, and other routines called from VAXELN Pascal, VAX C, and VAX FORTRAN programs.
Drivers	Control communication between application programs and external devices.
Network Service	Controls data transmission between network nodes, manages a network name table, and provides a runtime interface for managing a DECnet network.
Ethernet/IEEE 802 Datagram Service	Provides network interface routines that VAXELN application programs can use to communicate over a Carrier Sense Multiple Access/Collision Detect (CSMA/CD) LAN.
Internet Services	Provide an Ethernet network interface that VAXELN applications can use to communicate with other applications in an Internet network.
File Service	Provides support for file-oriented disk and tape I/O operations and remote file access.
Error Logging Service	Writes data that identifies hardware errors, volume changes, and system events to an error log file that exists on the local target system or on a remote system over the Ethernet.
LAT Host Services	Provide an interface that application programs can use to communicate with devices attached to terminal servers.
DECwindows Server	Provides a common means for DECwindows applications to interact with graphics workstations.

[1]A required component

Table 1–1 (Cont.): Runtime System Components

Component	Description
DECwindows User-Environment Components	Provide Window Manager and terminal emulator support.
Command Language Utility (ECL)	Provides an interactive interface you can use to maintain files, execute programs, and control the runtime system environment.
Display Utility (EDISPLAY)	Displays system-level and job-specific resource information on a target system video terminal.
Performance Utility Collector	Collects application program performance data.
Remote Terminal Utility	Lets you connect to a remote computer system from a terminal on another computer system by using a SET HOST command
LAT Control Program Utility (LATCP)	Provides an interactive interface you can use to manage and monitor local area transport (LAT) service node characteristics and activities
Local Debugger Component[2]	Lets you debug a VAXELN application from the target processor's console terminal.

[2]Usually not included in final system

1.2 VAXELN Programming Concepts

A VAXELN application's design and development are based on the concept of *concurrency*, the simultaneous execution of multiple programs and parts of programs. Concurrency is a proven approach for applications that require cooperation among programs to solve specific problems quickly and efficiently.

VAXELN programs execute as *jobs*. A typical VAXELN application consists of multiple jobs that each have functionally independent components called *processes*.

1.2.1 Processes: Execution Agents for Programs and Program Parts

A process is a functionally independent entity that provides the execution context for a program image or part of a program image. Each process in a VAXELN system represents a specialized task. The main section of program code (the *program block* for VAXELN Pascal programs, the main routine for C programs, and the main program for FORTRAN programs) executes as the *master process*. The kernel creates this process implicitly when the program starts executing.

1.2.2 Jobs: Families of Processes

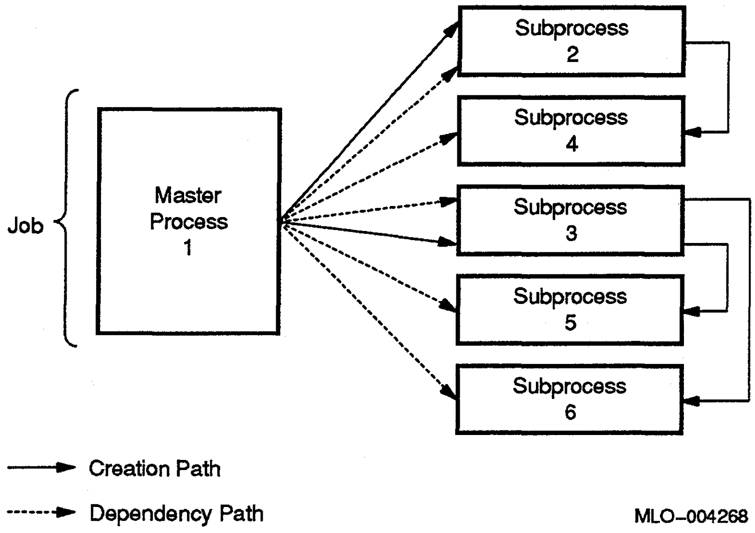
Collectively, the processes associated with a running program constitute a job. A job consists of a master process and zero or more subprocesses that can execute concurrently.

A job can be thought of as a family of processes. A job's master process and subprocesses create other subprocesses dynamically. Once created, a process stays active until it exits, another process deletes it, its master process terminates, it encounters an error from which it cannot recover, or it finishes executing the associated code segment. A programmed exit (see Section 3.2) is the most controlled means of forcing process termination.

Figure 1–4 illustrates the creation and dependency paths for a process family consisting of a master process and five subprocesses.

When a master process terminates under any circumstances, the kernel removes the corresponding job, its master process and associated subprocesses, and shared data from the system and replenishes the system's memory resources.

Figure 1–4: Process Family



1.2.3 Concurrency: Processes Sharing Processor Resources

To take advantage of the VAXELN Toolkit's realtime efficiency, you design applications with the concept of concurrency in mind. Concurrency is built into the VAXELN software so that cooperating processes can share processor resources. While some processes wait for an event to occur or a resource to become available, other processes can execute. The kernel manages system resources so that all jobs and processes appear to execute simultaneously, although only one process actually executes on a processor at a time.

You determine whether jobs and processes should execute concurrently when designing your application. Concurrent programming has numerous system design advantages, including improved performance.

The VAXELN Kernel supports three levels of concurrency — *multitasking*, *multiprogramming*, and *multiprocessing* — which are described in Sections 1.2.3.1, 1.2.3.2, and 1.2.3.3, respectively.

1.2.3.1 Multitasking

Multitasking lets you divide an application program's functionality into a set of smaller, focused tasks that can execute concurrently. Each task executes as a separate dedicated process. For example, a program controlling a wing in an aircraft flight simulation application might consist of processes that specialize in tasks such as surface control and engine fire-up.

1.2.3.2 Multiprogramming

Multiprogramming is the concurrent execution of entire programs, including multitasking programs. The programs execute as jobs that may or may not run cooperatively; that is, Job A may or may not depend on Job B. However, the jobs of most VAXELN systems work together to accomplish mutual goals. For example, in an aircraft flight simulation application, a collection of cooperating jobs might emulate major components of an airplane, such as cockpit controls and instrumentation, navigation equipment, and left and right wings.

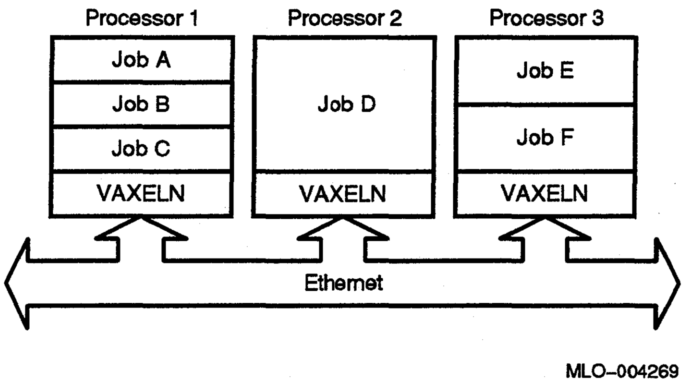
1.2.3.3 Multiprocessing

A VAXELN application's jobs can reside on one processor or they can be distributed among multiple processors. The concurrent execution of a VAXELN application's parts on multiple processors is called multiprocessing. The VAXELN Kernel supports the following multiprocessing configurations:

- Loosely coupled symmetric multiprocessing
- Tightly coupled symmetric multiprocessing
- Closely coupled symmetric multiprocessing

In a *loosely coupled symmetric configuration*, an Ethernet device links the processors, as shown in Figure 1–5. Each processor runs its own system image with its own jobs.

Figure 1–5: Loosely Coupled Multiprocessing Configuration

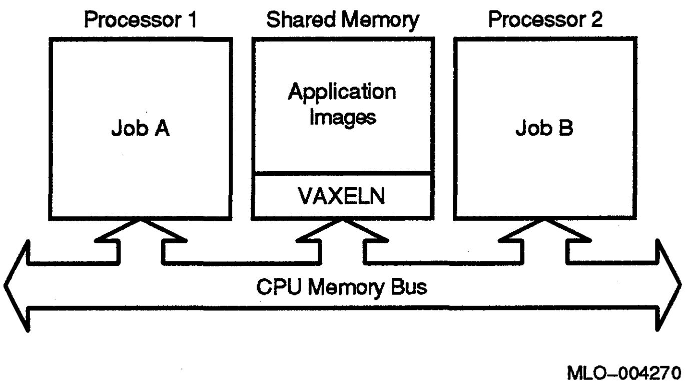


In a *tightly coupled symmetric configuration,* the hardware supports multiple processors on the same CPU bus, as shown in Figure 1–6. VAXELN supports tightly coupled symmetric multiprocessing on VAX 6000 series and VAX 8800 multiprocessor configurations. All processors share a copy of the VAXELN runtime components and application images. A job can execute on any processor (the default) or you can limit it to a specific subset of processors.

A *closely coupled symmetric configuration* consists of a VAX 6000 series, 8500, 8530, 8550, 8700, or 8800 primary system and one or more KA800 single-board computers (SBCs). The primary system can be a single processor or a tightly coulpled symmetric multiprocessing configuration. Each KA800 system is connected to the primary system's VAXBI bus and has its own copy of the VAXELN runtime components and application images.

Closely coupled configurations provide limited data sharing capabilities. Data in the primary system's memory is shareable and can be accessed by the attached KA800 systems. However, the primary system cannot gain access to data that is in the memory of the KA800 systems.

Figure 1–6: Tightly Coupled Symmetric Multiprocessing Configuration



As shown in Figures 1–7 and 1–8, the primary processor in a closely coupled environment can run a VAXELN or VMS system. When the primary processor is running a VAXELN system, you down-line load VAXELN systems into the KA800 processors by using a configuration file, a runtime procedure call, or an ECL command.

When the primary processor is running a VMS system, you use VAX Real-time Accelerator (RTA) software to load, control, and communicate with VAX RTA KA800 processors. For information about VAX RTA, see the *VAX Real-Time Accelerator Hardware / Software Installation Guide* and *VAX Real-Time Accelerator Software User's Guide.*

A common application for closely coupled multiprocessing is to distribute realtime I/O functions. You can achieve superior performance by offloading interrupt-intensive tasks to KA800 processors, freeing the primary processor for other functions. The KA800 processors can directly control the DRB32 direct memory access (DMA) parallel port device to distribute I/O control for high-speed data transfers and fast, predictable interrupt response time.

Figure 1–7: Closely Coupled Symmetric Multiprocessing Configuration with VAXELN Primary System

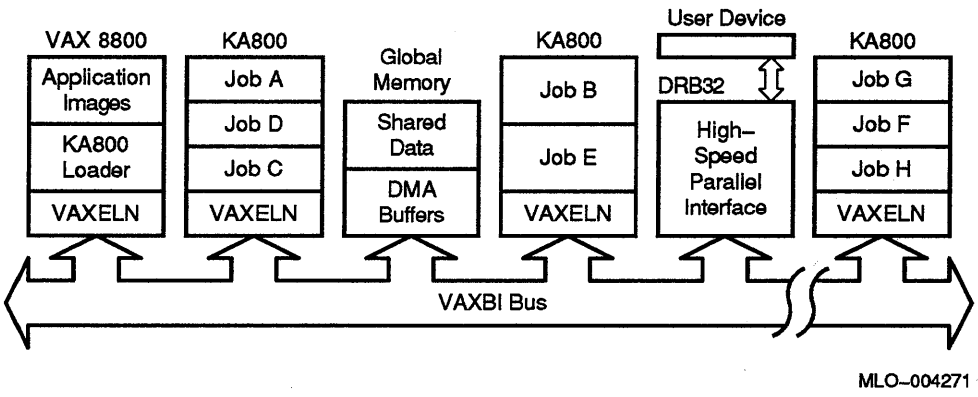
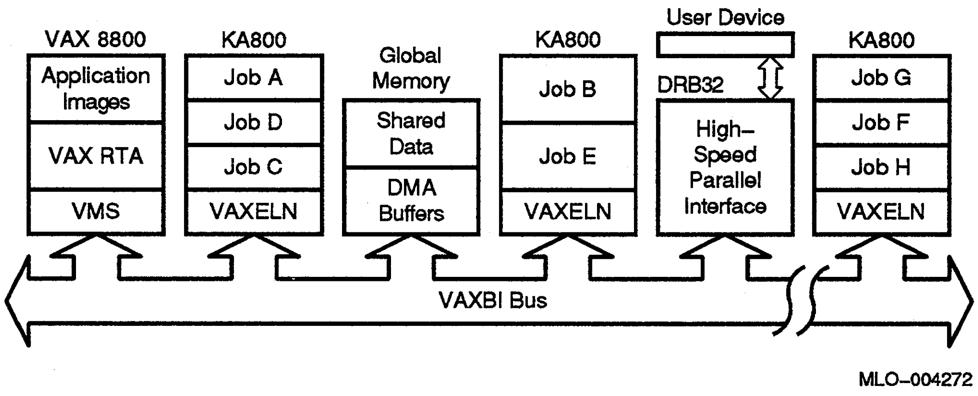


Figure 1–8: Closely Coupled Symmetric Multiprocessing Configuration with VMS Primary System



1.3 VAXELN Runtime Facilities

The VAXELN runtime components provide a rich software environment for programming dedicated realtime applications. These components consist of a kernel executive and a variety of runtime services that provide support for:

- Networking, Section 1.3.2
- Local area transport (LAT) communication, Section 1.3.3
- System security, Section 1.3.4
- File oriented disk and tape I/O, Section 1.3.5
- Device drivers, Section 1.3.6
- DECwindows, Section 1.3.7

1.3.1 Kernel

The *VAXELN Kernel* defines a set of objects that it uses to control the sharing of resources and to synchronize communication between the jobs in a system. The kernel manipulates these objects in response to procedure calls that are issued from application programs. In addition, the kernel provides the following types of facilities to both user and system programs:

- Process, job, and memory management
- Process synchronization
- Communication
- Device and interrupt handling
- Exception handling

Chapter 2 describes the kernel data structures and the operations in which they can be used. Chapter 3 explains how the kernel manages processes, jobs, and memory. Chapters 4 to 7 discuss synchronization, communication, device handling, and exception handling, respectively.

1.3.2 Network Services

The VAXELN Toolkit includes Ethernet/IEEE 802 datalink drivers for supported network devices. Each of the datalink drivers supports the VAXELN Ethernet/IEEE 802 Datagram Service, VAXELN Network Service, and VAXELN Internet Services. The Datagram Service provides network interface routines that VAXELN systems can use to communicate with other types of systems using system-independent communications protocols.

The Network Service is a supplied program image that controls message transmission between network nodes, manages a network name table, and provides a runtime interface for managing a DECnet network. You configure a Network Service for each target node used in a multinode application. The Network Service preserves the methods for sending and receiving messages, whether jobs communicate on the same node or between nodes; data transmission across network nodes is transparent to your programs.

The VAXELN Internet Services support Internet networking protocols over an Ethernet medium. The services consist of runtime routines that applications can use to control the Internet Services, convert byte order of Internet and host physical addresses, manipulate Internet addresses, communicate over the Internet using sockets, and retrieve and set socket characteristics.

Chapter 8 describes the datalink drivers and explains how to use the Datagram Service. Chapters 9 and 10 describe VAXELN DECnet and Internet Services, respectively.

1.3.3 LAT Host Services

The local area transport (LAT) host services enable VAXELN system nodes running LAT host services to communicate with devices attached to dedicated terminal server nodes running LAT server services. Using these services, VAXELN applications can perform terminal I/O operations and can use control interfaces to manage and monitor LAT environments. In addition, the LAT host services support a utility that you can use to manage and monitor a VAXELN LAT environment interactively.

See Chapter 11 for more information.

1.3.4 Authorization Service

The VAXELN Toolkit includes an optional Authorization Service that provides system security for network applications. The Authorization Service protects system resources and data by maintaining a data base of a system's authorized users and identifying users who issue network requests.

The Network Service and File Service use the Authorization Service to protect the resources and data that they control. The Network Service running on a particular node accepts circuit connections only from users who are listed in the Authorization Service's data base. The File Service provides read, write, and delete protection for files on disks that it controls. Likewise, your application programs can use the service to protect their resources and data.

See Chapter 12 for more information.

1.3.5 File Service

The File Service is a set of system disk and tape driver services that enable VAXELN application programs to perform file-oriented disk and tape I/O operations. The File Service consists of a disk File Service and a tape File Service and provides for remote file access.

The disk File Service uses FILES–11 On-Disk Structure Level 2 services and is compatible with the VMS, Version 4.4, file system and the VMS record management services (RMS). Files are sequentially organized. Programs can use sequential or random access for creating, reading, and writing sequential disk files.

The tape File Service is based on Version 3 of the ANSI-standard magnetic tapes and is compatible with the VMS, Version 4.4, tape file system. You can use this service to transport files to and from VMS systems.

See Chapter 13 for more information.

1.3.6 Device Drivers

The VAXELN Toolkit simplifies VAX device support by providing pregenerated device drivers that you can include in your VAXELN systems. These drivers provide support for a variety of disk, tape, printer, terminal, Ethernet, and realtime devices.

See Chapter 14 for more information.

1.3.7 DECwindows Support

The VAXELN Toolkit provides DECwindows support for creating network transparent distributed applications that perform two-dimensional, integer coordinate drawing and windowing operations. The toolkit includes the following DECwindows software:

- A DECwindows server image that you can build into VAXELN systems that run on the following workstations:

  VAXstation II/GPX
  VAXstation 2000
  VAXstation 3100 series (color video option)
  VAXstation 3200 (color video option)
  VAXstation 3500

- The DECwindows runtime libraries and tools you need to develop VAXELN DECwindows client applications
- A Window Manager and terminal emulators that enhance the user environment for VAXELN DECwindows client applications

See *VAXELN Guide to DECwindows* for more information.

Chapter 2

The VAXELN Kernel

The VAXELN Kernel is a small, realtime executive that controls target hardware resources and the execution of VAXELN system software. VAXELN applications typically require fast, predictable responses to interrupts. To meet this crucial need, the highly optimized kernel takes advantage of the VAX architecture and imposes minimal overhead between the application code and the hardware.

The kernel recognizes and operates on a set of realtime programming data structures, which it uses to control the sharing of resources and to synchronize communication between the jobs in a system. These structures include a set of *kernel objects* and two specialized structures called *mutexes* and *area lock variables*. The objects represent ongoing activities, such as process execution, and hardware and software resources, such as devices, memory regions, events, and messages. Mutexes and area lock variables are optimizations of kernel objects. Table 2–1 describes the kernel objects, and Table 2–2 describes the optimized structures.

Each VAXELN Kernel data structure is associated with a corresponding set of operations that are implemented as procedure calls. The kernel manipulates the structures and the resources associated with them in response to procedure calls that you issue from your application programs. Your high-level language programs call the kernel procedures directly to synchronize processes, to communicate between jobs or processes, and to handle device interrupts.

The kernel also handles system scheduling and memory allocation and maintains information about the entire VAXELN system and each system component — that is, the context for the system image and each program image.

This chapter describes and summarizes the kernel operations for the following:

- Kernel objects, Section 2.1
- Optimized data structures, Section 2.2

Chapters 3 to 6 describe the operations that the kernel performs in more detail.

2.1 Kernel Objects

The VAXELN Kernel objects represent ongoing activities, such as process execution, and hardware and software resources, such as devices, memory regions, events, and messages.

To guarantee the integrity of a kernel object, its fields are not directly accessible to a program. Instead, when the program calls the kernel to create a new object, the kernel dynamically allocates a block of memory for the object and returns an identifying value for it. You then refer to the object by specifying the identifying value in subsequent calls to kernel procedures. When you no longer need the object, you specify the identifying value in a call to the DELETE procedure.

In the VAXELN Pascal language, predeclared data types represent the kernel objects' identifying values. These predeclared types are AREA, DEVICE, EVENT, MESSAGE, NAME, PORT, PROCESS, and SEMAPHORE. To create and use an object, a program declares a variable of the object's type, calls the appropriate CREATE_*object_type* kernel procedure, and saves the returned, object value in the variable. The variable then assumes the object's identifying value, which you can use throughout the program to name the object. For example, the following lines of code declare a variable of type SEMAPHORE, create a SEMAPHORE object, and save the returned identifying value in the variable *main_lock*:

```
VAR
  main_lock : SEMAPHORE;
    .
    .
    .
BEGIN
    .
    .
    .
  CREATE_SEMAPHORE(main_lock);
    .
    .
    .
END.
```

You can then wait on or signal the semaphore anywhere in the program by using the variable to reference the object as follows:

```
  WAIT_ANY(main_lock);
    .
    .
    .
  SIGNAL(main_lock);
```

When the program no longer needs the object, you can delete it with a call to the DELETE procedure as follows:

```
  DELETE(main_lock);
```

The VAXELN Toolkit also provides kernel interfaces for VAX C and VAX FORTRAN programming. The data type definitions for the two languages are provided in the following definition modules:

Language	**Module**
C	**$vaxelnc** in VAXELNC.TLB
FORTRAN	'ELN$:FORTRAN_DEFS.FOR'

NOTE

Except for PORT values and AREA values for jobs on the same node, an object's identifying value is valid only within a job, even when the object is known in more than one job.

Table 2–1 summarizes the kernel objects. Sections 2.1.1 to 2.1.8 describe the objects in detail.

Table 2–1: Kernel Objects

Object	Description
AREA	Represents a region of physical memory accessible to all jobs executing on the same node in a local area network.
DEVICE	Represents a channel to an I/O device and associates an interrupt service routine (ISR) with the device's interrupt. DEVICE objects synchronize ISR and device driver process execution.
EVENT	Represents a flag that identifies the occurrence of a realtime event. Events synchronize process execution and access to shared data.
MESSAGE	Represents data that is transmitted between processes. Messages can be sent between two processes, two jobs, or two nodes in a local area network.
NAME	Represents an entry in a name table that associates a character string name with a message port or process. Port names can be local (known only on its own node) or universal (known on any node in the local area network).
PORT	Represents a system-maintained store for messages being sent or waiting to be received. Only the processes in the job that creates the port can receive messages from that port. However, any process in any job can send a message to the port. A program can connect two ports in the same or different jobs to form a circuit, which simplifies and increases the reliability of communication between jobs.
PROCESS	Represents a functionally independent entity that provides the execution context for a program image or part of a program image. The main program executes as a master process, which can control zero or more subprocesses. Collectively, a master process and its subprocesses constitute a job.
SEMAPHORE	Represents a synchronization gate that controls access to a shared resource. Binary semaphores enforce exclusive access to a resource. Counting semaphores permit metered access, allowing a specified number of processes simultaneous access to units of a resource.

2.1.1 AREA Objects

An AREA object represents a region of memory or another type of shared resource that can be shared among jobs on a single node in a VAXELN network. An AREA object contains an event or semaphore that can be used by the sharing jobs to synchronize access to the area's data. Areas with a size of 0 are valid and represent only the event or semaphore.

An AREA object has the following properties:

- A character string name of up to 31 characters that supplies a name for the area
- A *signaled* or *cleared* state if the area is associated with an event
- A count of the number of processes that can gain access to the area (or resource) without waiting for some other process to signal the area if the area is associated with a semaphore
- The maximum allowed value for the count, which is the maximum number of processes that can gain access to the area (or resource) simultaneously, if the area is associated with a semaphore
- A list of processes waiting for access to the area
- The associated region of memory

Chapter 5 discusses these properties and the kernel procedures that affect AREA objects.

AREA values are represented internally as 32-bit longwords. The kernel uses the longwords to locate AREA objects and their properties.

An AREA object occupies one block (128 bytes) of kernel pool.

The kernel allocates the region of memory associated with an area from physically contiguous 512-byte pages of physical memory and maps the region into the creating job's P0 virtual address space. The region occupies an integral number of memory pages and is aligned on a page boundary.

The following table lists the operations for which you can use AREA values and the procedures an application calls to perform the operations:

Operation	**Procedure**
Create an area or map an existing area, return an identifying AREA value and pointer to the area, and associate the area with an event or semaphore.	CREATE_AREA CREATE_AREA_EVENT CREATE_AREA_SEMAPHORE
Gain exclusive access to an area by waiting for that area to be signaled.	WAIT_ALL WAIT_ANY

Operation	Procedure
Signal the event or semaphore that is associated with an area.	SIGNAL
Clear an event associated with an area.	CLEAR_EVENT
Delete an area.	DELETE

2.1.2 DEVICE Objects

A DEVICE object represents a channel to an I/O device and associates an interrupt service routine (ISR) with the device's interrupt. When the device issues an interrupt, the kernel calls the device's ISR to service the device.

A DEVICE object has the following properties:

- A set of device characteristics established with the System Builder
- A communication region that lets a device driver and its ISR share data
- An ISR, which the kernel invokes when an appropriate interrupt occurs and to which the kernel passes the DEVICE value and communication region

Chapter 6 discusses these properties and the kernel procedures that affect DEVICE objects.

DEVICE values are represented internally as 32-bit longwords. The kernel uses the longwords to locate the DEVICE objects and their properties, such as the address of its communication region. DEVICE values are valid only within their own job.

A DEVICE object occupies one block (128 bytes) of kernel pool. If an ISR is connected, it also requires one block of pool or a page of communication region for its dispatcher.

The following table lists the operations for which you can use DEVICE values and the procedures an application calls to perform the operations:

Operation	Procedure
Create a DEVICE object and return an identifying DEVICE value.	CREATE_DEVICE
Wait for an ISR to signal a DEVICE object.	WAIT_ALL WAIT_ANY
Signal a DEVICE object from an ISR.	SIGNAL_DEVICE
Delete a DEVICE object.	DELETE

2.1.3 EVENT Objects

An EVENT object represents a flag that identifies the occurrence of a realtime event. Events synchronize process execution and access to shared data. An EVENT object records events in real time and stores that information until explicitly cleared by a program.

An EVENT object has the following properties:

- Either a *signaled* or a *cleared* state
- A list of processes waiting for the event to be signaled

Chapter 4 discusses these properties and the kernel services that affect EVENT objects.

EVENT values are represented internally as 32-bit longwords. The kernel uses the longwords to locate EVENT objects and their properties, such as the object state. An EVENT value is valid only within its own job unless the value is associated with an area (see Section 2.1.1).

An EVENT object occupies one block (128 bytes) of system pool.

The following table lists the operations for which you can use EVENT values and the procedures an application calls to perform the operations:

Operation	Procedure
Create an event and return an identifying EVENT value.	CREATE_EVENT
Wait for the signaling of an event.	WAIT_ALL WAIT_ANY

Operation	Procedure
Signal an event.	SIGNAL
Clear an event.	CLEAR_EVENT
Delete an event.	DELETE

2.1.4 MESSAGE Objects

A MESSAGE object represents data that is transmitted between processes. Messages can be sent between two processes, two jobs, or two nodes in a local area network.

A MESSAGE object has the following properties:

- Message data
- Message length

Chapter 5 discusses these properties and the kernel procedures that affect MESSAGE objects.

MESSAGE values are represented internally as 32-bit longwords. The kernel uses the longwords to locate MESSAGE objects and their properties. MESSAGE objects are valid only within their own job.

The associated message data is allocated in contiguous 512-byte pages of physical memory and is mapped by the creating or receiving job's P0 virtual address space. Therefore, the data always occupies an integral number of memory pages and is aligned on a page boundary. (These characteristics suit the message data well for a VAX DMA-device I/O buffer.) Since P0 address space is used, all processes in a job can share the message data.

The following table lists the operations for which you can use MESSAGE values and the procedures an application calls to perform the operations:

Operation	Procedure
Create a message, map its data into the job's P0 address space, and return an identifying MESSAGE value and a pointer to the data.	CREATE_MESSAGE

Operation	Procedure
Send a message to a message port and remove the message data from the sending job's address space.	SEND
Remove a message from a message port, map the message data into the receiving job's P0 address space, and return an identifying MESSAGE value and a pointer to the message data.	RECEIVE
Delete a message.	DELETE

2.1.5 NAME Objects

A NAME object represents an entry in a name table that associates a character string name with a message port or process.

Name objects have the following properties:

- A character string of up to 31 characters that names an existing message port or process
- The value of the message port or process being named
- For port name objects, the property *local* or *universal*

Port name objects and their associated character strings are stored in either a local or a universal name table. The kernel maintains the local name table for name objects used within a node. The Network Service helps to maintain the universal name table; it contains valid name objects for nodes in the local area network.

NOTE

The processors in a closely coupled symmetric multiprocessing configuration constitute one Ethernet node and share the same local name table. Therefore, the images running on the processors must create unique local names.

A NAME object for a process is not kept in a name table; it is associated with a PROCESS object.

Chapter 5 discusses these properties and the kernel procedures that affect NAME objects for processes and message ports, respectively.

Identifying NAME values are 32-bit longwords that are valid only within their own job. A NAME object occupies one block (128 bytes) of kernel pool. A universal name also requires 64 bytes of dynamic memory in the local Network Service and 64 bytes in the system acting as the network's current *name server*. (See Chapter 9 for more information.)

The following table lists the operations for which you can use NAME values and the procedures an application calls to perform the operations:

Operation	**Procedure**
Create a name and an identifying NAME value.	CREATE_NAME
Return the PORT value associated with a name (not valid for process names).	TRANSLATE_NAME
Name a process by creating a unique NAME object that associates a character string with a process.	KER$NAME_OBJECT (Pascal only)
Delete a name.	DELETE

2.1.6 PORT Objects

A PORT object represents a system-maintained store for messages being sent and waiting to be received. Only processes in the job that creates a port can receive messages from that port. However, any process in any job can send messages to a port.

Each executing job in a system has a unique message port, or *job port*, created when the first process in the job is started. A job can use its job port to receive messages from other jobs. Programs can create additional message ports dynamically with the CREATE_PORT procedure.

A PORT object has the following properties:

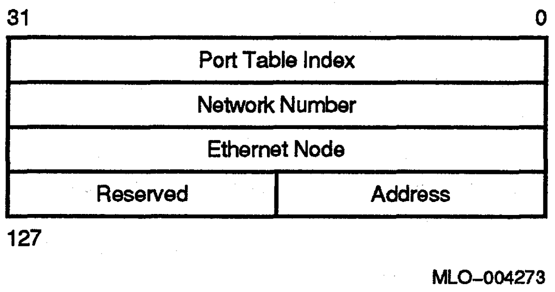
- The maximum number of queued messages
- A list of queued messages, to be removed from the port by the RECEIVE procedure
- The state of the port's circuit: unconnected, connected, or in a special state during the establishment of a connection

- If connected, the PORT value identifying the port to which the port object is connected

Chapter 5 discusses these properties and the kernel procedures that affect the state of PORT objects.

PORT values are 128-bit values that identify a message port as shown in Figure 2–1.

Figure 2–1: PORT Value Representation



Each PORT object occupies one block (128 bytes) of kernel pool and requires one entry in the kernel's port address table.

The following table lists the operations for which you can use PORT values and the procedures an application calls to perform the operations:

Operation	Procedure
Create a port and return an identifying PORT value.	CREATE_PORT
Return a unique PORT value for the calling job for communicating between jobs.	JOB_PORT

Operation	Procedure
Wait to receive a message.	WAIT_ALL WAIT_ANY
Connect and disconnect circuit ports.	CONNECT_CIRCUIT DISCONNECT_CIRCUIT
Let the calling process wait for a circuit connect request on a port.	ACCEPT_CIRCUIT
Delete a port.	DELETE

When a message arrives at a port, any process waiting on that port can continue if its wait conditions are satisfied. The receiver process calls the RECEIVE procedure to get the message. Only processes in the job that creates a port can receive messages from that port with RECEIVE.

2.1.7 PROCESS Objects

A PROCESS object represents a functionally independent entity that provides the execution context for a program image or a part of a program image. The main program executes as a master process, which can control zero or more subprocesses. Collectively, a master process and its subprocesses constitute a job. A job can contain any number of processes within a limit of 4096 objects for each job.

A PROCESS object has the following properties:

- One of 16 levels of process priority
- One of the process states *running*, *ready*, *waiting*, or *suspended*
- A user name and a user identification code (UIC)

Chapter 4 discusses these properties and the kernel services that affect PROCESS objects.

PROCESS values are represented internally as 32-bit longwords. They are valid only within their own job.

The following table lists the operations for which you can use PROCESS values and the procedures an application calls to perform the operations:

Operation	Procedure
Create a process and return an identifying PROCESS value.	CREATE_PROCESS
Get the PROCESS value of the calling process.	CURRENT_PROCESS
Set a process's priority.	SET_PROCESS_PRIORITY
Suspend a process's execution.	SUSPEND
Resume execution of a process.	RESUME
Wait for another process to terminate.	WAIT_ALL WAIT_ANY
Force another process into an exception condition.	SIGNAL
Exit from a process.	EXIT
Delete a process.	DELETE

2.1.8 SEMAPHORE Objects

A SEMAPHORE object represents a synchronization gate that controls access to a shared resource. Binary semaphores enforce exclusive access to a resource. Counting semaphores permit metered access, allowing a specified number of processes simultaneous access to units of a resource.

A SEMAPHORE object has the following properties:

- A count of the number of processes that can gain access to the resource without waiting for some other process to signal the semaphore
- The maximum allowed value for the count, which is the maximum number of processes that can gain access to the resource simultaneously
- A list of processes waiting for the semaphore to be signaled

Chapter 4 discusses these properties and the kernel procedures that affect SEMAPHORE objects.

A SEMAPHORE object occupies one block (128 bytes) of system pool.

SEMAPHORE values are represented internally as 32-bit longwords. The kernel uses the longwords to locate SEMAPHORE objects and their properties, such as its current count. A SEMAPHORE value is valid only within its own job unless the value is associated with an area (see Section 2.1.1).

The following table lists the operations for which you can use SEMAPHORE values and the procedures an application calls to perform the operations:

Operation	Procedure
Create a semaphore and return an identifying SEMAPHORE value.	CREATE_SEMAPHORE
Wait for the signaling of a semaphore.	WAIT_ALL WAIT_ANY
Signal a semaphore.	SIGNAL
Delete a semaphore.	DELETE

2.1.9 Kernel Object Implementation

Although it is usually not necessary for a VAXELN programmer to know the details of the kernel's implementation of objects, the following points are useful in answering system configuration questions:

- The kernel allocates all objects, except PROCESS objects, from a pool of fixed-length blocks of memory. The number of blocks in the pool is set with the System Builder. When the system is booted, the kernel initializes the pool, maps the blocks into system space, and links the blocks into a list of free blocks. The fixed size of the blocks makes allocating and deallocating objects efficient.
- The identifying value returned by the kernel for a newly created object is not the virtual address of the object. Instead, it is a 32-bit value consisting of two indexes. The indexes are used to look up the address of the object in a two-level table maintained by the kernel for each job. These values are thus unique for each job in the system.

- The object table grows dynamically as the job creates more objects. The kernel allocates the table from system memory and pool blocks. The top-level table is allocated in a 512-byte page of memory that can hold pointers to 128 second-level tables. Each second-level table occupies one 128-byte pool block that can hold up to 32 object addresses. Thus, you can create up to 4096 objects for a job.

The preceding description applies to all objects except ports. Because a PORT value is valid anywhere in the network, it also includes the DECnet or Ethernet node address and additional fields reserved for future use. Thus, a PORT value is 128 bits long. Also, the indexes in a PORT value are used for a table that describes all the ports in the system, rather than just the ports in a job. The size of the port table is also set with the System Builder, and the table is allocated by the kernel when the system is booted.

Although the kernel's method for representing identifying values might seem complicated, it allows you to validate identifying values in a few VAX instructions. Furthermore, the method of representation is not important for VAXELN programming.

2.2 Optimized Data Structures

The kernel also recognizes and operates on two specialized data structures: mutexes and area lock variables. These structures are optimizations of kernel objects; locking a mutex can be faster than waiting on a mutual exclusion semaphore, and locking an area synchronization variable can be faster than waiting on a shareable memory area.

The locations of the MUTEX and AREA_LOCK_VARIABLE data type definitions are as follows:

Language	Module
VAXELN Pascal	$MUTEX in the RTLOBJECT.OLB
C	**$mutex** in the VAXELNC.TLB
FORTRAN	'ELN$:FORTRAN_DEFS.FOR'

Table 2–2 summarizes the optimized structures. Sections 2.2.1 and 2.2.2 describe the structures in detail.

Table 2–2: Optimized Data Structures

Structure	Description
AREA_LOCK_VARIABLE	Represents a variable that resides in an area object for synchronizing job access to the associated area. Using this variable, a process can lock an area to gain exclusive access. When the process locks the area, the process does not have to issue a wait before accessing the associated area unless the area is already locked.
MUTEX	Represents an optimized binary semaphore. A process can lock a mutex to gain exclusive access to a shared resource. When the process locks the mutex, the process does not have to issue a wait before accessing the resource unless the mutex is already locked.

2.2.1 AREA_LOCK_VARIABLE Data Structure

The AREA_LOCK_VARIABLE data structure provides an alternative means for synchronizing access to areas between jobs. Area lock operations can be used to improve the performance of AREA wait and signal operations.

Area-locking operations enable jobs to synchronize access to an area by using a synchronization variable of type AREA_LOCK_VARIABLE in the area's data portion. You can use an area lock variable only if the area is created with an associated binary semaphore that is properly initialized. You can do this with CREATE_AREA, and its implied binary semaphore, or with CREATE_AREA_SEMAPHORE with initial and maximum counts of 1. No error status is returned if you use an AREA_LOCK_VARIABLE with an area that is not associated with a binary semaphore.

Area-locking operations can be more efficient than calling the WAIT_ANY and SIGNAL procedures with areas. When a process locks an area to gain exclusive access, the process does not have to call a WAIT_ANY procedure unless some other process has already locked the area.

The following table lists the operations for which you can use area lock variables and the procedures an application calls to perform the operations:

Operation	Procedure
Initialize (unlock) a synchronization variable (of type AREA_LOCK_VARIABLE) in the data portion of an area.	ELN$INITIALIZE_AREA_LOCK
Lock (wait on) an area.	ELN$LOCK_AREA
Unlock (signal) an area.	ELN$UNLOCK_AREA

An area lock variable is represented internally as a 16-bit counter. The variable must be within an area's data portion. A single process in the application calls ELN$INITIALIZE_AREA_LOCK to initialize the counter to −1.

Once an area lock variable is initialized, subsequent calls to the ELN$LOCK_AREA and ELN$UNLOCK_AREA procedures increment and decrement the counter, respectively.

- When ELN$LOCK_AREA increments the counter and the result is greater than 0, the area has already been locked by another process. Thus, the procedure calls the WAIT_ANY procedure to wait for the area to be unlocked.
- When ELN$UNLOCK_AREA decrements the counter and if the result is greater than or equal to 0, another process is waiting for the area. To satisfy that wait, ELN$UNLOCK_AREA calls the SIGNAL procedure to unlock the area.

2.2.2 MUTEX Data Structure

The MUTEX data structure is an optimization of a binary semaphore. The meanings of mutex operations are similar to the comparable operations on binary semaphores. The difference is that when a process locks a mutex to gain access to a shared resource, the process does not have to call the WAIT_ANY procedure unless some other process has already locked the mutex. The result is significantly more efficient than that obtained using WAIT_ANY and SIGNAL procedures on binary semaphores.

The following table lists the operations for which you can use mutexes and the procedures an application calls to perform the operations:

Operation	Procedure
Initialize (unlock) a mutex and create an associated semaphore.	ELN$CREATE_MUTEX
Lock (wait on) a mutex.	ELN$LOCK_MUTEX
Unlock (signal) a mutex.	ELN$UNLOCK_MUTEX
Delete the semaphore created for a mutex.	ELN$DELETE_MUTEX

A mutex is represented internally as a 6-byte record containing a 16-bit counter and a SEMAPHORE value. A call to ELN$CREATE_MUTEX initializes the counter to −1 and the SEMAPHORE value to a binary semaphore with an initial count of 0.

Once a mutex is initialized, subsequent calls to the ELN$LOCK_MUTEX and ELN$UNLOCK_MUTEX procedures increment and decrement the counter, respectively.

- When ELN$LOCK_MUTEX increments the counter and the result is greater than 0, the mutex has already been locked by another process. Thus, the procedure calls the WAIT_ANY procedure to wait for the mutex to be unlocked.
- When ELN$UNLOCK_MUTEX decrements the counter and the result is greater than or equal to 0, another process is waiting for the mutex. To satisfy the wait, ELN$UNLOCK_MUTEX calls the SIGNAL procedure to unlock the mutex.

Deleting a mutex with the ELN$DELETE_MUTEX procedure sets the counter to 0, indicating that the mutex is locked. If you try to lock or unlock a mutex after it has been deleted, the internal call to WAIT_ANY fails and returns the status value KER$_BAD_VALUE.

Chapter 3

Job, Process, and Memory Management

The VAXELN Kernel manages jobs, processes, and system memory. The programs that comprise a VAXELN application execute as jobs. When you build a VAXELN system, the kernel creates a job for each program image that you specify; the images execute automatically when the system starts on the target hardware. The kernel also creates jobs in response to calls to the CREATE_JOB procedure and when you issue appropriate VAXELN debugger or ECL commands.

An application can use the CREATE_JOB procedure to create a job dynamically or to create a job after dynamically loading a program image with the dynamic program loader (see Section 3.3.4). A *program image* is a copy of all the code and initial data necessary to run the program.

A job consists of one master process that executes the program's main routine (program block, **main** function, or main program, depending on the language) and zero or more subprocesses that execute concurrently with the master process and with each other. The master process and subprocesses synchronize their activities by using the kernel objects, mutexes, and area lock variables and the associated procedures that manipulate them. The procedures create, delete, or otherwise affect the state of the structures represented by the data types AREA, AREA_LOCK_VARIABLE, DEVICE, EVENT, MESSAGE, MUTEX, NAME, PORT, PROCESS, and SEMAPHORE.

A program creates subprocesses by calling the CREATE_PROCESS procedure. Each subprocess executes a routine that defines the executable code and data available to one or more dynamically created processes. In VAXELN Pascal, C, and FORTRAN, these routines are called *process blocks*, *functions*, and *integer functions*, respectively.

A job can be thought of as a *process family*. The way processes are created implies a hierarchy: the CREATE_JOB procedure or the System Builder creates a job and a corresponding master process that runs a program; that program then can call the CREATE_PROCESS procedure to create subprocesses to execute the program's process blocks and functions. The subprocesses can also call CREATE_PROCESS to create subprocesses. Execution of the master process holds the object values of all subprocesses; thus, if the master process exits, all subprocesses and the memory and objects created by the job are deleted.

The processes in a job can share data that is declared externally (outer-level data). Jobs on a single node in a VAXELN network can share data by using AREA objects.

You can combine any number of jobs with the VAXELN runtime software to form a VAXELN system image. The VAXELN Kernel keeps track of the current jobs in a system. Therefore, if a program calls CREATE_JOB and then exits, the created job continues executing. With this procedure, a VAXELN program can create a new process family, in which the main program can be any program that was originally configured into the system or loaded with the dynamic program loader. The new job is independent of other jobs and has its own data and code. Similarly, multiple processes within a job can execute the same code segment.

When you build a system, you can specify any number of programs to execute when you load the system onto the target processor. A running VAXELN application can contain any combination of multitasking, multiprogramming, and multiprocessing job configurations.

A job remains active until the master process finishes executing its main routine code. A process remains active until it exits, another process deletes it, its master process terminates, it encounters an error from which it cannot recover, or it finishes executing the associated code segment. The exit operation provides the most controlled means of forcing process termination.

A process can delete itself or any other process within the same job. You cannot restart a deleted process; in general, you should use SIGNAL or EXIT to force a process to terminate.

When a job or master process terminates, the kernel deletes all the job's subprocesses and shared data from the system.

This chapter provides information about programming job, process, and memory management. The topics discussed include the following:

- Job activation and termination, Section 3.1
- Subprocess activation and termination, Section 3.2
- Scheduling, Section 3.3
- Kernel procedures for processes and jobs, Section 3.4
- Memory management, Section 3.5

3.1 Job Activation and Termination

The VAXELN Kernel creates a job implicitly when you select the **Run** option for a program image that you specify in the System Builder's Program Description Menu. The image executes automatically when the system starts on the target hardware. If you do not select the **Run** option, you can load a program and create jobs dynamically. You can load a program image by using one of the following:

- System Builder
- ELN$LOAD_PROGRAM procedure
- LOAD PROGRAM debugger command
- LOAD/PROGRAM or RUN ECL command (RUN also creates the job)

After the program image is loaded, you can:

- Use the CREATE_JOB procedure to create a job dynamically, using a program that was loaded with the System Builder
- Use the CREATE_JOB procedure after dynamically loading a program image with the ELN$LOAD_PROGRAM procedure
- Use the CREATE JOB debugger command to create a job
- Use the EXECUTE/WAIT ECL command to create a job

The LOAD PROGRAM and CREATE JOB debugger commands and the LOAD/PROGRAM, EXECUTE/WAIT, and RUN ECL commands are described in the *VAXELN Development Utilities Guide*.

When a job is created, the kernel establishes the job's P0 address space and the P1 address space (stack) for the job's master process (the program block). The processes in a job, including the master process and subprocesses, share the P0 space. Program arguments are stored in P0 space so that the PROGRAM_ARGUMENT function and the I/O runtime routines (for opening files) can access them.

The System Builder and Program Loader detect oversized jobs and issue appropriate warning messages. If you receive such a message, make sure you have allocated enough P0 virtual address space for each job in your system. The kernel will delete a job if not enough P0 space is available to create the job.

No files are open initially. However, you can implicitly open an input file, an output file, or a file named in the program block's header with the first I/O operation on that file.

The kernel activates the program block's routine body. It initializes data, using the program block's declaration section; then it executes the block's compound statement (BEGIN ... END).

A job terminates when the main routine's code completes execution, when the job's master process is terminated by the DELETE or EXIT procedure, or when an unhandled exception occurs (such as an unhandled QUIT exception caused when another process signals this process). When a job terminates, its existing subprocesses terminate, open files are closed, and the job's resources are returned to the kernel. If files are closed due to job termination, data in buffers can be lost. If you want the kernel to send a termination message to a specified port, use the NOTIFY parameter with the CREATE_JOB procedure.

You can use VAXELN utility procedures to establish an exit handler to perform cleanup operations following the termination of a job with the EXIT procedure (see Chapter 7).

3.2 Subprocess Activation and Termination

When a process in a job calls CREATE_PROCESS, the kernel creates a subprocess, establishes a new stack (P1 virtual address space) for the process, and prepares it for execution, beginning at the first statement in a process's routine code. The new process is in the ready state; it begins actual execution immediately or later, depending on its priority and the scheduling algorithms. (For information about process states and scheduling, see Section 3.3.)

A subprocess terminates when one of the following occurs:

- Execution of the main routine code terminates.
- The process calls the EXIT procedure.
- The process is deleted by a call to the DELETE procedure.
- An unhandled exception occurs in the process. (For example, an unhandled QUIT exception can occur when another process signals this process.)
- The job's master process terminates.

When a subprocess terminates, the kernel frees its P1 virtual address space (stack space) and the kernel pool space associated with the subprocess's activation. Objects it created and did not delete remain active, since the kernel cannot detect whether the object is in use by more than one process in the job. These objects are acquired by the job's processes that are deleted only when the job's master process is deleted.

NOTE

> Be careful when using the DELETE procedure to delete a process. Processes terminated by DELETE are not terminated in an orderly way and cannot be restarted. Deletion of a process is intended as an emergency method to stop a process; ordinarily, you should use SIGNAL or EXIT to terminate a process in an orderly way.

When terminating a process, the kernel also takes action so that:

- If another process of the job is currently waiting for the process to terminate, the wait is satisfied.
- If the call to CREATE_PROCESS that activated the process specified an *exit_status* argument, the exit status of the terminated process is stored in the designated data item.

These actions are not taken if the subprocess terminates because the master process terminated.

Processes are terminated in an orderly way with the SIGNAL or EXIT procedure or when they return from the outermost procedure block. (See Chapter 7 for a discussion of VAX stack architecture and call frames.)

The orderly termination of a process has two special consequences:

- The debugger notifies the user that the process is going away, if the debugger is active in the process.
- If the process is a master process (that is, if the job is terminating), the kernel activates an *exit handler* feature so that resources can be cleaned up by the code that allocated them.

When a process signals another process to quit, the quitting process can handle the raised exception KER$_QUIT_SIGNAL (see Section 3.4.18). The exception handler can perform special operations for the process, such as cleaning up resources, before the process exits.

A program can set up an exit handler by using the toolkit's exit utility procedures, ELN$DECLARE_EXIT_HANDLER and ELN$_CANCEL_EXIT_HANDLER. The ELN$DECLARE_EXIT_HANDLER procedure causes a program-defined exit handler routine to be called when the job terminates. When the exit handler routine is no longer needed, the program can delete it with a call to ELN$_CANCEL_EXIT_HANDLER.

3.3 Scheduling

The VAXELN Kernel schedules an application's execution based on a preemptive priority scheduling scheme that is driven by states and priorities of a system's jobs and processes. This scheduling scheme is described in Sections 3.3.1 to 3.3.5.

3.3.1 Processes and Process States

A process is a code segment that the kernel can schedule and execute independently as part of a VAXELN job. A process is created statically when you build your system or dynamically at runtime and remains active until it terminates. While active, a process is always in one of four *process states*: run, ready, wait, or suspend. Table 3–1 describes these states, and Figure 3–1 illustrates valid state transitions.

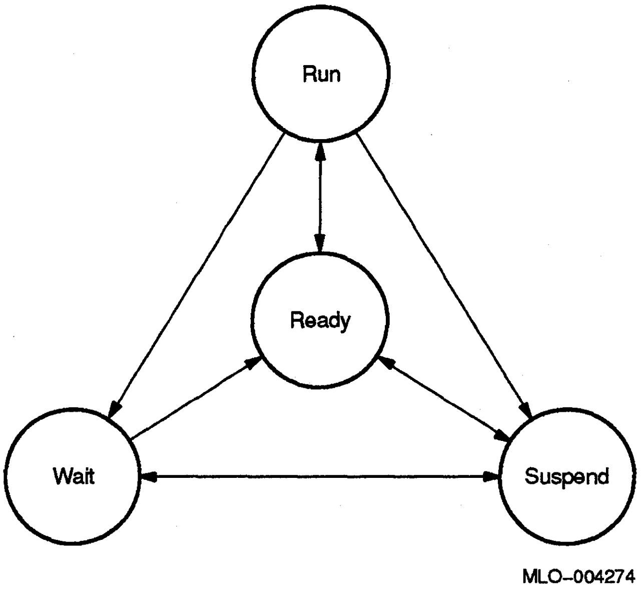
Table 3–1: Process States

State	Description
Run	The process has control of the processor and is currently executing.
Ready	The process is not executing but is ready to execute as soon as the scheduler allows. When an application creates a process, the process enters the ready state.
Wait	The process is waiting for a specified set of conditions to be satisfied, such as an amount of time to elapse, an event or series of events to occur, or the receipt of a message. A process enters the wait state by calling one of the following procedures: • WAIT_ANY — Wait for any of the listed conditions to be satisfied. • WAIT_ALL — Wait for all the listed conditions to be satisfied. • RESUME — Reenter the wait state if the process was waiting prior to being suspended with a call to the SUSPEND procedure. Another process must issue the call to RESUME.
Suspend	The process cannot reenter the ready state until another process in the same job reactivates the suspended process with a call to the RESUME procedure. A process can put itself or any other process in the same job into the suspend state with a call to the SUSPEND procedure.

The rules for process state transitions are as follows:

- Ready is the initial state for a process.
- When a process's wait conditions are satisfied, it enters the ready state. If the scheduling state of the system is such that the process should run immediately, the process enters the run state.
- The scheduler selects a ready process to enter the run state based on the system's jobs and process priorities.
- A process in the run state enters the ready state when the process is preempted by a higher priority process.
- A process in the run state enters the wait state when the process issues a call to WAIT_ANY or WAIT_ALL that blocks due to the wait conditions not being satisfied.

Figure 3–1: Valid Process State Transitions



- If a process is in the run or ready state when it is suspended, it enters the ready state when it is resumed. If the scheduling state of the system is such that the process should run immediately, the process enters the run state when it is resumed.
- If a process is in the wait state when it is suspended and not all the wait conditions are satisfied when the process is resumed, it reenters the wait state. If the scheduling state of the system is such that the process should run immediately, the process enters the run state when it is resumed.
- If a process was in the wait state when it was suspended and all the wait conditions are satisfied when the process is resumed, it enters the ready state.

3.3.2 Job and Process Scheduling

The order in which processes enter the run state depends on job and process scheduling. The VAXELN Kernel selects a process to run based on a preemptive, priority scheduling scheme; round-robin and time-sliced scheduling are not available.

To accommodate preemptive priority scheduling, you must assign a priority to each job and process in a VAXELN system. You can assign the priorities when you build the system, or you can change them dynamically with the procedures SET_JOB_PRIORITY and SET_PROCESS_PRIORITY. Job priorities can range from 0 to 31 (0 is the highest and 16 is the default). Process priorities can range from 0 to 15 (0 is the highest and 8 is the default). Therefore, within a job, processes can have 16 levels of priority independent of the job's priority.

Figure 3–2 illustrates the structure of job and process scheduling priorities.

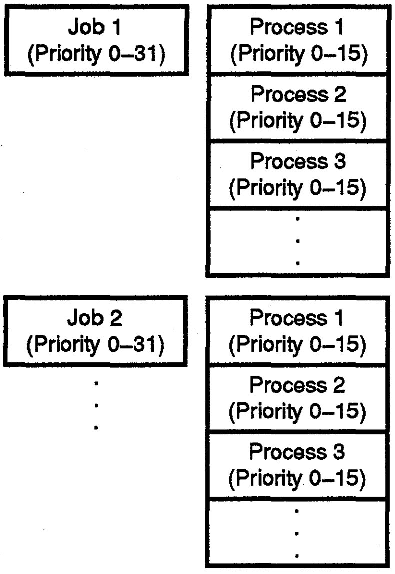
The VAXELN driver jobs run at higher priorities. For example, the datalink driver normally runs at job priority 1, the console driver runs at job priority 2, and the disk and tape drivers run at job priority 5. If an application includes one or more jobs that need to run at a job priority higher than that of the datalink driver and the jobs can run at the same job priority, you can set their job priorities to 0 and vary the process priorities.

The kernel scheduler considers a job ready to execute if one or more processes in that job are in the ready state. The kernel scheduler gives preference to the ready jobs and processes that have the highest priorities. The scheduler identifies the job with the highest priority and then selects that job's highest priority process for execution. The jobs in a system, whether they are executing or idle, are rescheduled when one or more of a job's processes enters the ready state.

Job rescheduling is illustrated by the following example, in which JOB1 has a higher priority than JOB2:

1. JOB1 has only one process, the master process; at a certain point, it executes WAIT_ANY to wait for a message to arrive at its job port.

Figure 3–2: Job and Process Priorities



2. JOB1 now has no processes in the ready state, so JOB2 is given control (assuming that at least one of its processes is ready).
3. When a message arrives at JOB1's port, the wait condition is satisfied, and JOB1's master process becomes ready again. Since JOB1's priority is higher, it is given control of the CPU again, preempting JOB2.

When two or more jobs have equal priority, the scheduler gives control to the ready process that has the highest priority among those jobs, preempting lower-priority processes.

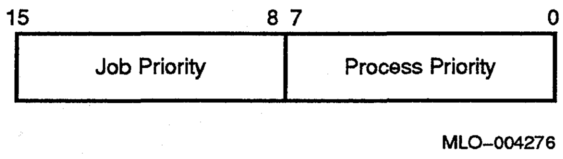
When a job is preempted and one or more jobs in the ready queue have the same job priority and the same highest priority ready process as that of the preempted job, the scheduler's action depends on the job preemption algorithm in effect. The default algorithm rotates the preempted job by placing it in the ready queue behind the jobs of equal job and process priority. However, if you selected *No* for the **Rotating job preempt** entry on the System Builder's System Characteristics

Menu when you built your system, the scheduler places the preempted job in the ready queue ahead of the jobs of equal job and process priority.

The scheduler's use of 32 job priorities and 16 process priorities might imply that the job and process priorities are unified to form one of 512 possible combined priority values and that the processes are scheduled against each other using this combined value. Rather, jobs are scheduled first followed by processes; the overall priority of a process, therefore, is limited by the priority of its job.

Figure 3–3 illustrates the internal representation of the combined job and process priority values.

Figure 3–3: Combined Priority Representation



Process rescheduling, or switching, within a job can be enabled and disabled with the procedures ENABLE_SWITCH and DISABLE_SWITCH. When switching is disabled, no other process in the current job can run. This feature provides a mechanism by which, for example, a process can control the access to a data set. (A finer mechanism is the use of semaphores, discussed in Chapter 4.)

Since process rescheduling is automatic and predictable, you can design systems that execute without noticeable delays — even though programs sit idle while others execute. In principle, the execution speed of an application is the speed of the slowest thread of execution.

The definition of important delay is essentially the definition of real-time performance for your application. It is impossible to exactly synchronize a computer or computer program with external phenomena. Instead, to satisfy the practical definition of *realtime*, the system must contain processes, which — given control of the CPU — can respond to external events in an acceptable amount of time. Furthermore, the processes should have high enough priority to ensure that they are not preempted while they are reacting to important external events.

Generally, realtime systems work best if the processes in charge of specific events are properly designed for, and synchronized with, those events. Only then should process priorities enter in, as a fine-tuning mechanism; priorities are not a means of synchronization. Chapter 4 summarizes issues related to synchronizing processes with each other or with external events.

For information about scheduling in multiprocessing configurations, see Section 3.3.5.

3.3.3 Initialization Programs and System Start-Up

When you use the System Builder to configure your program images, you can specify *Yes* for the **Init required** entry (see the *VAXELN Development Utilities Guide*). This characteristic means that the program is an initializing program that will be created and made eligible to run — along with other initializing programs, in order of job priority — when the system is started. Start-up of initializing programs precedes that of noninitializing programs.

While an initializing program runs, no jobs of lower priority are started until the program either calls the INITIALIZATION_DONE procedure or terminates. The INITIALIZATION_DONE procedure informs the kernel that the calling program has completed an initialization sequence, and other programs can be created and made eligible to run. (The calling program continues to run until some other occurrence causes it to block.)

The INITIALIZATION_DONE procedure makes it possible to synchronize the start of several programs in a system. For example, suppose a system has descriptions of the following programs:

program1	**Run, Init required, Priority** *5*
program2	**Run**
program3	**Run, Init required, Priority** *6*
program4	**Norun**

When the resulting system is started, the initializing programs are created and made eligible to run, one at a time, in the order of their job priorities, followed by the noninitializing programs. Here, *program1* is started first. (Remember that with job priorities, low numbers mean high priorities.) When *program1* calls INITIALIZATION_DONE, other initializing programs, beginning with *program3*, can be created and made eligible to run; meanwhile, *program1* continues running until some other occurrence causes it to block. If *program1* does not call INITIALIZATION_DONE, it must run to completion before *program3* or any other program is started.

Program2 is not started until both initializing programs have run or called INITIALIZATION_DONE. *Program4* is not started automatically; it must be activated by a CREATE_JOB call from one of the other programs, a debugger CREATE JOB command, or an ECL EXECUTE or RUN command.

3.3.4 Loading Programs

Normally, the programs that are available to run using the CREATE_JOB procedure are specified with the System Builder. To allow the system to react to new situations without being rebooted, however, VAXELN provides utility procedures that can be used to dynamically load and unload program images after the initial system is built. After a program image is dynamically loaded, CREATE_JOB is used to execute the program image.

The $LOADER_UTILITY module provides the following procedures:

- ELN$LOAD_PROGRAM, which loads a specified image file into a running system. The file is opened in the context of the caller, so the file name must be specified in enough detail to correctly identify the file. The file can reside on the system or on a remote node; you do not need to have a file system on the node to which the program is being loaded. Arguments specify the initial stack size, job and process priority, and whether or not the debugger should be given control when the program starts.
- ELN$UNLOAD_PROGRAM, which unloads the specified program from the system.

One restriction is that shareable images that the dynamically loaded program references must be included in the system at system build time. The **Guaranteed image list** entry on the Edit System Characteristics Menu allows you to specify the images that are needed by the dynamically loaded programs. These specified images are merged with those needed by other programs, and the System Builder resolves any interdependencies.

Another entry on the same menu, **Dynamic program space**, specifies the number of memory pages that can be used by dynamically loaded programs. The number is a quota and does not cause the pages to be allocated until the program is actually loaded. (For more information, see the *VAXELN Development Utilities Guide*.)

3.3.5 Scheduling in Multiprocessing Configurations

Each processor involved in a loosely or closely coupled multiprocessing configuration (see Figures 1–5, 1–7, and 1–8) executes its own copy of a VAXELN system image. Thus, the kernel uses the single-processor scheduling rules to schedule the jobs and processes on each processor participating in these configurations.

However, in a tightly coupled symmetric multiprocessing configuration (see Figure 1–6), application components running on different processors share a single copy of the VAXELN system image, including the kernel. In this case, the kernel can select a ready job to run on any available processor. Once a job begins to run on a processor, all its subprocesses run on that processor also. If the job is not eligible to run on the selected processor, the kernel reschedules the job for execution on a valid processor. The scheduling of a job for a particular processor may preempt the processor's execution of a lower-priority job.

3.4 Kernel Services for Processes and Jobs

The kernel services affecting the state of PROCESS objects are summarized in Sections 3.4.1 to 3.4.20.

3.4.1 CREATE_JOB Procedure

The CREATE_JOB procedure creates a new job that executes a specified program image. The procedure returns the new job port value. The caller can use this value to send messages to the new job. The same value can be obtained within the new job by the JOB_PORT procedure. For program images that require arguments, you can specify the arguments as strings in an optional argument list. The argument list must specify all required argument values for the specified program image.

An optional argument identifies a port to be notified of the created job's termination. If this argument is present, a *termination message* is sent to the port when the new job terminates. The termination message is the integer completion status of the created job's master process. If the argument is omitted, no message is sent.

The job's master process can return an explicit status with the EXIT procedure; if it specifies no status and completes successfully, the default status returned in the termination message is 1 (success). An unhandled exception condition causes the value of the exception to be returned.

CREATE_JOB runs a program image already built into the system (with the System Builder), or it executes program images that are loaded dynamically with the ELN$LOAD_PROGRAM procedure after the initial system is built.

3.4.2 CREATE_PROCESS Procedure

The CREATE_PROCESS procedure creates a new subprocess running the specified process block or function, returning the new PROCESS value that identifies the process. An optional list of up to 31 arguments can be passed to the created process.

An optional integer variable receives the final (exit) status of the created process. The variable must be in shared space. Such a value can be returned by the created process with the EXIT procedure. If the argument is omitted, no such status is returned. An unhandled exception condition causes the value of the exception to be returned.

3.4.3 CURRENT_PROCESS Procedure

The CURRENT_PROCESS procedure returns a PROCESS value that identifies the calling process.

3.4.4 DELETE Procedure

The DELETE procedure removes the PROCESS object from the system. When a process is deleted, if another process is waiting for its termination, that aspect of its wait condition is satisfied permanently.

When a master process is deleted, all subprocesses in the same job are deleted, along with the data and kernel objects created by processes in the job. The exit status of a deleted process is KER$_NO_STATUS.

3.4.5 DISABLE_SWITCH Procedure

The DISABLE_SWITCH procedure disables process switching for the job from which it is called. The calling process continues executing, regardless of the priorities of other processes in the job, until switching is reenabled with ENABLE_SWITCH.

If the process that calls DISABLE_SWITCH blocks and requires action from another process in the same job before it can resume, deadlock results — that is, the blocked process cannot unblock.

NOTE

> Process switching is reenabled automatically if the process calls EXIT or deletes itself.

DISABLE_SWITCH is necessary only when a process must perform an operation with assurance that it will not be preempted by other processes in the job.

3.4.6 ENABLE_SWITCH Procedure

The ENABLE_SWITCH procedure restores preemptive process scheduling, or switching, for the calling job. When process switching is enabled, the control of the CPU is given to the highest-priority process in the job that is ready to run. The procedures ENABLE_SWITCH and DISABLE_SWITCH count the number of times they are called; switching is enabled only if the number of calls to ENABLE_SWITCH is equal to the number of calls to DISABLE_SWITCH for a particular process.

3.4.7 EXIT Procedure

The EXIT procedure causes an immediate exit from the calling process. The procedure is similar to deleting the current process, except that it can optionally return an exit status to the process that created it. Process switching, if disabled by the process, is reenabled automatically, so control goes to the highest-priority process in the job that is ready to run. If the calling process is the master process, all the objects it owns, including subprocesses, are deleted; all open files are closed.

3.4.8 KER$GET_JCB Procedure

The KER$GET_JCB procedure returns a job control block (JCB) address. In a tightly coupled symmetric multiprocessing configuration — for example, the VAX 8800 multiprocessor — the procedure saves the current interrupt priority level (IPL), raises the IPL to 4 so that the job will not be switched to run on another processor, gets the JCB address, and restores the initial IPL. (In a single-processor configuration, the procedure accesses the JCB address without raising the IPL.) The returned address can then be used to read fields in the JCB.

User-mode programs in a tightly coupled multiprocessing configuration must use this procedure to access fields of the JCB. Kernel-mode programs in the same configuration can either use this procedure or perform the equivalent set of operations, including raising the IPL to 4. The ability to use this procedure in single-processor configurations, where it is not necessary to protect against a job being switched to a different processor, is provided so that the same source code can be used in all configurations without modification.

3.4.9 KER$GET_USER Procedure

The KER$GET_USER procedure returns the user identity of either the calling process or the partner process connected by a circuit to the caller's port. An optional argument specifies a port connected in a circuit; if this argument is supplied, the port must be connected in a circuit that the caller has accepted with the ACCEPT_CIRCUIT procedure. Valid information is not returned if the caller initiated the connection with CONNECT_CIRCUIT; that is, KER$GET_USER can provide information only about the object of a connection, not the subject.

Other optional arguments return the user name string and the UIC of either the calling process or the partner process. If the circuit is from a remote user, but there is no Authorization Service available in the system — that is, the **Authorization required** entry on the System Builder's Edit Network Node Characteristics Menu is No — KER$GET_USER returns 0 for the UIC parameter.

3.4.10 INITIALIZATION_DONE Procedure

The INITIALIZATION_DONE procedure informs the kernel that the calling program has completed an initialization sequence and that other programs can be created and made eligible to run. This procedure does not cause the calling job to block. The calling job continues to run until some other occurrence causes it to block.

The INITIALIZATION_DONE procedure is exclusively for programs that have the System Builder **Init required** program attribute.

NOTE

Context switching is disabled during initialization.

3.4.11 KER$NAME_OBJECT Procedure

The KER$NAME_OBJECT procedure names a specified process by creating a unique NAME object that associates a character string with the process. The procedure helps you identify the process when you use the remote debugger and other VAXELN development utilities.

This procedure is similar to the CREATE_NAME procedure that creates names for message ports (see Chapter 5), except that process names do not have the local or universal attribute that is associated with port names.

NOTE

> KER$NAME_OBJECT is used only in Pascal programs; to get the equivalent process-naming feature in C, you call KER$CREATE_NAME with a special set of arguments. See the *VAXELN C Runtime Library Reference Manual* for details.

3.4.12 KER$RAISE_PROCESS_EXCEPTION Procedure

The KER$RAISE_PROCESS_EXCEPTION procedure raises the asynchronous exception KER$_PROCESS_ATTENTION in the specified process.

3.4.13 RESUME Procedure

The RESUME procedure resumes the execution of a suspended process. A resumed process is ready to run but is not necessarily running. If the process was waiting when it was suspended, the wait is repeated when it is resumed. Asynchronous exceptions that occurred during the suspension are raised when the process runs, including the exception KER$_QUIT_SIGNAL that results from signaling the process itself.

3.4.14 Setting a Job's Processor Eligibility

A job's processor eligibility is determined when the job is ready to run based on information in the job's job control block (JCB). An application program can alter this eligibility information while executing by calling the KER$SET_JOB_ELIGIBILITY procedure. An argument supplies Boolean values that indicate job eligibility for each processor in your target configuration. TRUE means a job is eligible to run on a processor; FALSE means a job is not eligible to run on a processor. Whether the master process or a subprocess calls the procedure, the call changes the processor eligibility for the entire job. If a job's new eligibility makes the job ineligible to run on its current processor, the kernel reschedules the job for execution on a valid processor; otherwise, no rescheduling takes place.

The KER$SET_JOB_ELIGIBILITY procedure is most useful for programs that run in multiprocessor configurations. However, code that includes the procedure can run on both single-processor and multiprocessor configurations. On a single-processor system, the procedure changes the job's eligibility mask but has no other effect, even if the user argument specifies ineligibility for the single processor.

In multiprocessor configurations, jobs are initially eligible to run on any available processor. If the configuration includes a VAX 8800 multiprocessor and a device driver job calls the CREATE_DEVICE procedure, the kernel ties the job to the processor that handles the device's interrupts. This lets the driver raise the processor's IPL with a call to DISABLE_INTERRUPT to synchronize access to the device communication region. Synchronization using an elevated IPL is not possible if interrupts are being handled by the other processor.

For multiprocessor configurations that let devices interrupt any processor (such as the VAX 62*nn* multiprocessor), you can use the KER$SET_JOB_ELIGIBILITY procedure to make a user-created job eligible on a specified set of processors. (This is also true for a driver running on a VAX 8800 multiprocessor, as long as the driver does not use an elevated IPL to synchronize access to the device communication region.) However, the procedure affects only the job for which the call is made; it does not keep other jobs, including system jobs such as the debugger and drivers that Digital supplies, from running on the specified processors.

In a tightly coupled multiprocessor configuration, at least one available processor must be eligible to run the job. If the job cannot run on any of the processors that are up and running as part of the configuration, the kernel returns the status value KER$BAD_VALUE.

3.4.15 SET_JOB_PRIORITY Procedure

The SET_JOB_PRIORITY procedure resets the scheduling priority of the current job to an integer in the range 0 to 31. Priority 0 is the highest. The initial priority for a job can be set by the System Builder as part of a program description or by the ELN$LOAD_PROGRAM procedure; the default is 16. Raising job priority causes the calling job to continue execution at the higher job priority. Lowering job priority allows a ready job with higher (or equal) combined job and process priority, if there is one, to gain control of the processor; otherwise, the calling job continues execution at the lower job priority.

Jobs and processes in a VAXELN system are scheduled on a preemptive priority basis. When scheduling an idle processor or arbitrating possible job preemption, the scheduler allocates the processor to the ready job with the highest combined job and process priority. That is, the scheduler selects the job with the highest job priority or, among jobs of equally high job priority, the job with the highest-priority ready process. Preemption occurs when a process entering the ready state becomes the highest-priority ready process in its job, such that the ready job then has a higher combined job and process priority than the running job.

The scheduling scheme can be extended to allow a running job to give up control to a ready job of equal combined job and process priority, without lowering its own priority. If running job *a* issues a call to SET_JOB_PRIORITY that specifies its current priority, one of the following occurs:

- If another job, *b*, of the same combined priority is ready, job *b* is placed in the running state. The voluntarily preempted job *a* is placed in the ready queue behind remaining jobs of equal combined priority.
- If no other job of the same combined priority is ready, the running job continues in the running state.

3.4.16 SET_PROCESS_PRIORITY Procedure

The SET_PROCESS_PRIORITY procedure resets the scheduling priority of a process to an integer in the range 0 to 15. Priority 0 is the highest. The initial priority for the processes in a job can be set by the System Builder as part of a program description or by the ELN$LOAD_PROGRAM procedure; the default is 8.

When arbitrating possible process preemption within a job, the scheduler selects the process with the highest process priority. Preemption occurs within a job when a process becomes ready with higher priority than the job's current process.

The scheduling scheme can be extended to allow a running process to give up control to a ready process of equal priority within the same job, without lowering its own priority.

If process switching is enabled and process *a* issues a a call to SET_PROCESS_PRIORITY that specifies its own process value and its current priority, one of the following occurs:

- If another process, *b*, of the same priority within the same job is ready, process *b* is placed in the running state. The voluntarily preempted process *a* is placed in the ready queue behind remaining processes of equal priority within the same job.
- If no other process of the same priority within the same job is ready, the running process continues in the running state.

3.4.17 KER$SET_USER Procedure

The KER$SET_USER procedure sets the user identity of the current process. A string of up to 20 characters specifies the user name to be associated with the process. An integer supplies the user identification code (UIC) to be associated with the process.

3.4.18 SIGNAL Procedure

Signaling a process with a call to SIGNAL raises the exception KER$_QUIT_SIGNAL for that process. If the process needs to perform special operations, such as deallocating resources, before exiting, it must have established an exception handler to handle the KER$_QUIT_SIGNAL exception. If the process does not have an established exception handler or if the exception handler resignals the exception, the kernel forces the process to exit. The exception handler should resignal the exception if the job is to exit after the special operations are completed.

3.4.19 SUSPEND Procedure

The SUSPEND procedure suspends the execution of a process. If the process is waiting, as a result of a WAIT_ANY or WAIT_ALL call, it is removed immediately from the waiting state and then suspended. If the process is resumed later, the wait is repeated.

3.4.20 WAIT_ANY and WAIT_ALL Procedures

The WAIT procedures make a process wait for 0 to 250 *wait conditions* (conditions pertaining to the state of objects) to be satisfied. WAIT_ANY allows the invoking process to continue if a wait condition is satisfied; WAIT_ALL requires that all the conditions be satisfied simultaneously. A wait for a PROCESS object is satisfied when the process terminates.

Waiting causes no modification to a PROCESS object, and all waiting processes continue if their wait conditions are otherwise satisfied. Both procedures can specify a timeout argument, which defines either a time interval or absolute time after which the waiting process proceeds regardless of the states of the objects.

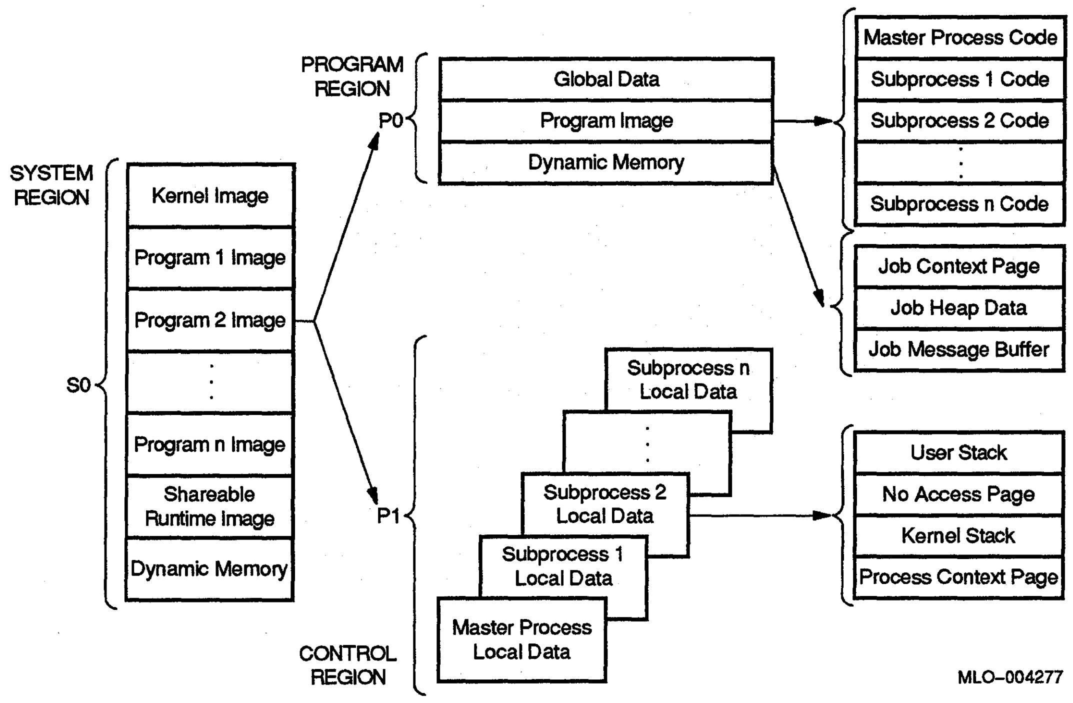
3.5 Memory Management

VAXELN uses the VAX memory management hardware to map jobs in a virtual address space. Although knowledge of VAX memory management is not essential for understanding this section, you may find it more useful if you are already familiar with VAX memory management terminology. Figure 3–4 illustrates a typical mapping.

Each job created by VAXELN executes a program image. You build program images into the system image with the System Builder or load them dynamically with the program loader. The shareable runtime library modules and kernel are not included as part of a program image but are images themselves.

When a VAXELN system is booted, the kernel maps the system image (kernel, program, and shareable runtime images) into the *S0 virtual address space* (the system region). The system region maps the system image and kernel data, as shown in Figure 3–5.

Figure 3–4: Memory Allocation



When the kernel creates a job, it generates a P0 page table and maps the job's program image, data, and message buffers into *P0 virtual address space* (the program region) as shown in Figure 3–6. If multiple jobs in a system use the same program image, the kernel makes a copy of the image's read/write data for each job and lets all jobs share the same read-only code and data.

Figure 3–5: System Region

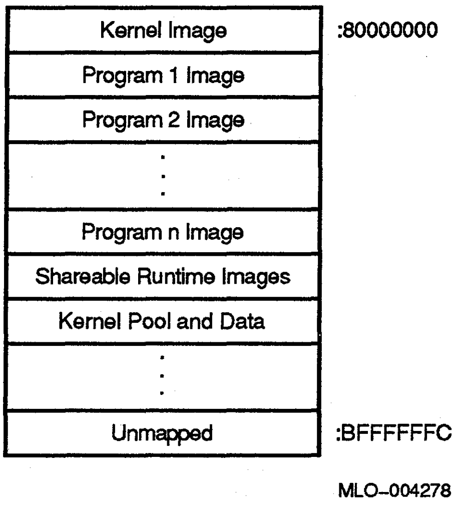
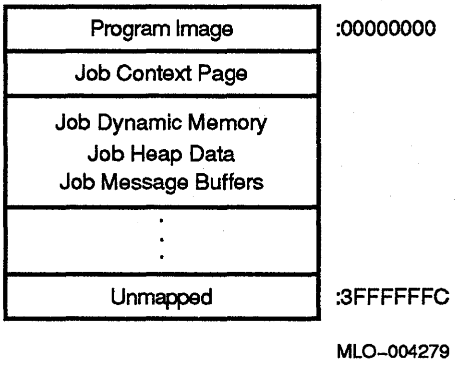


Figure 3–6: Program Region



The kernel uses P0 virtual address space for static variables and message text. The kernel makes a copy of the read/write data, although no copy is made of read-only code and data. If multiple jobs in a system run the same program, only one copy of the read-only code and data exists, with as many copies of the read/write data, message data, and heap data as jobs running the program. Since the runtime library uses heap data for many of its data structures, the kernel also maps the context of open file variables into P0 address space so the runtime libraries can use the variables for their data structures.

A job's processes share its P0 page table and P0 address space. Thus, the processes can access the same job-level data. The processes can coordinate their access to this data by using synchronization techniques. A pointer to a data item in the P0 address space can be passed to any process in the job. A pointer cannot be passed to a process in another job, since the pointer refers to a different data item in that job's P0 region.

In addition to setting up static memory mapping, the kernel manages the data associated with dynamically created processes. When the kernel creates a process, it generates a P1 page table and maps a kernel and user stack into *P1 virtual address space* (the control region). Each process in a job, including the master process, has its own pair of stacks, which store process-specific data, such as local variables and procedure call frames.

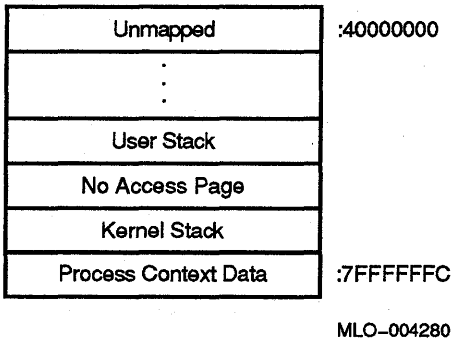
The kernel uses P1 virtual address space exclusively for dynamic memory; it does not map any of the program image. Kernel procedures and kernel mode programs use the fixed-sized kernel stack. The kernel expands the user stack as necessary, enabling programs to start out with minimal stack space. This feature saves space that can be wasted when memory is preallocated.

The kernel stack for a user-mode process occupies two pages. The stack is used by the VAXELN Kernel when executing kernel procedures and dispatching exceptions.

Kernel-mode processes have only a fixed-size kernel stack that is used by both the process and the VAXELN Kernel procedures. If the kernel-mode stack overflows, the fatal exception KER$_KERNEL_STACK is returned. When this exception is delivered, the kernel stack pointer is reset to the base of the original stack, and the previous contents of the stack are lost. The size of the kernel-mode stack is specified as a program attribute.

In addition to the stacks, the P1 address space contains process context data. This data represents context information that is used by the VAXELN Kernel, debugger, and runtime library routines. Figure 3–7 shows the P1 region of the VAX virtual address space.

Figure 3–7: Control Region



3.5.1 Managing Stack Usage

When the kernel creates a process, it generates a P1 page table and maps a kernel and user stack into P1 virtual address space (the control region). Each process in a job (including the master process) has its own pair stacks, which store process-specific data, such as local variables and procedure call frames.

For most programs, VAXELN manages stack usage sufficiently. Kernel procedures and kernel-mode programs use the fixed-size kernel stack. The kernel expands the user stack as necessary, enabling programs to start out with minimal stack space. This feature saves space that can be wasted when memory is preallocated.

You may need to control stack usage in the following cases:

- When stack usage varies widely during process execution. The kernel extends user stacks as necessary. However, since the kernel knows nothing about a program's behavior, it does not trim stacks. Thus, if the stack space allocated for a process significantly exceeds the amount of space that the process requires at a certain point during execution, space is wasted.
- When the stack size that you specify for a kernel-mode program causes stack space to be wasted. The kernel allocates the size that you specify to each process in the program's job. Again, if the stack usage for each process varies significantly, stack space may be wasted.
- When kernel stack overflows occur. Kernel stack overflows may occur because kernel stacks are not dynamically extended as are user stacks.

Your programs can control these conditions by calling the ELN$DEALLOCATE_STACK and ELN$ALLOCATE_STACK procedures. These procedures extend and contract the stacks during program execution. Use the ELN$DEALLOCATE_STACK procedure to trim a stack by a specified number of bytes, without trimming beyond the page containing the current stack pointer (SP). If the stack does not contain the specified space, the kernel trims the stack to the page in which the procedure is running. Thus, you can trim the stack to the currently needed size by specifying an overly large number.

Use the ELN$ALLOCATE_STACK procedure to verify the availability of an amount of stack space. If the stack space is not available, the procedure allocates the additional space needed to satisfy the request. This procedure is most useful for allocating stack space for kernel-mode programs that demand more stack space than was allocated when the system was built. This procedure is not as useful for user-mode programs because the kernel automatically extends the stack as needed by the process.

If a program produces an exception that indicates an invalid kernel stack, you should suspect inadequate stack size (kernel stack overflow) as a possible cause. For an example of how a kernel stack overflow can occur, consider the following situation. When a program running in kernel mode issues a call to WRITELN, the procedure's arguments (and other information) are pushed onto the kernel stack allocated for that program. The WRITELN procedure in turn calls a routine in the runtime library, which pushes yet more information onto the stack. Since kernel stacks are not automatically extended at runtime, this single call to WRITELN can cause the stack to overrun its allotted size and result in system failure.

Any kernel-mode program that calls nested subroutines can encounter kernel stack overflows. To prevent such overflows, you must allocate adequate kernel stack space for kernel-mode programs. If you suspect that kernel stack overflows are occurring, specify a larger kernel stack size in the program's description; then rebuild the system. To avoid or correct the problem at runtime, call the ELN$ALLOCATE_STACK procedure from the offending program.

3.5.2 Allocating Memory

The procedures summarized in Sections 3.5.2.1 to 3.5.2.5 allocate and free memory.

3.5.2.1 ALLOCATE_MEMORY Procedure

The ALLOCATE_MEMORY procedure allocates physical memory pages (not necessarily contiguous) into contiguous virtual address space of the job that calls it. The allocated memory can be placed at a specified virtual address or at a virtual address selected by the kernel. The procedure returns the address at which the memory is allocated.

The caller specifies the size of the needed memory in bytes, but allocation is done in units of memory pages (512-byte pages). The size is rounded up to page-sized units before the allocation. Allocation always begins on a page boundary.

If the allocation virtual address was selected by the kernel, the address will be in the P0 or shared region of the job's virtual memory. The caller can specify any virtual address, so it is possible to allocate memory in the P1 or stack region, as well as at a particular memory location in P0.

Most high-level languages provide a higher level and more controlled means of allocating and freeing dynamic memory — for instance, the Pascal NEW procedure and the C **calloc** or **malloc** functions. Use these procedures if you do not need to allocate memory at a specific location, or if you need to allocate memory in different units than a page (512 bytes). The smallest unit you can allocate with NEW is 8 bytes.

Use the ALLOCATE_MEMORY procedure for large temporary memory allocations or to allocate memory at a specific virtual or physical address. ALLOCATE_MEMORY is a low-level operation that is used by programs that need direct control of memory allocation or is used as a building block to provide a higher-level service.

The ALLOCATE_MEMORY procedure also allows a kernel-mode caller to specify the exact physical address at which to start the allocation. If you specify a physical starting address, the memory allocated is physically contiguous. This feature is intended for specialized applications, for example, multiported memory or video bitmap memory. The kernel does not restrict the use of this parameter and does not check that the value is consistent with the state of the system. Therefore, it is possible to accidentally *double map* pages of memory that are already in use.

3.5.2.2 KER$ALLOCATE_SYSTEM_REGION Procedure

The KER$ALLOCATE_SYSTEM_REGION procedure allocates memory in system (S0) address space. The memory allocated is virtually and physically contiguous, and the virtual addresses come from the system region you specify on the System Builder's System Characteristics Menu.

This procedure can be called only by programs running in kernel mode.

You might use KER$ALLOCATE_SYSTEM_REGION to map a device's I/O space control status registers (CSRs) into S0 virtual address space. Typically, the kernel maps the I/O space for a system's device CSRs into S0 address space at initialization time, and calls to CREATE_DEVICE return a pointer to the first CSR for a device. The kernel does not do this mapping for all devices. For example, the mapping is not done for devices on systems that use an integral bus. Device drivers for such devices can map the registers into S0 address space by specifying the appropriate physical address and size in a call to KER$ALLOCATE_SYSTEM_REGION.

When you finish using an area of S0 space, use KER$FREE_SYSTEM_REGION to free it; deleting a process or job does not free S0 space.

3.5.2.3 FREE_MEMORY Procedure

The FREE_MEMORY procedure frees the physical memory pages that are mapped to particular virtual addresses in the caller's address space. The caller specifies a base virtual address and a size in bytes. The procedure frees memory pages in the inclusive range from the base to the top.

NOTE

> Be careful when you free memory that was not explicitly allocated by the caller, since it is difficult to determine the use of the virtual address range. For instance, deleting the process's stack can have unpredictable results.

Dynamically allocated memory is normally freed with the language-specific runtime library procedures provided by Pascal and C, that is, the Pascal DISPOSE procedure and the C **free** or **cfree** functions. Pointers to the freed memory become invalid.

3.5.2.4 KER$FREE_SYSTEM_REGION Procedure

The KER$FREE_SYSTEM_REGION procedure frees memory in S0 address space that was previously allocated with the KER$ALLOCATE_SYSTEM_REGION procedure. The memory is freed from the system region you specify on the System Builder's System Characteristics Menu.

This procedure can be called only by programs running in kernel mode.

When you are finished using an area of S0 space, use KER$FREE_SYSTEM_REGION to free it; deleting a process or job does not free S0 space.

3.5.2.5 KER$MEMORY_SIZE Procedure

The KER$MEMORY_SIZE procedure scans the kernel memory data base and returns the initial main memory, the current free memory, and the current largest free memory block size (in 512-byte pages). The largest free block size is the size of the largest physically contiguous block of free memory. This value is useful if you need to create large MESSAGE or AREA objects, because these objects require contiguous memory for their data buffers.

While the KER$MEMORY_SIZE procedure performs the memory scan, other kernel operations are stopped; therefore, call this procedure only when necessary.

3.5.3 Loading VAXELN System Images onto KA800 Processors

One way of setting up a closely coupled symmetric multiprocessing environment is to use the ELN$LOAD_KA800_PROCESSOR procedure. By calling this procedure, a VAXELN application program can dynamically load VAXELN system images from a VAX 8*nnn* primary processor into the memory of KA800 processors attached to the primary processor's VAXBI bus. The ELN$LOAD_KA800_PROCESSOR procedure provides a runtime interface to the KA800 loader, ELN:KA800_LOADER.EXE. At runtime, the loader waits on a port for load requests.

To use the ELN$LOAD_KA800_PROCESSOR procedure, you must build the KA800 loader's program image into the system that is to run on the primary processor. You must also include modules from the runtime libraries, as appropriate for the programming language you are using.

A call to the ELN$LOAD_KA800_PROCESSOR procedure must specify the VAXBI number and adapter number of the processor into which the system is to be loaded, the file specification of the system image to be loaded, and a variable that receives the load status. An optional argument lets you specify whether the kernel debugger is to be invoked when the system image is loaded. If you specify TRUE for the debugger argument, you must build the local debugger into the system image from which the ELN$LOAD_KA800_PROCESSOR call is made.

The system image that is being loaded into a KA800 processor can reside on the primary processor or on a remote node. If you specify a system image file that resides on a remote node, you must identify the node in the specification as follows:

area.node_number

The following program calls ELN$LOAD_KA800_PROCESSOR to load the remote system image 1.10::RTDISK:[CCSMP_APP]CCSMP.SYS into the memory of a KA800 processor with local debugging enabled.

```
MODULE load_ka800;

INCLUDE $KA800_LOAD_UTILITY;

PROGRAM load_sys(INPUT,OUTPUT);

VAR
  load_stat : KA800_CODE;

BEGIN
  ELN$LOAD_KA800_PROCESSOR(BI_NUMBER := 0,
                           ADAPTER_NUMBER := 7,
                           FILE_SPEC := '1.10::RTDISK:[CCSMP_APP]CCSMP.SYS'
                           LOAD_STATUS := load_stat,
                           KDEBUG := TRUE);
END;
END.
```

You can use the ELN$LOAD_KA800_PROCESSOR procedure to reboot a previously booted KA800 processor.

You can also load a system image into the memory of a KA800 processor by entering a configuration file for the **Argument(s)** entry on the KA800 loader's Program Description Menu. *VAXELN Development Utilities Guide* explains the configuration file and how to load and boot a KA800 processor in a closely coupled symmetric multiprocessing configuration.

Chapter 4
Synchronization

In addition to performing scheduling and memory management tasks, the kernel coordinates the operations on kernel data structures. One category of such operations is *process synchronization*. Process synchronization is a mechanism for coordinating the concurrent execution of two or more processes. Using kernel procedures you can synchronize processes in the same job or processes in different jobs.

You must synchronize processes when they share a resource, depend on the completion of another process's execution, or wait for an external event to occur. The ability of a process to gain exclusive access to a shared resource is called *mutual exclusion*. The ability of a process to coordinate its activities with other processes is called *event response*.

To attain controlled access to limited resource units or coordinate event response, processes wait for one or more conditions to exist by calling the WAIT_ALL or WAIT_ANY kernel procedure. These procedures provide a method by which processes wait and define the condition under which processes can proceed again.

Section 4.1 provides a general overview of how to synchronize process execution. The rest of this chapter explains how to use VAXELN Kernel procedures to synchronize processes based on the following:

- A specified time, Section 4.2
- Process completion, Section 4.3
- Signaling of a semaphore or mutex, Section 4.4
- The occurrence of an event, Section 4.5

4.1 Synchronizing Process Execution

You synchronize execution of an application's processes by doing the following:

- Creating AREA, DEVICE, EVENT, PORT, PROCESS, and SEMAPHORE kernel objects for which the processes can wait
- Using the WAIT_ANY or WAIT_ALL kernel procedure to make processes wait for the kernel objects
- Defining the way processes are to resume after waiting

The WAIT_ANY and WAIT_ALL procedures accept a list of up to 250 AREA, DEVICE, EVENT, PORT, PROCESS, or SEMAPHORE values (including various combinations of object types). When using the WAIT_ANY procedure, the calling process waits until any one of the specified objects is signaled. When using the WAIT_ALL procedure, the calling process waits until all the specified conditions are satisfied simultaneously. When an object is signaled and all other specified conditions are satisfied, the wait is otherwise satisfied. Once a wait is satisfied (or otherwise satisfied), the process returns to the ready state and can continue executing. An optional result argument receives the number of the argument that satisfied the wait for a call to WAIT_ANY. You can also supply a timeout argument, which defines either a time interval or absolute time after which the waiting process can proceed, regardless of the states of specified objects.

NOTE

When you specify more than four objects in calls to WAIT_ALL and WAIT_ANY, you should account for the following additional overhead that is incurred:

- Additional pool blocks are required to wait for the specified objects. When creating a process, the kernel allocates one pool block for the control structures that allow the process to wait for four objects. For example, if you specify five to eight objects, the kernel allocates an additional pool block to support the wait. If you specify nine objects, the kernel permanently allocates two additional pool blocks to the process. The second pool block could support waits for nine to twelve objects.

    - Additional time needed to process waits. As the number of objects increases, the time required to process, test, and satisfy a wait increases.

    Specifying a large number of objects may also affect the process latency of driver processes that run at high job priorities. Such processes require quick response time to device interrupts. For more information, see Chapter 6.

Each call to WAIT_ANY or WAIT_ALL causes the calling process to wait for a resource or event as follows:

- Waiting on an AREA, EVENT, or SEMAPHORE object means waiting for the object to be signaled.
- Waiting on a DEVICE object means waiting for the connected interrupt to be signaled by an interrupt service routine (ISR).
- Waiting on a PORT object means waiting for a message to arrive at that port.
- Waiting on a PROCESS object means waiting for the identified process to terminate.

Wait operations affect the kernel objects and the processes waiting on those objects in the following ways:

- Satisfying a wait on a SEMAPHORE object causes the kernel to decrement the semaphore count. At most, one process continues when a semaphore is signaled.
- Satisfying a wait on an EVENT, PORT, or PROCESS object causes no modification to the object, and all waiting processes continue if their wait conditions are satisfied.
- Satisfying a wait on an AREA object depends on whether the area is associated with an event or semaphore. If the area is associated with an event, the object is not modified and all waiting processes continue if their wait conditions are satisfied. If the area is associated with a semaphore, the kernel decrements the semaphore count and at most, one process continues. Areas associated with events provide a mechanism for interjob event synchronization.
- Satisfying a wait on a DEVICE object causes the object to be cleared if the wait is satisfied by an ISR signaling the object. Only one process continues as a result of the action of an ISR.

The WAIT procedures return immediately with an error if one of the argument objects does not exist or is deleted. Both procedures also return immediately if the necessary conditions were satisfied before the call was made.

Since the WAIT_ALL procedure waits for a number of conditions to be simultaneously satisfied, *deadlock* cannot occur. Deadlock occurs when two or more processes wait for the same set of resources, each holding onto some resources while waiting for others to become available, such that no process can get all the resources it needs to continue. Since WAIT_ALL does not lock up some resources while waiting for others to become available, deadlock is not a problem, provided that all the conditions (events, semaphores, and so forth) are known and are listed in a single call. WAIT_ALL is also a more efficient way to wait for two or more objects, which need to be satisfied simultaneously, than using multiple calls to WAIT_ANY.

To program process synchronization, you can use the WAIT_ALL and WAIT_ANY procedures along with the following routines:

Routine	**Description**
CLEAR_EVENT	Sets the state of an event or an area's event to EVENT$CLEARED.
CREATE_AREA	Creates a new area or maps an existing area of memory into the calling job's P0 virtual address space and associates the area with a binary semaphore.
CREATE_AREA_EVENT	Creates a new area or maps an existing area of memory into the calling job's P0 virtual address space and associates the area with an event.
CREATE_AREA_SEMAPHORE	Creates a new area or maps an existing area of memory into the calling job's P0 virtual address space and associates the area with a semaphore.
CREATE_EVENT	Creates an event.
CREATE_MUTEX	Creates a mutex.
CREATE_PROCESS	Creates a process.
CREATE_SEMAPHORE	Creates a semaphore.

Routine	Description
CURRENT_PROCESS	Gets the identifier for the current process.
DELETE	Deletes an area, event, semaphore, or process.
DELETE_MUTEX	Deletes a mutex.
GET_TIME	Returns a processor's system time.
KER$GET_UPTIME	Returns a time interval indicating the time that has elapsed since system initialization.
LOCK_MUTEX	Locks a mutex.
SET_TIME	Sets a processor's system time.
SIGNAL	Signals an event, a semaphore, or an area's associated event or semaphore.
UNLOCK_MUTEX	Unlocks a mutex.

The rest of this chapter explains how to use the preceding routines and WAIT_ALL and WAIT_ANY to synchronize process execution using the following:

- Time values, Section 4.2
- Process completion, Section 4.3
- Semaphores, Section 4.4
- Events, Section 4.5

For descriptions of the routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

AREA, MESSAGE, and PORT objects are used for programming process and job communication, and DEVICE objects are used for programming application device handling. For information about using areas, ports, and the wait procedures to program process and job communication, see Chapter 5. Chapter 6 explains how to use device objects and the wait procedures to program VAXELN device handlers.

4.2 Using Time Values to Synchronize Process Execution

You can synchronize process execution by waiting for a specified date and time to occur or for a time interval to elapse. An application can wait on a time value in addition to or instead of kernel objects. If you synchronize processes using time values, you may need to get and set the system time. Section 4.2.1 explains how to synchronize processes by waiting on time and Section 4.2.2 explains how to retrieve and set the system time.

4.2.1 Waiting on Time

To wait on time, a process must issue a call to WAIT_ALL or WAIT_ANY that specifies a signed, 64-bit time value. A time value can be an absolute time (a specific date and time) that indicates when the process can continue or a time interval relative to the current system time that indicates how long the process must wait before continuing.

You specify an absolute time in the following format:

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

The following table defines the absolute time value components:

Component	Meaning	Value Range
dd	Day of the month	1 to 31
mmm	Month	JAN to DEC
yyyy	Year	1858 to 9999
hh	Hours	0 to 23
mm	Minutes	0 to 59
ss	Seconds	0 to 59
cc	Hundredths of a second	0 to 99

You specify a time interval as follows:

'dddd hh:mm:ss.cc'

The following table defines the time interval value components:

Component	**Meaning**	**Value Range**
dddd	Days	0 to 9999
hh	Hours	0 to 23
mm	Minutes	0 to 59
ss	Seconds	0 to 59
cc	Hundredths of a second	0 to 99

By convention, positive time values represent absolute time; negative time values represent time intervals.

If you do not specify a time value, the calling process unblocks only when the specified wait object conditions are otherwise satisfied.

You can issue a conditional wait that will not block, by specifying a timeout value of 0. Specifying 0 ensures that the WAIT procedures check for satisfied conditions and then return immediately. If the returned wait result is 0, the wait condition specified by the objects was not satisfied.

The kernel expects and returns time values in 64-bit time value format. Thus, applications must convert absolute time and time interval format strings to and from 64-bit time value format, as appropriate. Runtime routines for converting time value formats are available in Pascal, C, and FORTRAN. See the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual* for more information.

The VAXELN Pascal, C, and FORTRAN language runtime libraries provide routines for dealing with time values conveniently. For example, you can use the routines to convert a time value to an ASCII string for printing, or an ASCII string in an absolute or interval time format to a time value for time value operations.

4.2.2 Retrieving and Setting the System Time

Before specifying an absolute time value in a call to WAIT_ANY or WAIT_ALL, you should set the system time. The system time is absolute and is maintained by the VAXELN Kernel as a 64-bit binary number. The system time is interpreted as the number of 100-nanosecond intervals since the base time, 00:00:00.00, November 17, 1858. The kernel uses the system's interval timer to maintain the system time. Thus, the system time is in effect for all jobs running on that system.

You can set and get a system's time by using the SET_TIME, GET_TIME, and KER$GET_UPTIME procedures.

A processor's system time is not necessarily preserved across power failures and is not set to a default value by the kernel or other system software. Thus, you should use SET_TIME to initialize the system time in an initialization job (see the *VAXELN Development Utilities Guide*) and in a handler for the KER$_POWER_SIGNAL exception. For example:

```
SET_TIME(TIME_VALUE('10-MAR-1990 00:00:60'));
```

You can also set the system time with the debugger and ECL command SET TIME. You can display the system time using the debugger and ECL command SHOW TIME. For information about the SET TIME and SHOW TIME commands, see the *VAXELN Development Utilities Guide*.

If you set the system time while a wait that specifies an absolute timeout value is pending, one of the following effects occurs:

- If you set the system time back, the timeout period is increased by the amount you set the time back.
- If you set the system time forward to a time that does not exceed the absolute timeout value, the waiting time is reduced to the difference between the new system time and the original timeout value.
- If you set the system time forward to a time that exceeds the absolute timeout value, the timeout occurs immediately.

If a wait with an interval timeout value is pending, resetting the system time does not change the remaining timeout period.

To retrieve the system time, use the GET_TIME or KER$GET_UPTIME procedure. GET_TIME returns the current system time. KER$GET_UPTIME returns a time interval indicating the time that has elapsed since system initialization. The negative value representing the time interval decreases continuously regardless of system time resets.

The following example illustrates the use of SET_TIME, GET_TIME, and KER$GET_UPTIME:

```
MODULE time;

INCLUDE $KERNEL;

PROGRAM time;

VAR
  set_time_value,
  get_time_value,
  uptime_value : LARGE_INTEGER;
  time_str : VARYING_STRING(23);

BEGIN

  WRITE('Enter date and time (dd-mmm-yyyy hh:mm:ss.cc): ');
  READLN(time_str);
  set_time_value := TIME_VALUE(time_str);
  SET_TIME(set_time_value);           { Set the system time. }

  GET_TIME(get_time_value);           { Get the current system time. }
  time_str := TIME_STRING(get_time_value);
  WRITELN('The time is now ', time_str);

  KER$GET_UPTIME(,uptime_value);      { Get the elapsed system time. }
  time_str := TIME_STRING(uptime_value);
  WRITELN('The system uptime is ', time_str);

END;
END.
```

4.3 Synchronizing Process Execution Based on Process Completion

Waiting for a process means waiting until the process has terminated. When one process waits for another, the second process is usually created by the waiting process, which needs it to complete some task before the waiting process can continue.

The actions of the two processes are synchronized in the following way:

1. The first process (Process A) creates the process it must wait for (Process B).
2. Process A then calls WAIT_ALL or WAIT_ANY to wait for Process B.
3. Process B executes its process block until it terminates.
4. The termination of Process B satisfies the wait condition for Process A.
5. Process A continues its execution with the line of code following its call to WAIT_ALL or WAIT_ANY.

To wait for a process, the process that wishes to wait (Process A) must specify the PROCESS variable associated with the process to be waited for (Process B) in a call to the WAIT_ALL or WAIT_ANY procedure. When Process A creates Process B and then waits for it, the same PROCESS variable is used in both the CREATE_PROCESS and the WAIT calls. When Process A does not create Process B, Process B's PROCESS variable must be globally accessible or must have been passed to Process A as an argument when Process A was created or in a message.

The CREATE_PROCESS procedure has an optional EXIT parameter that allows the creating process to receive an exit status from the created process when the latter exits, if it terminates its execution with the EXIT procedure. The created process can supply the EXIT_STATUS value to indicate whether it has accomplished its task successfully. This EXIT_STATUS value is returned to the creating process in the variable passed as the EXIT argument to CREATE_PROCESS. When its wait has been satisfied by the termination of the created process, the creating process can check the EXIT status and take the appropriate action, based on the success or failure of the process it created.

4.4 Using Semaphores to Synchronize Process Execution

Semaphores act as gates that control access to resources such as global variables, hardware resources, or the CPU. A semaphore maintains a count of the available units of a resource, such as the number of disk drives available, the number of gates available at an airport, or, for a railroad semaphore, the number of tracks (0 or 1) available to a train going in a particular direction.

A semaphore's count value changes as processes wait on and signal the semaphore or area. As the value changes, it controls the execution of waiting processes, letting at most one process enter the ready state when a semaphore or area associated with a semaphore is signaled. When a process signals a semaphore or area associated with a semaphore, the semaphore count is incremented. When a process waits on a semaphore or an area associated with a semaphore, the process waits until the semaphore count is greater than 0. When the count exceeds 0 (and, for WAIT_ALL, if all other wait conditions are satisfied) the process unblocks. When the kernel selects the process to execute, the procedure call returns and the process proceeds. If a wait is satisfied when the semaphore is signaled, the kernel decrements the semaphore count.

The following sections explain how to do the following:

- Create semaphores, Section 4.4.1
- Wait on and signal semaphores, Section 4.4.2
- Delete semaphores, Section 4.4.3
- Use mutexes, Section 4.4.4

4.4.1 Creating Semaphores

An application creates and initializes a semaphore with a call to CREATE_SEMAPHORE. A call to CREATE_SEMAPHORE must specify initial and maximum count integer values.

The kernel also creates a semaphore when an application calls CREATE_AREA or CREATE_AREA_SEMAPHORE. When an application calls one of these procedures, the kernel creates an area and an associated semaphore. Like calls to CREATE_SEMAPHORE, calls to CREATE_AREA_SEMAPHORE must specify initial and maximum count integer values. In the case of CREATE_AREA, the kernel automatically initializes the initial and maximum count values to 1.

Depending on the maximum count value that you specify, semaphores are either binary or counting. A *binary semaphore* has a maximum count of 1. You use a binary semaphore to guard a single item — often, a shared variable — from access by more than one process. The binary semaphore acts as a gate, letting only one process at a time get through to the resource behind it. When you signal a binary semaphore, the gate opens for one process and then closes.

A semaphore that has a maximum count greater than 1 is a *counting semaphore*. A counting semaphore is like a gate that lets multiple processes through or a meter that keeps count of a finite resource's available units.

In both cases, the initial semaphore count determines the initial disposition of processes that issue a wait for the semaphore or area, independent of any signaling processes.

A SEMAPHORE value created by a call to CREATE_SEMAPHORE can be used only within the job that creates it. The value identifies the same semaphore throughout the job. Multiple processes in the job can use the semaphore by sharing a variable or by passing the SEMAPHORE value as a process argument.

SEMAPHORE values that the kernel associates with areas are valid in different jobs running on the same node. Thus, an application can use such an area to synchronize job execution. For information about using areas associated with semaphores, see Section 5.4.

4.4.2 Waiting On and Signaling Semaphores

A process that wants to use a controlled resource waits on the semaphore or area by calling WAIT_ALL or WAIT_ANY. If the semaphore count is greater than 0, the count is decremented, and the process enters the ready state. If the count is 0, the process waits until another process signals the semaphore or area. If several processes wait for the same semaphore, the kernel places them in a queue in the order in which they call WAIT_ANY or WAIT_ALL.

A process signals a semaphore or area when it no longer requires a resource. The SIGNAL procedure increments the semaphore count and, at most, one process unblocks if the wait is otherwise satisfied. If a process unblocks, the count is decremented. Thus, a semaphore's maximum count represents the available units of the resource being controlled.

An application that needs to meter access to a 10-unit disk driver is an example of an application that might use a counting semaphore. The following Pascal example creates such a semaphore, initializes the semaphore such that all units are available initially, and places the SEMAPHORE value in the variable *unit_available*:

```
VAR
  UNIT_AVAILABLE: SEMAPHORE;
BEGIN
  CREATE_SEMAPHORE(UNIT_AVAILABLE,10,10);
```

After a process creates the semaphore, other processes needing disk drives wait on the semaphore by specifying *unit_available* in a call to WAIT_ANY or WAIT_ALL. Because the initial count is 10, the first 10 processes that wait on the semaphore continue immediately, and each time a process continues, the kernel decrements the count. When the count reaches 0 (assuming no process has released its drive in the meantime), the kernel places all processes that wait on the semaphore in the waiting state. These processes remain in the waiting state until a process releases its drive.

The semaphore uses its count to meter disk drive availability. When a process is through using its drive, it can return the drive to the pool of available drives by signaling *unit_available*. The process at the head of the semaphore's queue can then access a drive and continue. If the queue is empty, the next process to wait on the semaphore accesses the free drive.

The following scenario illustrates the use of a binary semaphore to guard a shared data base:

- A central data base is shared by a family of transaction-processing processes. When the master process begins execution, it creates a semaphore with maximum and initial counts equal to 1.
- The master process then creates worker subprocesses, as the need arises, to handle incoming data base inquiries.
- Each subprocess waits on the semaphore before accessing the shared data base and signals the semaphore when it is finished.

Since the maximum value of the binary semaphore is 1, only one process can access the data base at a given time. Other processes must wait until the current worker signals the semaphore. When the semaphore is signaled, the next process can access the data base.

4.4.3 Deleting Semaphores

A process can delete a semaphore by specifying its value in a call to the DELETE procedure. When a process deletes a semaphore, the kernel unblocks all processes that are waiting on that semaphore and returns the completion status KER$_BAD_VALUE. When a process deletes an area that has an associated semaphore, the kernel removes all processes waiting on the area in the same job from the waiting state and returns KER$_BAD_VALUE.

4.4.4 Using Mutexes to Optimize Waiting and Signaling Operations

You can improve the performance of binary semaphore waiting and signaling operations by using a *mutex*. (Mutex is an abbreviation for mutual exclusion semaphore.) Mutexes allow you to achieve the same effect as a binary semaphore without calling a kernel service unless contention exists.

You can create, delete, lock, and unlock mutexes by using mutex procedures. To use these procedures, you must include one of the following modules:

Language	**Module**
VAXELN Pascal	$MUTEX from RTLOBJECT.OLB
C	**$mutex** from VAXELNC.TLB
FORTRAN	"ELN$:FORTRAN_DEFS.FOR"

You create and initialize a mutex by specifying a variable of type MUTEX in a call to the ELN$CREATE_MUTEX procedure. The procedure initializes the mutex counter to −1, creates a SEMAPHORE object with an initial count of 0 and maximum count of 1, and stores the semaphore's identifying value in one of the mutex variable's fields. In addition to specifying a variable for the mutex, you can specify a variable that receives the completion status.

Once you create a mutex, you can use the ELN$LOCK_MUTEX and ELN$UNLOCK_MUTEX procedures to lock and unlock the mutex. ELN$LOCK_MUTEX provides the calling process with exclusive access to a shared resource. Generally, when a process locks a mutex, the process does not need to issue a wait before accessing the resource. The kernel issues a wait only if the mutex is already locked.

A process can relinquish exclusive access to a shared resource by calling ELN$UNLOCK_MUTEX. This procedure signals the semaphore if a process is waiting for access.

If a binary semaphore is open (count = 1) and the semaphore is signaled, a count overflow error occurs. In contrast, the locking and unlocking of mutexes is not protected by this exception-raising mechanism. Under certain circumstances, in which two or more processes contend for a mutex, unlocking an already unlocked mutex can cause the calling process to block indefinitely. Therefore, when using mutexes, you should adhere to the following guidelines:

- You must make sure the first operation on an initialized mutex is a lock operation.
- You must pair lock and unlock operations within the code of the processes using the mutex.

When a process is finished using a mutex, it can delete it by calling ELN$DELETE_MUTEX. This procedure deletes the mutex and the semaphore associated with it. A call to ELN$DELETE_MUTEX must specify the mutex you are deleting. You can also specify a variable to receive the completion status.

4.5 Using Events to Synchronize Process Execution

An EVENT object represents the occurrence of an application-defined, realtime event. An event can be in one of two states: signaled or cleared. You can specify an event's initial state when you create the event.

The following sections explain how to do the following:

- Create events, Section 4.5.1
- Wait on, signal, and clear events, Section 4.5.2
- Delete events, Section 4.5.3

4.5.1 Creating Events

An application creates and initializes an event with a call to CREATE_EVENT. A call to CREATE_EVENT must specify the event's initial state: EVENT$CLEARED or EVENT$SIGNALED.

The kernel also creates an event when an application calls CREATE_AREA_EVENT. When an application calls this procedure, the kernel creates an area and an associated event. Like calls to CREATE_EVENT, calls to CREATE_AREA_EVENT must specify the event's initial state — EVENT$CLEARED or EVENT$SIGNALED.

4.5.2 Waiting On, Signaling, and Clearing Events

An event's state changes as processes clear and signal the event or area. Processes wait on an event by specifying the event's value in calls to WAIT_ANY or WAIT_ALL. If the event is in a signaled state, the processes continue immediately. Otherwise, the processes wait for another process to signal the event or area. Once the event is signaled, all waiting processes unblock if their wait conditions are otherwise satisfied.

An EVENT value created by a call to CREATE_EVENT is valid only within the job that creates it. The value identifies the same event throughout the job. Multiple processes in the job can use the event by sharing a variable or by passing the EVENT value as a process argument.

EVENT values that the kernel associates with areas are valid in different jobs running on the same node. Thus, an application can use such an area to synchronize job execution. For information about using areas that are associated with events, see Section 5.4.

The particular realtime event represented by an EVENT object is application-specific. The conditions under which the EVENT object is signaled define its relationship to a realtime, real-world event. To the VAXELN Kernel, however, the EVENT object has only the properties signaled and cleared; nothing intrinsic in the EVENT object determines which process can signal it or what the signal means to waiting processes. The application designer must ensure that event signal and wait operations occur in a manner appropriate to the event's real-world meaning. For example, the following Pascal code fragment creates the EVENT object *lights_on*, and after determining that the lights are on, signals the event for processes that may be waiting for it.

```
VAR
  lights_on: EVENT;

BEGIN
  CREATE_EVENT(lights_on, EVENT$CLEARED);
    .
    .
    .
  /* Check whether lights are on. */
    .
    .
    .
  SIGNAL(lights_on);
```

The satisfied wait condition has no effect on the event's state. Once an event is signaled, it remains signaled until a process clears it with a call to the CLEAR_EVENT procedure. Processes waiting on the event have that part of their wait condition satisfied immediately.

The CLEAR_EVENT procedure sets the state of an event or an area's event to EVENT$CLEARED.

The following scenario illustrates the use of events:

- A family of processes executes a series of steps that controls the operation of a chemical plant. One master process controls the sequencing of several other worker subprocesses. Each subprocess executes independently until it completes a step, at which time it must synchronize its execution with the master process.
- The master process is the first to execute, and it creates two events with initial states of *cleared*, that is, not signaled. The master process then creates each subprocess and gives it a control step to perform.
- The subprocesses race each other to complete their assigned work, and as each one finishes, it executes a WAIT procedure, specifying the first of the two events.
- When the master process determines it is time to perform the next control step, the master process signals the first event, which causes all the waiting subprocesses to continue.
- As the subprocesses finish the second control step, they again execute a WAIT procedure, but this time they specify the second event.

- After the appropriate amount of time, the master process clears the first event and then signals the second event. The worker subprocesses again continue, and so it goes until the work is finished.

4.5.3 Deleting Events

A process can delete an event by specifying its value in a call to the DELETE procedure. When a process deletes an event, the kernel unblocks all processes that are waiting on that event and returns the completion status KER$_BAD_VALUE. When a process deletes an area that has an associated event, the kernel unblocks all processes waiting on the area in the same job and returns KER$_BAD_VALUE.

Chapter 5
Communication

Processes and jobs exchange information by applying interprocess and interjob communication techniques. Processes in the same job communicate by using module-level variables and queues. Jobs communicate by passing messages and sharing areas of memory. Message-passing allows jobs to communicate whether or not they execute on the same node; sharing memory areas restricts communication to jobs executing on the same node.

This chapter explains how to do the following:

- Share module-level data, Section 5.1
- Share packets of data using queues, Section 5.2
- Pass messages, Section 5.3
- Share memory areas, Section 5.4

5.1 Sharing Module-Level Data

A job's processes can communicate by sharing data. Most data is potentially shareable. However, a routine's local variables and value parameters are not shareable; the kernel stores this data in process-specific P1 virtual address space. Thus, the addresses of such data are meaningful only within the process allocating the data.

The processes in a job can share module-level data that you declare outside routines: constants, variables, procedures, functions, and process blocks. The kernel makes this data available to all processes in a job by storing the data in the job's P0 virtual address space. Since concurrently executing processes compete for the global data, you control access by using semaphores and mutexes.

Processes can share the following entities by name (more than one process can refer to the variable by its name):

- Pascal outer-level variables
- C variables declared with the attribute **extern**, **globaldef**, **globalref**, or **static**
- FORTRAN global commons

Sharing can also be accomplished with pointers and, in Pascal, with process block variable parameters.

Sharing constant data, including variables declared with the Pascal attributes READONLY or VALUE or the C attribute **readonly**, presents no programming problems. However, the sharing of data that is modified by one or more processes must be carefully managed to prevent unpredictable program behavior.

In Pascal, you can use the following constructs to process data shared by processes within a job:

- The READ_REGISTER and WRITE_REGISTER routines. (They are not restricted to operations on actual device registers.)
- The procedures INSERT_ENTRY and REMOVE_ENTRY, when used on the head and tail entries of a queue.
- The ADD_INTERLOCKED function.

In C, the **add_interlocked** function is the only atomic (indivisible) function you can use safely to process data shared by a job's processes. (For information about sharing packets of data in C, see Section 5.2.)

If you perform more complicated operations on shared data, you must synchronize access to the data. While one process is executing code that can modify the data, no other process can execute code that has access to the data. The synchronization must be done with kernel procedures or with the mutex routines (which call the kernel procedures when necessary).

NOTE

Failure to synchronize access to shared data results in unpredictable program behavior. A program that works on one processor model can fail on another; a change to the VAXELN system might cause program failure.

Additional guidelines regarding shared data follow:

- Dynamic variables. Data allocated by the Pascal NEW procedure or the C **calloc**, **malloc**, or **realloc** function can be shared. The Pascal NEW and DISPOSE procedures and the C **calloc**, **malloc**, **realloc**, **free**, and **cfree** functions operate on single data items.
- File variables and file pointer variables. Pascal file variables and C file pointer variables (and the associated internal-file data structures) are subject to the same rules as other data. Most operations that use these variables are modify operations. Failure to synchronize access to a file can result in scrambled input or output data or in a runtime error (in Pascal if the runtime routines detect simultaneous access).
- Initialization of shared data. You should initialize shared Pascal outer-level variables, C external variables, and FORTRAN global commons in the master process before creating subprocesses. Otherwise, you might forget that the initialization operation must be synchronized. For example, you must initialize a mutex variable with a call to ELN$CREATE_MUTEX before you lock or unlock the mutex.
- Record locking. Programs that use shared data often must protect data that is more complicated than single variables. For example, if multiple processes are updating records in a File Service file, they must synchronize access to the shared data and protect (or lock) records in the file. Otherwise, two read/write sequences on the same record can become interleaved.
- Shared messages. A message and its associated pointer can be manipulated by more than one process in a job, but the operations must be properly synchronized. For example, if process A deletes a message while process B is preparing to send it, the program may produce unpredictable results, such as the following:
  - Process B might receive the status KER$_BAD_VALUE from the SEND procedure because the message value is no longer valid.
  - Process B might incur an exception when it tries to access the message's buffer.
  - Process B might access new, unrelated data by using the address of the original message buffer.
- Communication regions. The communication region of an interrupt service routine (ISR) is shared between the ISR and the device driver processes. The program logic of the device driver must ensure that nonatomic operations are synchronized.

- Device registers. Device registers are not shared data in the sense used in the preceding guidelines. However, an ISR and device driver processes may need to synchronize access to the registers. In some cases, the registers symbolize the responses of a device to events that occur on a bus, such as read and write requests. The only routines you can use to ensure predictable operations on device registers are the Pascal READ_REGISTER and WRITE_REGISTER routines, the C **read_register** and **write_register** routines, or the FORTRAN ELN$READ_REGISTER and ELN$WRITE_REGISTER routines.

5.2 Sharing Packets of Data Using Queues

In addition to sharing global data, a job's processes can communicate by using *queues*. Queues provide an efficient, highly structured means for a job's processes to exchange large packets of information. This section discusses the use of absolute queues — queues that use links that contain the absolute address of an entry to which it points. If you are programming in C, you have the option of using self-relative queues — queues that use links that contain a displacement from the present queue entry. Self-relative queues let two separate processes address the same queue, with each process able to treat the queue as residing at a different location in its virtual address space. For information about using self-relative queues, see the *VAXELN C Runtime Library Reference Manual.*

VAXELN provides the predeclared data types and procedures you need to create and maintain queue structures. The procedures for inserting and removing queue entries use VAX machine instructions specifically designed to synchronize queue operations automatically. Thus, two processes can access a queue simultaneously: one can insert an entry while the other removes an entry.

Use the queue data types (QUEUE_ENTRY and QUEUE_POSITION) and procedures (START_QUEUE, INSERT_ENTRY, and REMOVE_ENTRY) to pass data messages between two or more processes within a job. Using queues this way is more efficient within a job than using the SEND and RECEIVE procedures. Although you can use SEND and RECEIVE to send messages between processes in the same job, they are better suited to passing messages between jobs on the same or different systems in a network (see Section 5.3).

Typically, you use a semaphore with each queue to signal the transition of the queue from an empty state to a nonempty state. The INSERT_ENTRY procedure and the INSQUE instruction give information that allows you to synchronize queue operations. Therefore, the queue and the semaphore can work together to maintain lists and synchronize and schedule processes.

Example 5–1 shows how you can use queues as a structured and efficient means of communicating between processes. The module consists of an initialization procedure and two process blocks. The procedure initializes the queues *free_list* and *done_list*; creates the semaphores *free_list_has_entry* and *done_list_has_entry* for metering the content of each queue; and fills the *free_list* queue with entries.

The process blocks, *producer* and *consumer*, communicate by inserting entries onto and removing entries from the two queues. The processes *producer_process_1* and *producer_process_2* share the *producer* process block, which removes available entries from the *free_list* queue, fills the entries with data, and inserts the filled entries onto the *done_list* queue. After a producer process inserts the first entry onto the *done_list* queue, the process signals the *done_list_has_entry* semaphore to let the consumer process know that a *done_list* queue entry is available.

The *consumer* process block removes available entries from the *done_list* queue, writes the entry's buffer data, and inserts the empty entries onto the *free_list* queue. After the consumer process inserts the first entry onto the *free_list* queue, the process signals the *free_list_has_entry* semaphore to let the producer processes know that a *free_list* queue entry is available.

Example 5–1: Using Queues for Process Communication

```
MODULE producer_consumer;

  { This module uses queues with semaphores to communicate between
    processes. }

TYPE                                  { A record type that represents }
  entry = RECORD                      {  a queue entry. }
          links  : QUEUE_ENTRY;       { The entry's flink and blink. }
          ident  : INTEGER;           { Producer identifier. }
          buffer : LARGE_INTEGER;     { A user data buffer. }
          END;

CONST
  producer_max = 25;
  consumer_max = 50;

VAR
  free_list, done_list   : QUEUE_ENTRY;    { Declare queue headers. }
  free_list_has_entry    : SEMAPHORE;      { Declare semaphores. }
  done_list_has_entry    : SEMAPHORE;

PROCEDURE initialize;

  { This procedure initializes the free_list and done_list queues,
    creates the semaphores free_list_has_entry and done_list_has_entry,
    and fills the free_list queue with entries. }

CONST
  initial_free_count = 10;     { Declare a maximum of 10 free entries. }

VAR
  entry_ptr : ^entry;          { Declare a pointer to an entry. }
  first_entry : BOOLEAN;       { Declare the first entry flag. }
  entry_counter : INTEGER;     { Declare an entry counter. }
```

Example 5–1 Cont'd on next page

Example 5–1 (Cont.): Using Queues for Process Communication

```
BEGIN
  WRITELN('Initializing queues...');
  START_QUEUE(free_list);       { Initialize queues. }
  START_QUEUE(done_list);
  WRITELN('Creating semaphores...');
  CREATE_SEMAPHORE(free_list_has_entry, 0, 1);
  CREATE_SEMAPHORE(done_list_has_entry, 0, 1);
  WRITELN('Filling the free list queue with entries...');
  FOR entry_counter := 1 TO initial_free_count DO
    BEGIN
      NEW(entry_ptr);
      INSERT_ENTRY(free_list,
                   entry_ptr^.links,
                   first_entry,
                   QUEUE$TAIL);
      IF first_entry THEN                 { If this is the first entry, }
        SIGNAL(free_list_has_entry);      {  let the producer process   }
                                          {  know that an entry is      }
                                          {  available.                 }
    END;
  WRITELN('Initialization done...');
  WRITELN;
  END; { initialize }

PROCESS_BLOCK producer(producer_number : INTEGER);

  { This process block removes entries from the free_list queue, fills
    the entries with data, and inserts them on the done_list queue for
    the consumer process block. }

VAR
  entry_ptr : ^entry;                { Declare a pointer to an entry. }
  first_entry, empty : BOOLEAN;      { Declare first entry and empty
                                           flags. }
  producer_loop_counter : INTEGER := 0; { Declare a loop counter. }

BEGIN
  empty := TRUE;
  REPEAT
    IF empty THEN                         { Is the free_list queue empty? }
      WAIT_ANY(free_list_has_entry);      { If it's empty, wait for an    }
                                          {  entry and let the consumer   }
                                          {  continue.                    }
```

Example 5–1 Cont'd on next page

Example 5–1 (Cont.): Using Queues for Process Communication

```
    REMOVE_ENTRY(free_list,
                 entry_ptr::^QUEUE_ENTRY,
                 empty,
                 QUEUE$HEAD);
    GET_TIME(entry_ptr^.buffer);
    entry_ptr^.ident := producer_number;
    INSERT_ENTRY(done_list,
                 entry_ptr^.links,
                 first_entry,
                 QUEUE$TAIL);
    IF first_entry THEN               { If this is the first entry, }
        SIGNAL(done_list_has_entry);  {  let the consumer process   }
                                      {  know that an entry is      }
                                      {  available.                 }
    producer_loop_counter := producer_loop_counter + 1;
  UNTIL producer_loop_counter = producer_max;
END;  { producer }

PROCESS_BLOCK consumer;

  { This process block removes entries from the done_list queue,
    operates on the data, and inserts the entries on the free_list
    queue for the producer process block. }

VAR
  entry_ptr : ^entry;               { Declare a pointer to an entry. }
  first_entry, empty : BOOLEAN;     { Declare first entry and empty
                                          flags. }
    consumer_loop_counter : INTEGER := 0; { Declare a loop counter. }
```

Example 5–1 Cont'd on next page

Example 5–1 (Cont.): Using Queues for Process Communication

```
BEGIN
  empty := TRUE;
  REPEAT
    IF empty THEN                        { Is the done_list queue empty? }
      WAIT_ANY(done_list_has_entry);     { If it's empty, wait for an    }
                                         {  entry and let the producer   }
                                         {  process continue.            }
    REMOVE_ENTRY(done_list,
                 entry_ptr::^QUEUE_ENTRY,
                 empty,
                 QUEUE$HEAD);
    WRITELN('Data received from Producer ', entry_ptr^.ident:1,
            ' --- buffer value = ', entry_ptr^.buffer::INTEGER);
    INSERT_ENTRY(free_list,
                 entry_ptr^.links,
                 first_entry,
                 QUEUE$TAIL);
    IF first_entry THEN                  { If this is the first entry, }
       SIGNAL(free_list_has_entry);      {  let the producer process   }
                                         {  know that an entry is      }
                                         {  available.                 }
    consumer_loop_counter := consumer_loop_counter + 1;
  UNTIL consumer_loop_counter = consumer_max;
END;  { consumer }

PROGRAM queue_communication(OUTPUT);

VAR
  main : PROCESS;
  producer_process_1, producer_process_2 : PROCESS;
  consumer_process : PROCESS;

BEGIN  { main }
  initialize;
  CREATE_PROCESS(producer_process_1, producer, 1);
  CREATE_PROCESS(producer_process_2, producer, 2);
  CREATE_PROCESS(consumer_process, consumer);
  WAIT_ALL(producer_process_1,
           producer_process_2,
           consumer_process);
  WRITELN('Finished producing and consuming...');
  WRITELN('Program exiting...');
  EXIT;
END; { main }

END; { producer_consumer }
```

5.3 Passing Messages

A *message* is a block of contiguous bytes of memory that is transmitted between processes in the same or different jobs. The kernel maps message data into a job's unique, protected P0 virtual address space, making the data available to all processes in that job. Within a single processor, the kernel uses VAX memory management hardware to distinguish the virtual address space for each job. Within a local area network, the virtual address space for each job resides in the memory of a different target processor. By passing messages, the jobs in a VAXELN system can use the same mechanism to share data efficiently and transparently in both single-processor and network configurations.

Processes send messages to and receive messages from system-maintained queues called *ports*. The ports in a system store messages that are waiting to be sent or received. Calls to the CREATE_PORT procedure create ports dynamically and associate them with unique PORT values that can be used throughout the application: within the creating job, within other jobs on the same node, or within jobs on other network nodes.

One of the principal reasons for dividing an application into separate jobs is to distribute an application's jobs across a local area network (LAN). To allow interjob communication and to make the distribution of applications across LANs transparent, you can use the CREATE_NAME procedure to associate port values with *port names*. When ports are associated with names, a process can call the TRANSLATE_NAME procedure to look up a name in a table and use the returned port value to communicate with other processes and jobs. Port names can be up to 31 characters long and can be either local or universal. *Local names* are known only to processes and jobs on the node on which they are created. *Universal names* are known to processes and jobs on all VAXELN nodes in the local area network.

VAXELN systems can pass messages by using *datagrams* or *virtual circuits*. The datagram method, which uses the DECnet–VAX datagram, requires no explicit connection sequence and provides fast communication with low overhead. However, this method cannot guarantee message delivery or sequence (although the probability of received messages being correct is extremely high). Using datagrams, a process can obtain the value of any *named* port in the system, whether the port is on the same node or on a different node on the Ethernet.

The virtual circuit method, which uses the network services protocol within the VAXELN Network Service, is the preferred method for VAXELN systems to pass messages. Virtual circuits require two ports, usually in different jobs, to be connected as a pair. Despite the overhead of setting up and handling a virtual circuit connection, circuits offer the following advantages:

- Guaranteed delivery and sequence
- Flow control
- Message size is not limited by the underlying physical media characteristics due to automatic message segmentation and reconstruction

When a job sends a message to another job on the same node, the kernel unmaps the message buffer's address from the sending job's virtual address space and maps the address to the receiving job's address space. If the jobs are on different Ethernet nodes, the VAXELN Network Service transports the data across the network and places it in the receiving job's virtual address space. (Network configurations limit message size to the maximum imposed by relevant network devices.)

In addition to moving data, applications can use message-passing to synchronize and coordinate multiple processes and jobs. Most of the VAXELN services use message-passing to organize their work.

5.3.1 Messages

A message, as recognized by most network devices, is a block of contiguous bytes of memory. Usually, network devices, particularly Ethernet devices, impose a maximum size on a message. A network message also typically requires a number of bytes at the beginning of the message (a *protocol header*) to identify the rest of the message.

The kernel provides a MESSAGE object to describe a block of memory that can be moved from one job's virtual address space to another's. The block of memory is called the message data and is allocated dynamically by the kernel. A MESSAGE object and its data are created by calling the CREATE_MESSAGE procedure.

Message data is allocated by the kernel from physically contiguous, page-aligned blocks of memory, which allows the kernel to store the complete description of a message of reasonable length in a single MESSAGE object. Message data is mapped into a job's P0 virtual address region, so it is potentially accessible to all the processes in the job.

5.3.2 Message Ports

A PORT object represents a system-maintained message queue. A port is unique in that its identifying value is valid within all application jobs, not just within a particular job or jobs on a particular node. In other words, PORT values can be passed as arguments, sent in messages, or obtained from the RECEIVE procedure with certainty that they identify a unique destination for messages, somewhere in the application network. PORT values can be used with WAIT_ANY and WAIT_ALL to synchronize programs with the receipt of messages.

A message port can hold a maximum number of messages, specified when the port is created (the default is four). Messages are removed from a port by the RECEIVE procedure in first-in/first-out (FIFO) order. If more messages than the maximum are sent, they may be lost. (For exceptions, see Section 5.3.7.) A large message limit requires no more overhead than a small limit. Only the messages themselves determine the amount of memory consumed.

PORT values are assigned dynamically by the kernel to identify a particular message port. New values are returned by the CREATE_PORT procedure and are valid until used with the DELETE procedure to explicitly delete the port. For example, the following Pascal code fragment creates a new message port, limited to 10 messages, and puts the PORT value in the variable *newport*. The identifier *newport* is then used in later SEND, RECEIVE, and other message operations that require a PORT value.

```
VAR
  newport: PORT;

BEGIN
  CREATE_PORT(newport, LIMIT := 10);
```

5.3.3 Named Message Ports

To facilitate communication between jobs, the kernel provides a NAME object, a name table entry that associates character string names with message ports. Names are represented as separate objects to allow a port to have multiple names, if desired.

A process in the application that expects to communicate with processes outside its job can broadcast the necessary information about one or more of its message ports by creating names for them. If the process needs to communicate with a process on a different network node, it creates a *universal name*; if all communication occurs within a single node, a *local name* suffices. A local name is guaranteed to be unique within the local node. Universal names are guaranteed to be unique throughout the local area network. The translation and other maintenance of universal names is a function of the Network Service, as described in Chapter 9.

NOTE

> The processors in a closely coupled symmetric multiprocessing configuration constitute one Ethernet node and share the same local name table. Therefore, the images running on the processors must create unique local names.

These names are created with the CREATE_NAME procedure and can be deleted with DELETE. A NAME value specifies a name string of 1 to 31 characters for an associated port. The name string is used to obtain the PORT value of the associated port with the TRANSLATE_NAME procedure. That is, a program can look up a name in the name table and use the resulting PORT value to communicate with other jobs or processes.

Applicationwide services, such as disk drivers, commonly use such names. The disk driver makes names available for its message ports (for example, DUA0) so that another job or process can quickly translate the name into a PORT value for use in sending messages. In the case of a disk, program I/O is typically done with language-specific I/O procedures, whose runtime software performs the necessary name translation and message transmission for you.

When designing a system and writing the programs for it, you decide which processes are the communicators and create names appropriately. You then develop the programs and test the communication to your satisfaction. If you later decide to reconfigure the application — for example, by moving all the programs onto a single network node or, conversely, distributing programs among several newly added nodes — only the final program development step, system building, must be repeated, to describe the new hardware/software configuration. No changes to the programs themselves are necessary, because calls to TRANSLATE_NAME in the new application will obtain port information based on the new configuration.

Name strings can also be used directly (for example, as a parameter to the CONNECT_CIRCUIT procedure), in which case the translation is done by the procedure.

5.3.4 Message Transmission

To send a message, you must declare a pointer to the type of data you want to send, specify the size of the message buffer (C and FORTRAN), supply the pointer to CREATE_MESSAGE, use the pointer to fill in the message data, and supply the MESSAGE value to the SEND procedure. The following Pascal code example sends a message:

```
VAR mtext: ^VARYING_STRING(80);
  command: MESSAGE;
  destination: PORT;
    .
    .
    .
BEGIN
  CREATE_MESSAGE(command,mtext);
  mtext^ := 'START';
    .
    .
    .
  SEND(command,destination);
END.
```

An application can send a message as it was created or can send part of a message. To send part of a message, the call to SEND must specify a message size, indicating the length in bytes of the message data to be sent.

The SEND procedure removes the message data from your job's address space and places the MESSAGE object in the destination port. The SEND procedure also provides the following information to the receiver:

- The value of the sending process's job port or optionally, a different reply port specified by the sender
- The value of the destination port

- The size of the message sent

The receiver process waits for a message to arrive on its port and then uses the RECEIVE procedure to obtain it. The RECEIVE procedure automatically maps the message data into the receiver's address space, returns a MESSAGE value for the receiver's use, and optionally returns the identification of the reply port and destination port.

To reply to the message's originating job, the receiver uses the value of the reply port from RECEIVE, formulates an answer, and sends a reply to the reply port. (The receiver can use the same message data to form the reply; it need not create a new message.)

The receiver process must know beforehand the formats of the messages it can receive. That is, the sender and receiver must have established a *message protocol*. Defining a protocol is the basic design task in interjob communication.

For example, if the receiver is a server, it must know a set of predefined commands to which it will respond; it can return an error message to the sender (in most cases, an operator's terminal) if it receives a message that does not contain a valid command.

5.3.4.1 Expedited Messages

A distinct form of message, called an *expedited message*, is recognized by the kernel and the Network Service. An expedited message can bypass the normal, sequential flow control provided by the system.

For example, a transmitting process may have sent many messages to a receiving process, but before all the messages are received by the receiver, the transmitter may decide that the previous messages should be ignored, if possible. In this case, the transmitter can send an expedited message telling the receiver to halt.

Most applications do not need to use expedited data messages, because expedited data messages are restrictive, and there is no guarantee that an expedited message will be received before normal data messages. However, remote expedited data messages provide an interface to the DECnet Network Services Protocol *interrupt message* service, which is used by established protocols, such as the Data Access Protocol.

The following facilities and restrictions apply to expedited data messages:

- An expedited data message is sent by specifying a Boolean value to the EXPEDITE parameter of the SEND procedure.

- An expedited data message can contain a maximum of 16 bytes of data.
- Only one unreceived expedited message is allowed in a port. If a second expedited message is sent before the first is received, it has the same effect as a normal data message when the port is at its message limit; that is, either an error status is returned or an exception is raised, or the sending process waits until the first message has been received, depending upon the setting of the FULL_ERROR parameter when the circuit is connected or accepted.
- An expedited data message is received using the normal RECEIVE procedure but returns the alternate success status value, KER$_EXPEDITED. Therefore, if a receiver process needs to know if a message is an expedited or a normal message, and the protocol being used does not indicate which it is, the receiving process can compare the status to KER$_EXPEDITED.
- Expedited data messages queued to a port are received by the RECEIVE procedure before any normal data messages are received.

5.3.5 Datagrams and Circuits

Ports and messages can be used in two ways to transmit data:

- In the *datagram* method, one process can obtain the value of a port anywhere in the local area network, and can send the port a message with the SEND procedure.
- In the *circuit* method, any two ports can be bound into pairs called *circuits*. After establishing the circuit, the sending process has one port of its own bound to another port, which usually is in a different job or on a different network node. The sender sends the message to its own port, and the message is routed automatically to the other port in the circuit. Processes can both send to and receive from a circuit port.

In either method, a process can use the WAIT_ALL or WAIT_ANY procedure to wait for the receipt of a message on a port.

The datagram method requires no connection sequence, but correct delivery of datagrams to the destination is not guaranteed. (However, the datagram method guarantees with high probability that received messages are correct.) Also, datagram transmissions cannot be sent and received in a guaranteed order; that is, two messages sent to the same destination port can arrive in a different order.

Although circuits incur setup and handling costs, they offer the following advantages over the basic datagram method:

- Guaranteed delivery and sequence. Messages sent through circuits are guaranteed with high probability to be delivered — if the physical connection is intact — and to be delivered in the same sequence in which they are sent. The circuit method guarantees that the message arrives at the destination port regardless of its location or, if the message fails to arrive, that the sender is notified that the message could not be delivered.
- Flow control. Options of the procedures ACCEPT_CIRCUIT and CONNECT_CIRCUIT allow you to control the flow of messages through a circuit. That is, you can prevent a sending process from sending too many messages to a slower receiving process.
- Segmentation. Messages can have any length, and, if the transmission is across the network, the network services will divide the message into segments, transmit the segments in sequence, and reassemble them at the destination node.
- A user interface through Pascal I/O routines. The OPEN procedure permits you to open a circuit as if it were a file and to use the I/O routines, such as READ and WRITE, to transmit messages.

No performance penalty is incurred with the circuit method for messages transmitted on the same network node and only a small penalty is incurred over the network. For full generality, programs should assume that the sending and receiving jobs may be distributed on different nodes in a network. The circuit method is preferred for sending messages in almost every instance.

5.3.6 Programming with Circuits

You establish circuits between two ports by using the CONNECT_CIRCUIT and ACCEPT_CIRCUIT procedures. Options let you control the flow of messages through a circuit. For example, you can prevent a sending process from sending too many messages to a slower receiving process.

A process aimed to establish a circuit calls CONNECT_CIRCUIT and designates a destination port in another process. A connection-request message is sent to the designated port. Consider the following Pascal example:

```
CONNECT_CIRCUIT(myport, DESTINATION_NAME := 'request_server');
```

The variable *myport* is an existing port in the calling process that forms its half of the circuit. The string *'request_server'* specifies the destination name. CONNECT_CIRCUIT translates this name to designate the destination port.

The call to CONNECT_CIRCUIT causes a process to wait for the connection request to be accepted. The interval between the time the connection request is initiated for a port and the time the circuit is accepted should be no greater than the interval specified for the **Connect time** entry on the System Builder's System Characteristics Menu. For example, if the connect time is 45 seconds, the circuit should be accepted no later than 45 seconds after the call to CONNECT_ CIRCUIT initiates the connection request. A longer delay may cause the circuit to go into a bad state.

Elsewhere, the accepting process calls the ACCEPT_CIRCUIT procedure to wait for a connection-request message on the designated port. For example:

```
VAR
  server : NAME;
  receiver_port : PORT;
    .
    .
    .
CREATE_PORT(receiver_port, LIMIT := 10);
CREATE_NAME(server,'request_server',receiver_port);
    .
    .
    .
{
{ Wait for a connection request. When the wait is satisfied, a
{ circuit is established between the requestor and receiver_port.
{}
    .
    .
    .
ACCEPT_CIRCUIT(receiver_port);
```

You can request multiple connections on the same port, but you must distribute connections to other ports as they are received.

Consider the following Pascal example:

```
VAR
  server : NAME;
  receiver_port, connect_port : PORT;
BEGIN
  CREATE_PORT(receiver_port, LIMIT := 10);
  CREATE_PORT(connect_port);
  CREATE_NAME(server, 'request_server', receiver_port);
  ACCEPT_CIRCUIT(receiver_port, CONNECT := connect_port);

  { Wait for a connection request.  When the wait is satisfied, a
    circuit is established between the requestor and connect_port. }
    .
    .
    .
```

At this point, the acceptor can take a variety of actions to communicate with the requestor. For example, the acceptor can create a subprocess to continue the dialog and pass the subprocess the port value (*connect_port*) identifying its half of the circuit. The ACCEPT_CIRCUIT procedure can notify you of error conditions, such as an unreceived message in *receiver_port* or another connection request for which acceptance is pending.

When a process issues a call to ACCEPT_CIRCUIT, the kernel issues a call to the WAIT_ANY procedure for that process. When a message arrives at the port, the port is signaled and the kernel issues a call to RECEIVE, assuming that a connection request message is in the port. If the message is not a connection request, the kernel reissues the wait. For this reason, you cannot use the SIGNAL or KER$RAISE_PROCESS_EXCEPTION procedure to signal a process waiting for a circuit to be accepted; nor can you use the debugger to halt the process. To avoid this behavior, wait on the port and accept the circuit after the wait is satisfied.

Circuits are broken when either partner calls the DISCONNECT_CIRCUIT procedure. The SEND and RECEIVE procedures notify their callers if the designated port was disconnected by returning the status value KER$_DISCONNECT. As part of the corrective action for this condition, an application program must call the DISCONNECT_CIRCUIT procedure to disconnect the partner port. If appropriate, the program can then try to reestablish the circuit connection. Consider Example 5–2.

Example 5–2: Disconnecting the Partner Port After a Disconnect Operation

```
MODULE msg_symbol_ex;

INCLUDE $KERNELMSG;

PROGRAM use_msg_symbol(INPUT, OUTPUT);

VAR
  one_second : LARGE_INTEGER;
  data_port : PORT;
  dest_port_name : VARYING_STRING(8);
  msg : MESSAGE;
  stat : INTEGER;
    .
    .
    .
BEGIN
    .
    .
    .
  CREATE_PORT(data_port);
  REPEAT
    WAIT_ANY(TIME := one_second);
    CONNECT_CIRCUIT(data_port,
                    DESTINATION_NAME := dest_port_name,
                    STATUS := stat);
  UNTIL stat := KER$_SUCCESS;

  SEND(msg, data_port, STATUS := stat),

  { If the send operation fails because the circuit was disconnected
  { by the partner process, the sender process must clean up by
  { disconnecting the port on its end of the circuit.  Once both ports
  { are disconnected, reestablish the circuit connection and try
  { to send the message again.
  {}

  IF stat = KER$_DISCONNECT THEN
    BEGIN

      { Disconnect the port before trying to reestablish the
      { connection.
      {}

      DISCONNECT_CIRCUIT(data_port);
```

Example 5–2 Cont'd on next page

Example 5–2 (Cont.): Disconnecting the Partner Port After a Disconnect Operation

```
            CONNECT_CIRCUIT(data_port,
                            DESTINATION_NAME := dest_port_name,
                            STATUS := stat);
            SEND(msg, data_port, STATUS := stat),
        END;
      .
      .
      .
END.
END;
```

If the disconnect condition exists, the call to DISCONNECT_CIRCUIT cleans up and prepares *data_port* for another connection.

5.3.7 Port Limits and Flow Control

An advantage of using a circuit for a message exchange is that the kernel and Network Service provide a function called *flow control.* Under flow control, the flow of messages from a transmitting process to a receiving process is controlled to ensure that unreceived messages do not consume excessive memory in the system.

When a process sends a message with SEND, the message is queued in a specified destination port. If the transmitting process can produce messages faster than the receiving process can consume them and if no limit is placed on the number of messages that can be queued, the messages might use all the available memory. To avoid that situation, ports have a limit on the number of unreceived messages that can be queued at a time; the limit is specified when the port is created.

5.3.7.1 Flow Control with Unconnected Ports

If a port that is not connected in a circuit is full and an application sends a message to the port, the call to SEND returns a failure status or exception. If the port is not on the same node, the message can be lost.

5.3.7.2 Flow Control with Circuits

If a circuit-connected port is full, the sender is, by default, put into the waiting state until the port is no longer full. The transmission is then successfully completed. The implicit waiting performed by the SEND procedure evens the flow of messages between the transmitting process and receiving process without having to explicitly program for the condition.

Since some applications may not need implicit waiting, an argument to the ACCEPT_CIRCUIT and CONNECT_CIRCUIT procedures allows the calling process to specify that it wants a SEND call to return an error status or exception rather than to wait.

5.3.8 Programming Considerations for Message Communication

When programming message communication, consider the following:

- When programs use circuits to communicate, you must ensure that the programs cooperate. One program is to call the CONNECT_ CIRCUIT procedure, and the other program is to call the ACCEPT_ CIRCUIT procedure. If the programs do not cooperate, various results are possible, including loss of pool blocks used for the connect request, return of the KER$_CONNECT_TIMEOUT status value, unexpected satisfied waits, or the circuit entering a bad state.
- When a program issues a call to CONNECT_CIRCUIT for a port, an ACCEPT_CIRCUIT for that port must be pending, or the interval between the time the connection request is initiated and the time the circuit is accepted should be no greater than the interval specified for the **Connect time** entry on the System Builder's System Characteristics Menu. A longer delay may cause the circuit to go into a bad state.
- A program cannot operate on a port while the port is being used in an ACCEPT_CIRCUIT or CONNECT_CIRCUIT operation. When a program calls the ACCEPT_CIRCUIT and CONNECT_ CIRCUIT procedures to establish a circuit connection, the kernel and Network Services perform a sequence of operations. This sequence may satisfy a wait request issued by another process in the job unexpectedly.

- Once a program establishes a circuit connection, multiple processes can perform simultaneous SEND and WAIT/RECEIVE operations. In this situation, the WAIT can be satisfied even if the call to RECEIVE returns the status value KER$_NO_MESSAGE. The combination of the wait being satisfied and the status value being returned results when a program uses multiple receivers or when the SEND procedure resumes after a flow control suspension. In the multiple receiver case, another receiver process may have received the message. In the flow control case, internal flow control mechanisms may have unblocked all processes waiting on the port. If a process receives a KER$_NO_MESSAGE status from a call to RECEIVE after a WAIT is satisfied, it should WAIT on the port again.

5.3.9 Kernel Services for Message Transmission

The kernel services affecting the state of MESSAGE, PORT, and NAME objects are summarized in Sections 5.3.9.1 to 5.3.9.12.

5.3.9.1 ACCEPT_CIRCUIT Procedure

The ACCEPT_CIRCUIT procedure causes the invoking process to wait for a circuit connection. On successful completion, the circuit is established between two ports.

The invoker's half of the circuit can be the port used to wait for the connection request or, optionally, a different port. This optional parameter allows a program, such as a resource service, to create a name for its connection-request port but to use a different port for the connection itself; in this way, the server could create a name for the first port to establish simultaneous circuits with several different processes or jobs. The only valid message that can be received at the connection-request port is the kernel's internal connection request; other messages are discarded by the system.

By default, when a process sends a message on a circuit with SEND, the operation waits if the partner port is full, a method called *flow control*. When you accept a circuit connection, you have the option of specifying that you want an error status or the corresponding exception instead of the implicit wait.

An optional argument supplies a data value that is received by the process requesting the circuit connection in its CONNECT_CIRCUIT call. Another optional argument receives data passed by the requesting process in its CONNECT_CIRCUIT call. These data values are called *connect data* and *accept data*, respectively, and are strings of up to 16 bytes.

5.3.9.2 CONNECT_CIRCUIT Procedure

The CONNECT_CIRCUIT procedure connects a port to a specified destination port and causes the invoking process to wait for the connection request to be accepted.

If a process calls ACCEPT_CIRCUIT with the destination port, the two ports are bound in a circuit. The destination port can be specified by using a name string established by the CREATE_NAME procedure or by using a PORT value giving the destination for the connection request.

By default, when a process sends a message on a circuit, the SEND procedure performs an implicit wait if the partner port is full — that is, contains its limit of unreceived messages; this type of *flow control* is usually used with circuits. With CONNECT_CIRCUIT, you have the option of disabling the implicit wait, causing SEND to receive an error status or raise an exception if the partner port is full.

An optional argument supplies data to the process receiving the connection request. Another optional argument receives data supplied by the accepting process in its ACCEPT_CIRCUIT call.

NOTE

The interval between the time a connection request is initiated for a port and the time the circuit is accepted should be no greater than the interval specified for the **Connect time** entry on the System Builder's System Characteristics Menu. For example, if the connect time is 45 seconds, the circuit should be accepted no later than 45 seconds after the call to CONNECT_CIRCUIT initiates the connection request. A longer delay may cause the circuit to go into a bad state.

5.3.9.3 CREATE_MESSAGE Procedure

The CREATE_MESSAGE procedure creates a MESSAGE object and allocates and maps its message data into the job's P0 address space for use by the SEND and RECEIVE procedures, returning the MESSAGE value that identifies the message and a pointer to the allocated message data. A program can use the pointer to the message data to store data that is to be moved to another job's address space.

5.3.9.4 CREATE_NAME Procedure

The CREATE_NAME procedure creates a name string of 1 to 31 characters for a specified port as an entry in a name table and returns the NAME value that identifies that name. An optional argument specifies that the new name is local (known only on its own node), universal (known on any node in the local area network), or both; local is the default. If the Name Service is not present in the system, all names are placed in the local name table, even if you specify universal or both. (For information about the Name Service, see Chapter 9.)

Names created by this procedure are guaranteed to be unique within the specified name space: local or universal. If you try to create a name that is not unique, the procedure does not create a NAME object and returns an error status.

When you create a universal name in a local area network configuration, the Name Service on each node in the local area network can translate universal names created by other nodes in the local area network.

5.3.9.5 CREATE_PORT Procedure

The CREATE_PORT procedure creates a message port, returning the PORT value that identifies the port. An optional integer expression supplies the maximum number of messages that can be queued to the port at one time. If the maximum is exceeded, the sender is notified; the default value is 4.

5.3.9.6 DELETE Procedure

The DELETE procedure removes the MESSAGE, PORT, or NAME object from the system.

When a message is deleted, it is unavailable for sending or receiving, and pointers to the message data become invalid.

When a port in a circuit is deleted, the connected port is disconnected, messages at the port are deleted, and the wait conditions of any waiting processes are satisfied with the completion status KER$_BAD_VALUE.

When a universal name is deleted, the Network Service on each node ensures that the deletion is reflected in the list of universal names. The deletion of local names is performed by the kernel on the local node and does not involve the Network Service.

5.3.9.7 DISCONNECT_CIRCUIT Procedure

The DISCONNECT_CIRCUIT procedure breaks the circuit connection between two ports. If a process is waiting for either port in the circuit, its wait condition is satisfied. A request for connection can be rejected by first calling ACCEPT_CIRCUIT and then calling DISCONNECT_CIRCUIT.

5.3.9.8 JOB_PORT Procedure

The JOB_PORT procedure returns a PORT object value identifying the caller's job port. A unique job port is created whenever a job is started.

5.3.9.9 RECEIVE Procedure

The RECEIVE procedure removes a message from a message port. The procedure maps the message data into the receiver job's virtual address space, returns a MESSAGE value identifying the message, and optionally returns PORT values identifying the reply port and destination port. The value is normally the same value supplied by the sender for the receiver's port.

An integer argument, optional for Pascal, receives the size in bytes of the message data.

5.3.9.10 SEND Procedure

The SEND procedure removes a message buffer from the sender's address space and then places the MESSAGE object describing the buffer in the destination message port. If the message is being sent through a circuit, the destination message port you specify is the sender's port, and the message arrives at the receiver's port.

By default, when a process sends a message on a circuit, the SEND procedure performs an implicit wait if the partner port is full, a method called flow control. When you accept a circuit connection, you have the option of specifying that you want an error status or the corresponding exception instead of the implicit wait.

Other SEND arguments, optional for Pascal, specify the length in bytes of the message data to be sent, specify a reply PORT value, and specify whether to expedite the message. The size of an expedited message must not exceed 16 bytes.

5.3.9.11 TRANSLATE_NAME Procedure

The TRANSLATE_NAME procedure returns a value identifying a named port. The specified name string is used to search for a NAME object with a matching string. If the NAME object is found, a value for the named port is returned.

You can specify that a name is to be looked up in the local name table, the universal name table, or both; the local name table is searched first if both are specified.

The Name Service provides the universal name table. Therefore, to translate universal names, the Name Service must be present in the system. An attempt to translate a universal name without the Name Service present causes the service to try to translate the name using the local name table. For more information about the Name Service, see Chapter 9.

5.3.9.12 WAIT_ANY and WAIT_ALL Procedures

A wait for a port, including a port in a circuit, is satisfied when it has a message in it. Waiting for a port causes no modification to the port, and all waiting processes continue if their wait conditions are otherwise satisfied. Both procedures can specify a timeout argument, which defines a time interval or absolute time after which the waiting process proceeds regardless of the states of the objects.

Normally, a process must call a WAIT procedure, then call RECEIVE. Calling RECEIVE without first calling a WAIT procedure may return a no-message status.

If a process needs to accept a circuit connection and wait for one or more other objects at the same time, it can call a WAIT procedure specifying the port and the other objects. When the wait is satisfied because a message is received (the PORT object is returned as the wait result), the process can call ACCEPT_CIRCUIT.

5.4 Sharing Memory Areas

Jobs executing on the same node can communicate by sharing an *area*, a common region of physically contiguous memory. Each job that shares an area must create it. When a job creates an area, the kernel maps the physical memory associated with the area to the job's P0 virtual address space. The first time an application creates an area, the kernel also associates the area with an event or semaphore. Jobs can use the event or semaphore to synchronize access to the area.

Once an application creates an area, the area remains available until all jobs sharing the area terminate or delete their references to the area.

VAXELN applications can use the following procedures to create and use areas:

Procedure	**Description**
CLEAR_EVENT	Sets the state of an area's event to EVENT$CLEARED.
CREATE_AREA	Creates a new area or maps an existing area of memory into the creating job's P0 virtual address space and associates the area with a binary semaphore.
CREATE_AREA_EVENT	Creates a new area or maps an existing area of memory into the creating job's P0 virtual address space and associates the area with an event.

Procedure	Description
CREATE_AREA_SEMAPHORE	Creates a new area or maps an existing area of memory into the creating job's P0 virtual address space and associates the area with a semaphore.
DELETE	Deletes an area.
ELN$INITIALIZE_AREA_LOCK	Initializes an area lock variable.
ELN$LOCK_AREA	Locks an area for exclusive access.
SIGNAL	Signals an area. If the area is associated with an event, the kernel sets the event to a signaled state. If the area is associated with a semaphore, the kernel increments the semaphore's count.
ELN$UNLOCK_AREA	Unlocks an area.
WAIT_ALL and WAIT_ANY	Waits on an area. If the area is associated with an event, the kernel lets the calling job access the area or causes the job to wait, depending on whether the event is signaled or cleared. If the area is associated with a semaphore, the kernel checks the semaphore's count, and based on the count value lets the calling job access the area or causes the job to wait. If the count is greater than zero, the kernel decrements the count and lets the calling job access the area. Otherwise, the job waits.

The following sections explain how applications can use these procedures to do the following:

- Create areas, Section 5.4.1
- Synchronize access to areas using events, Section 5.4.2
- Synchronize access to areas using semaphores, Section 5.4.3
- Use area lock variables to optimize waiting and signaling operations, Section 5.4.4
- Use areas to synchronize job execution, Section 5.4.5

- Delete areas, Section 5.4.6

NOTE

The systems in a closely coupled symmetric multiprocessing configuration cannot use areas as a means of sharing memory. To share memory, such systems must use the ALLOCATE_MEMORY routine to allocate memory at a physical address on the primary system, which is accessible to all processors.

5.4.1 Creating Areas

An application can create a new area or map an existing area of memory into a job's P0 virtual address space by calling the CREATE_AREA, CREATE_AREA_EVENT, or CREATE_AREA_SEMAPHORE procedure. A procedure call that creates a new area associates the area with an event or semaphore as follows:

Procedure	Object Type
CREATE_AREA	Binary semaphore
CREATE_AREA_EVENT	Event
CREATE_AREA_SEMAPHORE	Binary or counting semaphore

The event or semaphore controls access to the area. An application that uses one job to write to an area and lets several jobs read from that area might use an event to control area access. A counting semaphore lets an application specify a maximum number of jobs (maximum count value) that can access an area at a given time. A binary semaphore provides exclusive access to an area.

An application uses an area's event or semaphore to control access by changing the event or semaphore's state. The state of an event changes when the application signals or clears an area. The state of a semaphore changes when an application signals or waits on an area. Sections 5.4.2 and 5.4.3 explain how to synchronize area access.

Calls to the CREATE_AREA, CREATE_AREA_EVENT, and CREATE_AREA_SEMAPHORE procedures must specify an area variable, a data pointer, the size of the area (C and FORTRAN), and an area name. The area variable receives a value that identifies the area. You use this variable to identify the area in calls to other routines, such as CLEAR_EVENT, SIGNAL, WAIT_ALL, and WAIT_ANY.

The data pointer receives the area's base virtual address. A VAXELN Pascal data pointer also implicitly defines the area's size. The data pointer can be of any type except ^ANYTYPE and the procedure uses the size of that argument's type to determine the area's size. For C and FORTRAN applications, you must specify the area size in bytes. The value you specify is increased to the next multiple of 512. Specify a size of 0 to use an area only as a mechanism for synchronizing job execution (see Section 5.4.5).

A string of 1 to 31 characters specifies an area's name. The name must be unique within the application.

Calls to the CREATE_AREA_EVENT and CREATE_AREA_SEMAPHORE procedures also must specify arguments that initialize the event or semaphore that the procedure creates. You can initialize the state of an area's event to EVENT$CLEARED or EVENT$SIGNALED. You must specify initial count and maximum count values for a semaphore that CREATE_AREA_SEMAPHORE creates. (The CREATE_AREA procedure automatically initializes the initial and maximum count values to 1.)

Procedure calls that create an existing area only map the area to the calling job's P0 virtual address space. The use of a shared area, whether it is associated with an event or semaphore, must remain consistent throughout an application. When you create an area that is associated with an event, subsequent create area procedure calls specifying that area must be CREATE_AREA_EVENT calls and they must specify the same initial event state. Likewise, if you create an area that is associated with a semaphore, subsequent create area procedure calls specifying that area must be CREATE_AREA or CREATE_AREA_SEMAPHORE calls and they must specify the same initial and maximum count values. For example, if a call to CREATE_AREA_SEMAPHORE creates a new area named *common_area* and specifies 0 and 3 as the semaphore's initial and maximum counts, all subsequent calls to CREATE_AREA_SEMAPHORE that specify *common_area* must also specify 0 and 3 as the semaphore count values.

An optional virtual address argument lets you specify the starting job P0 virtual address at which the specified area is to be mapped. You can specify this argument in calls to CREATE_AREA, CREATE_AREA_EVENT, or CREATE_AREA_SEMAPHORE. If you do not specify an address, the kernel allocates a free range of P0 virtual addresses.

The following C example creates a 5000-byte area that has an associated semaphore with an initial value of 1 and a maximum count of 2:

```
#module cr_area_semaphore
#include $vaxelnc
   .
   .
   .
main()
{
  int         completion_status, size, init_count, max_count;
  AREA        area1
  char *      area1_ptr
  static $DESCRIPTOR(name_string1, "AREA_1");
   .
   .
   .
  init_count = 1;
  max_count = 2;
  size = 5000;

  ker$creat_area_semaphore(&completion_status,
                            &area1,
                            &area1_prt,
                            size,
                            &name_string1,
                            init_count,
                            max_count,
                            NULL);
   .
   .
   .
}
```

Once you have created an area, subsequent calls to CREATE_AREA, CREATE_AREA_EVENT, or CREATE_AREA_SEMAPHORE that specify the area you created map that area to the P0 virtual address space of each calling job. If you specified a virtual address in the procedure call that initially created an area, subsequent calls that specify that area must specify the same virtual address.

- When a shared area is mapped to the same virtual address for each sharing job, the area is position-dependent and pointer values are equivalent in each job's address space. Thus, the sharing jobs can place absolute and relative pointers in the area.
- When a shared area is mapped to different virtual addresses by different jobs because no virtual address was specified, the area is position-independent and the sharing jobs can place only relative pointers in the area.

In all cases absolute and relative pointers within an area must point to other addresses within the area if they are to be used by different jobs.

Jobs that share an area can map none, some, or all of the area's memory, depending on the area size that you specify. If a job shares part of an area, the shared part begins at the start of the area.

5.4.2 Synchronizing Access to Areas with Events

Jobs can synchronize access to a shared area that is associated with an event by waiting on, signaling, and clearing the area. Jobs can access the area so long as the event is in a signaled state.

To wait on an area, you must specify its AREA variable in the object value list of a call to the WAIT_ALL or WAIT_ANY procedure. A wait operation for an area that is associated with an event causes the calling job to wait for the area to be signaled. If the area's event is already in the signaled state, jobs calling the WAIT_ALL or WAIT_ANY procedure can access the area immediately. If an area's event is in the cleared state, calling jobs block and wait for another job to signal the area with a call to the SIGNAL procedure. The call to SIGNAL changes the state of the area's event to EVENT$SIGNALED and unblocks all waiting jobs.

An area's event remains in the signaled state until a job clears the event with a call to CLEAR_EVENT. The call to CLEAR_EVENT must specify the AREA object with which the event is associated.

Example 5–3 shows two modules that use an area associated with an event. The first module, *area_writer*, contains a job that writes data to the area. The *area_writer* module also creates two jobs, using the second module, *area_reader*. The reader jobs read data from the area, using the area's associated event as a synchronization mechanism. Messages synchronize the writer's ability to gain access to the area.

Example 5–3: Synchronizing Access to Areas with Events

```
/*******************************************************************/
/*                     Writer Module                                */
/*******************************************************************/
#module writer_prog
#include $vaxelnc
#include descrip

/*
 *  Declare external variables.
 */

static int   exit_status;

main()
{

  /*
   *  Declare master process local variables.
   */

  int         completion_status, area_size, i;
  char        *n, a_ch, ch[10];
  char *      area_ptr;
  AREA        area_with_event;
  static $DESCRIPTOR(area_name_string, "Shared_Area");
  PORT        job_port1, job_port2, job_port3, another_port2,
              another_port3;
  NAME        port_name2, port_name3;
  static $DESCRIPTOR(port_name_string2, "Port_For_Job2");
  static $DESCRIPTOR(port_name_string3, "Port_For_Job3");
  static $DESCRIPTOR(reader_program_name, "reader_prog");

  printf("This is from Job 1.");
  area_size = 50;

  /*
   *  Create an area of size 50 bytes and associate that area with
   *  a cleared event.
   */

  ker$create_area_event(NULL,
                 &area_with_event,   /* Longword containing area ID */
                 &area_ptr,          /* Data pointer                */
                 area_size,          /* Area size                   */
                 &area_name_string,  /* String - name of area       */
                 EVENT$CLEARED,      /* Initial state - clear       */
                 NULL);              /* Virtual address             */

  printf("\n Area created of size %d bytes", area_size);
```

Example 5–3 Cont'd on next page

Example 5–3 (Cont.): Synchronizing Access to Areas with Events

```
ker$create_port(NULL,
                &another_port2,
                NULL);

ker$create_name(NULL,
                &port_name2,
                &port_name_string2,
                &another_port2,
                NAME$LOCAL);

ker$create_port(NULL,
                &another_port3,
                NULL);

ker$create_name(NULL,
                &port_name3,
           &port_name_string3,
           &another_port3,
             NAME$LOCAL);

printf("\n\n Now create Job 2.\n\n");

ker$create_job(NULL,
               &job_port2,
               &reader_program_name,
               NULL,
               '2');

printf("\n\n Now create Job 3.\n\n");

ker$create_job(NULL,
               &job_port3,
               &reader_program_name,
               NULL,
               '3');

/*
 *  Initialize the area to all 'A's.
 */

for (n = area_ptr; n < &area_ptr[area_size];)
{
  *n++ = '\A';
}

/*
 *  Print out the area contents.
 */
```

Example 5–3 Cont'd on next page

Example 5–3 (Cont.): Synchronizing Access to Areas with Events

```
for (n = area_ptr; n < &area_ptr[area_size1];)
{
  a_ch = *n++;
  printf("\n ch = %c \n", a_ch);
}
/*
 *  Signal area_with_event to let the reader jobs use the data.
 *  Once the area is signaled, all reader jobs can gain access to
 *  the area.
 */
ker$signal(NULL, area_with_event);
for (i = 1; i <= 4; i++ )
{
  /*
   *  Use a mechanism to synchronize the two reader jobs finishing
   *  with the area.
   *
   *  Wait for messages from the two reader jobs.  The messages
   *  indicate that the reader jobs are finished using the area.
   */
          .
          .
          .
  /*
   *  Lock the area for exclusive access by clearing the area's
   *  event.  The writer job can then modify the data.
   */
  ker$clear_event(NULL, area_with_event);
  /*
   *  Send messages to the two reader jobs.  The messages indicate
   *  that the reader jobs are finished using the area and that the
   *  area is locked by this job.
   */
          .
          .
          .
  /*
   *  Modify the area's contents.
   */
```

Example 5–3 Cont'd on next page

```
    for (n = area_ptr; n < &area_ptr1[area_size1];)
      {
         *n++= *n + 1;
      }

    /*
     *  Print the new contents
     */

    {
       for (n = area_ptr; n < &area_ptr[area_size1];)
       {
          a_ch = *n++;
          printf("\n\n ch = %c \n\n", a_ch);
       }

       /*
        *  Signal the area so the reader jobs can read the new data.
        */

       ker$signal(NULL, area_with_event);
    }

    /*
     *  Mark the area and its associated event for deletion when it
     *  is no longer needed.
     */

    ker$delete(NULL,my_area1);
  }
}
/*********************************************************************/
/*                         Reader Module                             */
/*********************************************************************/

#module reader_prog
#include $vaxelnc
#include descrip

main()
{
  int         completion_status, area_size, i;
  char        *n, ch[50];
  char *      area_ptr;
  AREA        area_with_event;
  void        subprocess_code();
  static $DESCRIPTOR(area_name_string, "Shared_Area");

  area_size = 50;
```

Example 5–3 Cont'd on next page

Example 5–3 (Cont.): Synchronizing Access to Areas with Events

```
/*
 *  Map area_with_event to the P0 virtual address space for the
 *  reader jobs.  The call must specify the same argument data as
 *  was specified in the call that initially created the area.
 */

ker$create_area_event(NULL,
                &area_with_event,  /*Longword containing area ID */
                &area_ptr,         /* Data pointer               */
                area_size,         /* Area size                  */
                &area_name_string, /* String - name of area      */
                EVENT$CLEARED,     /* Initial state - clear      */
                NULL);             /* virtual address            */

for (i = 1; i <= 5 ; i++)
{

  /*
   *  Wait for the writer job to signal the area.
   */

  ker$wait_any(NULL,
               NULL,
               NULL,
               area_with_event);
             .
             .
             .
  /*
   *  Use the data in the area, send a message to the writer job
   *  indicating that we are finished using the area, and wait for
   *  a message from the writer job, indicating that all reader
   *  jobs are finished.
   */

             .
             .
             .
}

/*
 *  Mark the area and its associated event for deletion when it
 *  is no longer needed.
 */

ker$delete(NULL, area_with_event);
}
```

5.4.3 Synchronizing Access to Areas with Semaphores

Jobs can synchronize access to a shared area that is associated with a semaphore by waiting on and signaling the area. A job gains access to the area by waiting on it. When the job no longer needs access, it should signal the area, allowing other sharing jobs to gain access.

To wait on an area, you must specify the area's value in the object value list of a call to the WAIT_ALL or WAIT_ANY procedure. A wait operation for an area that is associated with a binary semaphore gives the calling job exclusive access to the area. If an area is associated with a counting semaphore, the semaphore's maximum count indicates the number of jobs that can wait on, and thus access, an area for read operations simultaneously. For example, if the maximum count value is 3, up to three jobs can access the area simultaneously. The kernel decrements the semaphore's count value by one for each satisfied wait on the area. When the semaphore's count value equals 0, subsequent calls to WAIT_ALL or WAIT_ANY that specify the area cause the calling jobs to block and wait for the area to be signaled. The kernel places the waiting jobs in a queue in the order in which they issued the calls to WAIT_ALL or WAIT_ANY.

NOTE

> If multiple jobs can gain simultaneous access to an area, the application should ensure the integrity of the shared data by allowing the jobs to only read the data. A job that writes to a shared area must have exclusive access.

Primarily, areas associated with counting semaphores are for synchronizing access to available units of a shared resource. In such a case, the area has a size of 0. For more information about using areas in this way, see Section 5.4.5.

When a job no longer needs to access an area that is associated with a semaphore, it can inform the kernel by specifying the area in a call to the SIGNAL procedure. The call to SIGNAL increments the semaphore's count value by one. If one or more jobs are waiting to access the area, the kernel lets the first waiting job in the semaphore's queue access the area and then decrements the semaphore's count by one. The next job in the queue waits for the next signal operation.

5.4.4 Using Area Lock Variables to Optimize Waiting and Signaling Operations

You can improve the performance of area access synchronization by using an area lock variable. An application can use an area lock variable only if the area is created with an associated binary semaphore that is properly initialized. Area lock variables allow you to achieve the same effect as an area with an associated binary semaphore (with initial and maximum counts of 1) without calling a kernel service unless contention exists. Generally, when a process locks an area to gain access to a shared resource, the process does not have to call the WAIT_ALL or WAIT_ANY procedure. The kernel issues a wait only if area is already locked.

To use an area lock variable, you must declare a variable of type AREA_LOCK_VARIABLE and place that variable in the data portion of the area. You then initialize the variable using a call to the ELN$INITIALIZE_AREA_LOCK procedure and then synchronize access to the area with calls to ELN$LOCK_AREA (instead of WAIT_ANY or WAIT_ALL) and ELN$UNLOCK_AREA (instead of SIGNAL).

To use the area-locking procedures in Pascal applications, you must include the module $MUTEX from the RTLOBJECT library. To use them in C programs, you must include the module **$mutex** from VAXELNC.TLB.

The ELN$INITIALIZE_AREA_LOCK procedure waits on the AREA object. When the wait is satisfied, the kernel places the area's semaphore in a closed state for subsequent lock and unlock operations, and then the procedure sets the area lock variable's initial state to unlocked. (The area semaphore's state changes only when lock contention exists.)

An area lock variable should be initialized only once by one process; *no error status is returned if the variable is initialized more than once.*

A job locks an area for exclusive access by calling the ELN$LOCK_AREA procedure. A call to this procedure must specify the area to be locked and the lock variable that controls access to the area. If the area is already locked, the calling job waits on the area. Generally, when a job locks an area, the job does not need to issue a wait before accessing the area. The kernel issues a wait only if the area is already locked.

A job can unlock an area by calling the ELN$UNLOCK_AREA procedure. This procedure releases an area and signals processes that are waiting to access it. A call to this procedure must specify the area to be unlocked and the lock variable that controls access to the area.

If a binary semaphore is open (count = 1) and the semaphore is signaled, a count overflow error occurs. In contrast, the locking and unlocking of areas is not protected by this exception-raising mechanism. Under certain circumstances (in which two or more jobs contend for an area) unlocking an already unlocked area can cause the calling job to block indefinitely. Therefore, when using area lock variables, you should adhere to the following guidelines:

- The first operation on an initialized area lock variable must be a lock operation.
- You must pair lock and unlock operations within the code of the jobs using the area.

5.4.5 Using Areas to Synchronize Job Execution

You can use an area of size 0 to synchronize job execution. In such cases, the AREA object represents an interjob event or semaphore or a user-defined resource. As an interjob synchronizing mechanism, an area functions the same as events and semaphores used to synchronize processes in the same job. Instead of synchronizing processes that execute as parts of the same job, you synchronize processes that execute as parts of different jobs. For more information about using semaphores and events to synchronize processes, see Sections 4.4 and 4.5.

An area can also represent the available units of a shared user-defined resource that is application-specific. The printers available on a system are an example of such a resource. The event or semaphore associated with the area serves as a resource access control mechanism.

Example 5–4 shows two modules that use an area associated with a counting semaphore to control the ability of three jobs to gain access to two shared resources. The first module does the following:

- Creates the area with an initial count of 1 and a maximum count of 2
- Waits on the area to gain access to a resource
- Initializes the two resources
- Signals the area twice to let two jobs gain access to the resource
- Creates two other jobs

- Allows the three jobs to use the two resources five times each, with only two of the jobs using the resources at a time.

The second module, which is executed by the two created jobs, maps the area to its P0 space, waits until the resource is available, uses the resource, and signals the area when the resource is no longer needed.

All three jobs use the area's semaphore to synchronize access to the shared resources.

Example 5–4: Synchronizing Job Execution with Semaphores

```
{*******************************************************************}
{*                          Module 1                              *}
{*******************************************************************}

MODULE cr_area_sema_1;

PROGRAM cr_area_sema_prog1(INPUT,OUTPUT);

CONST
  area_size = 0;

TYPE
  area_type = string(area_size);

VAR
  i : INTEGER;
  completion_status : INTEGER;
  resource : AREA;
  resource_ptr : ^area_type;
  job2_port, job3_port : PORT;

BEGIN
  writeln('Create resource area...');

  {
  {  Create an area of size 0 and associate that area with a counting
  {  semaphore that has initial and maximum count values of 1 and 2.
  {  As many as two jobs can gain access to the resource without
  {  waiting.
  {}
```

Example 5–4 Cont'd on next page

```
CREATE_AREA_SEMAPHORE(
              resource,            { Longword containing area ID }
              resource_ptr,        { Data pointer                }
              'Shared_Area',       { String - name of area       }
              1,                   { Initial count               }
              2,                   { Maximum count               }
              completion_status);  { Status                      }

{
{  Wait on the area. Since the initial count was 1, this job gains
{  access immediately.
{}

WAIT_ANY(resource);
              .
              .
              .
{  Set up the shared resource. }
              .
              .
              .

{
{  Signal the area's semaphore.  When the area is signaled, the
{  kernel increments the semaphore count so that another job can
{  use the area.  We signal the area twice to make the resource
{  available to two jobs.
{}

SIGNAL(resource);
SIGNAL(resource);

{ Create the other two jobs. }

{ Create the second job and pass it the program argument '2'. }
CREATE_JOB(job2_port, 'cr_area_sema_prog2', '', '', '2');

{ Create the third job and pass it the program argument '3'. }
create_job(job3_port, 'cr_area_sema_prog2', '', '', '3');

FOR i := 1 TO 5 DO
  BEGIN

    {
    {  Wait on the area until the resource is available. If the
    {  semaphore count is greater than 0, the job gains access to
    {  the area.  If the count is 0, the job waits.
    {}
```

Example 5-4 Cont'd on next page

Example 5–4 (Cont.): Synchronizing Job Execution with Semaphores

```
      WAIT_ANY(resource);
                .
                .
                .
      { Determine which resource is available and use it. }
                .
                .
                .
      {
      {  Signal the area's semaphore to indicate that the resource is
      {  no longer in use.  When the area is signaled, the kernel
      {  increments the semaphore count so that another job can use
      {  the area.
      {}
      SIGNAL(resource);
   END;
   WRITELN('Job 1 has used a resource 5 times.');
   {
   {  Mark the area and its associated semaphore for deletion when
   {  they are no longer needed.
   {}
   DELETE(resource);
END.
END;
{*********************************************************************}
{*                          Module 2                                 *}
{*********************************************************************}
MODULE cr_area_sema_2;
PROGRAM cr_area_sema_prog2(INPUT,OUTPUT);
CONST
  area_size = 0;
TYPE
  area_type = STRING(area_size);
VAR
  i, j : INTEGER;
  completion_status : INTEGER;
  job_number : STRING(1);
  resource : AREA;
  resource_ptr : ^area_type;
```

Example 5–4 Cont'd on next page

Example 5–4 (Cont.): Synchronizing Job Execution with Semaphores

```
BEGIN
  job_number := PROGRAM_ARGUMENT(3);  { Get the job number. }

  {
  {  Map the area of size 0 for the cr_area_sema_prog2 job.
  {}

  CREATE_AREA_SEMAPHORE(
                resource,              { Longword containing area ID }
                resource_ptr,          { Data pointer                }
                'Shared_Resource',     { String - name of area       }
                1,                     { Initial count               }
                2,                     { Maximum count               }
                completion_status);    { Status                      }

  FOR j := 1 TO 5 DO
    BEGIN

      {
      {  Wait on the area until the area is signaled.  If the
      {  semaphore count is greater than 0, the job gains access to
      {  the resource.  If the count is 0, the job waits.
      {}

      WAIT_ANY(resource);
                .
                .
                .
      { Determine which resource is available and use it. }
                .
                .
                .

      {
      {  Signal the area's semaphore to indicate that the resource is
      {  no longer in use.  When the area is signaled, the kernel
      {  increments the semaphore count so that another job can use
      {  the area.
      {}

      SIGNAL(resource);
    END;

  WRITELN('Job ', job_number, ' has used a resource 5 times');

  {
  {  Mark the area and its associated semaphore for deletion when it
  {  is no longer needed.
  {}
```

Example 5–4 Cont'd on next page

Example 5–4 (Cont.): Synchronizing Job Execution with Semaphores

```
  DELETE(resource);
END.
END;
```

5.4.6 Deleting Areas

You can delete an area by specifying the area's identifier in a call to the DELETE procedure. When you specify an area identifier with this procedure, it removes the calling job's reference to the area and unmaps the data from its P0 virtual address space. Any process in the job that created or mapped the area can delete it. The AREA object is not actually deleted until the last job that uses the area deletes its reference to the area.

Chapter 6

Device Handling

Device drivers are programs that control communication between application programs and external devices. In the case of realtime applications, most external devices are interrupt-driven — they communicate with the application only when they need service. A device requests service by sending an interrupt signal to the processor. The processor recognizes the signal, stops what it is doing, and services the request by executing a block of driver code called an *interrupt service routine* (ISR).

Once you decide on your application's device requirements, you build the relevant devices and drivers into your VAXELN system by specifying device characteristics on the System Builder's Device Description Menu (see the *VAXELN Development Utilities Guide*).

The VAXELN Toolkit provides a highly productive environment for developing application-specific device drivers. Using high-level languages, you can implement drivers for devices that have one or more units per controller. In addition, the toolkit supplies prototype driver code that you can study, and perhaps use, while programming device drivers.

You can design a device driver so that it executes as a job or as a callable routine. As a job, a device driver is a shareable resource available to all program images in a system. If a driver does not need to be shareable, you can code it as a callable routine. As a routine, a driver reduces overhead by eliminating job context switching.

Typically, a device driver job executes in kernel mode at a higher priority than jobs running other application programs and executes concurrently with the other jobs that use the related device.

A driver's activity depends on the characteristics and actions of the device it controls. However, you program a driver's general interface by declaring a variable of type DEVICE (which represents the hardware device) and an ISR. You then call VAXELN procedures that perform the following types of operations:

- Set up communication for I/O requests
- Associate a device with an ISR and a driver program
- Handle device interrupts
- Synchronize access to a device communication region
- Read data from and write data to a device's control status register (CSR) or data buffer

A driver's ISR provides an interface for handling device interrupts and power-failure recovery. When an interrupt occurs, the kernel executes the necessary machine instructions, and then calls the ISR to service the device. While servicing the device, the ISR communicates with the driver code by sharing an area of memory called the *communication region*. For example, an ISR might use this region to return device register data to the driver program.

A driver establishes a communication region when it creates a DEVICE object with a call to CREATE_DEVICE. All communication regions are potentially accessible to all ISRs. For example, for handling multivector devices, you can create two communication regions (with two CREATE_DEVICE calls) and then store a pointer to one region in the other's region. (For an example, see the VAXELN source module YCDRIVER.PAS.)

You synchronize a driver job's processes with an ISR by identifying a DEVICE object in calls to the WAIT and SIGNAL_DEVICE procedures. Driver processes wait on the DEVICE object while the ISR services a device interrupt. If multiple processes wait on the same object, the kernel queues them in the order in which the WAIT_ALL or WAIT_ANY procedure calls executed. Once the interrupt is serviced, the ISR satisfies the wait of the first process in the queue by signaling the DEVICE object with SIGNAL_DEVICE. This priority-based process scheduling eliminates the need for fork processing.

This chapter provides information about writing I/O device driver programs for handling device interrupts and power recovery. Specifically, the chapter explains how to do the following:

- Create and delete DEVICE objects, Section 6.1
- Handle device interrupts, Section 6.2

- Synchronize access to the device communication region, Section 6.3
- Set a driver job's processor eligibility, Section 6.4
- Read and write register data, Section 6.5
- Control DMA devices, Section 6.6
- Code VAXBI bus device drivers, Section 6.7
- Execute routines in kernel mode, Section 6.8
- Handle power-failure recovery, Section 6.9

6.1 Creating and Deleting DEVICE Objects

A device driver program, running in kernel mode, can create DEVICE objects by calling the CREATE_DEVICE procedure. The procedure associates a physical device with a driver program and an ISR. Once you create a DEVICE object, you can use its value as a binary semaphore to synchronize execution of the driver's ISR with the execution of other driver processes.

A call to CREATE_DEVICE must specify the device's name and a variable of type DEVICE that is to receive the DEVICE object's identifier. The device name must be one of the 1- to 30-character names established with the System Builder. The procedure uses the name to retrieve the device's characteristics.

The DEVICE variable can be a single variable or an array of 1 to 64 DEVICE elements. If you specify an array, the procedure creates an array of DEVICE objects and places the corresponding identifiers in the appropriate array elements.

Use an array if an ISR needs to communicate with multiple-unit devices, such as a 32-bit parallel port or dual-drive disk controller. In the case of a 32-bit parallel port, an ISR might use an array of 32 DEVICE elements to process the data that it receives on each port. Based on condition or bit information that the ISR receives, it can signal appropriate objects and make the associated driver processes eligible for execution.

A call to CREATE_DEVICE also can specify the name of the ISR that is to be associated with the DEVICE object or array of DEVICE objects. If you omit the argument, you drive the device by polling rather than with interrupts.

An optional relative vector argument specifies which vector of a multiple-interrupt-vector device should be connected to the ISR. (The base vector address appears on the System Builder's Device Characteristics Menu.) If you omit this argument, it defaults to 1 (the first vector). If you specify this argument in multiple calls to CREATE_DEVICE within a program, the vector value for each call must be unique; specifying the same value a second time causes the subsequent call to CREATE_DEVICE to return the status value KER$_DEVICE_CONNECTED. For example:

```
    .
    .
    .
CREATE_DEVICE('DUA0', first_device, VECTOR_NUMBER := 1);
    .
    .
    .
CREATE_DEVICE('DUA1', second_device, VECTOR_NUMBER := 2);
    .
    .
    .
CREATE_DEVICE('DUA2', third_device, VECTOR_NUMBER := 3);
    .
    .
    .
```

Other arguments that you can specify receive pointers to the device communication region, the first device control status register (CSR), the first adapter control register, and the interrupt vector in the system control block. In C and FORTRAN, you can also specify the size of the communication region.

You can also specify arguments that receive the device's interrupt priority level (IPL) and the name of a power-failure recovery routine. The recovery routine is called before any process or ISR is restarted if the processor enters a power-fail recovery sequence.

If your target configuration includes a VAX 8800 multiprocessor, a call to CREATE_DEVICE forces a driver job to run on the processor that handles the device's interrupts. It forces driver jobs for devices on VAXBI buses 2 and 3 to run on the primary processor and forces driver jobs for devices on VAXBI buses 0 and 1 to run on the secondary processor. If necessary, you can declare the job eligible to run on either processor with a call to KER$SET_JOB_ELIGIBILITY (see Section 6.4).

When a program is finished using a DEVICE object, it can delete the object with a call to the DELETE procedure. The kernel frees the memory used for the DEVICE object's communication region (invalidates pointers to that memory) and disconnects the ISR from the interrupt vector. Waiting processes are removed from their wait states immediately and receive the status value KER$_BAD_VALUE.

6.2 Handling Device Interrupts

After the CREATE_DEVICE procedure associates a device with an ISR, the kernel calls the ISR each time the device interrupts the processor. The ISR services the interrupt, using the device register pointer to gain access to the device registers. Typically, with devices that interrupt for several reasons, the ISR can examine the device's CSR to determine the reason for the interrupt.

An ISR uses the device communication region to supply a program with values that it receives from device registers. Only the data placed in the communication region is available to an ISR.

An ISR and the driver program synchronize their execution by waiting on and signaling a DEVICE object.

6.2.1 Waiting for an ISR to Service a Device Interrupt

Driver processes wait to be signaled while an ISR services a device interrupt. To initiate the wait, a process specifies the appropriate DEVICE identifier in a call to WAIT_ALL or WAIT_ANY. The ISR satisifies the wait when it finishes servicing the interrupt. If multiple processes are waiting on the same DEVICE object, the kernel queues them in the order in which the WAIT procedure calls execute. Once the interrupt is serviced, the ISR satisfies the wait of the first process in the queue.

6.2.2 Signaling the DEVICE Object After Service Completion

When an ISR finishes servicing a device interrupt, it signals the appropriate driver processes by specifying the appropriate DEVICE identifier in calls to SIGNAL_DEVICE. These calls unblock the processes that are waiting on the specified DEVICE object. An optional argument lets you identify an element in a DEVICE array.

6.3 Synchronizing Access to the Device Communication Region

While servicing a device, a driver program and an ISR communicate by sharing the device communication region. Since the communication region is a shared resource, access to the region must be synchronized. The driver program can synchronize access to the region by setting the processor's interrupt priority level (IPL).

VAX processors define 32 IPLs. IPL 0 is the lowest priority; IPL 31 is the highest. Table 6–1 lists the IPLs at which various system events occur.

Table 6–1: Interrupt Priority Levels

IPL (decimal)	Events
Hardware:	
31	Machine check; kernel stack not valid
30	Power failure
25–29	Processor, memory, or bus error
24	Clock (except MicroVAX, which is IPL 22)
16–23	Device IPLs, with 20–23 corresponding to UNIBUS or Q22-bus request levels 4–7, respectively
Software:	
9–15	Unused
8	DEVICE signal
7	Timer process
6	Closely coupled symmetric multiprocessing interrupt
5	Kernel debugger
4	Job scheduler
3	Process scheduler
2	Deliver asynchronous exception
1	Unused
0	User process level

You should consider a device's interrupt priority and job priority when synchronizing device driver programs. The default interrupt priority for the supplied device drivers is 5. You can change the interrupt priority for the supplied drivers and set the priority for user-written drivers by editing the value for the **Interrupt priority** entry on the System Builder's Device Description Menu. The priority values range from 4 to 7, with 4 being the highest priority. These values correspond to the VAX interrupt priority levels 14 (hexadecimal) to 17 (hexadecimal).

You can get the resulting interrupt priority by specifying a priority argument in the call to CREATE_DEVICE.

When synchronizing a device driver program, you should also consider the program's job priority. The default job priority for most supplied device drivers is 4. The default for supplied terminal drivers is 2. You can adjust the job priority for supplied drivers and set the priority for user-written drivers by calling the SET_JOB_PRIORITY procedure. However, you can use this mechanism only if you select *No* for the **Autoload driver** entry when you edit the System Builder's Device Description Menu. For more information about setting job priorities, see Section 3.3.2.

Setting the processor IPL provides synchronization because when the processor IPL is set to a certain level, interrupts assigned to that level and below (and their corresponding service routines) are disabled. This form of synchronization, though somewhat difficult to use, is efficient.

Depending on your target configuration, a driver program can use either the DISABLE_INTERRUPT and ENABLE_INTERRUPT procedures or the KER$LOCK_DEVICE and KER$UNLOCK_DEVICE procedures to raise and lower the processor's IPL. To use these procedures, the program must be running in kernel mode.

Use DISABLE_INTERRUPT and ENABLE_INTERRUPT if your target is a single processor or a VAX 8800 multiprocessor. DISABLE_INTERRUPT prevents entry to an ISR when a device interrupt occurs by raising the processor's IPL to the IPL of the device. While interrupts are disabled, no kernel procedures can be called; doing so causes unpredictable results.

For a driver job to use DISABLE_INTERRUPT on a VAX 8800 multiprocessor, the job must be running on the processor that handles the device's interrupts. If necessary, you can request specific processor eligibility while the driver job is executing by issuing a call to the KER$SET_JOB_ELIGIBILITY procedure (see Section 6.4).

To reenable device interrupts, lower the processor's IPL by calling ENABLE_INTERRUPT.

Use KER$LOCK_DEVICE and KER$UNLOCK_DEVICE if your target configuration includes a multiprocessor that lets devices interrupt any processor (such as a VAX 6000–3*nn* multiprocessor). KER$LOCK_DEVICE prevents entry to an ISR when a device interrupt occurs by raising the processor's IPL to the IPL of the device and setting a spin lock. The procedure locks out the ISR. If an interrupt for the device comes in on another processor while the spin lock is set, that processor spins on the lock until the driver clears it with a call to KER$UNLOCK_DEVICE.

NOTE

> If your target configuration may include a VAX multiprocessor that lets devices interrupt any processor, you should use the KER$LOCK_DEVICE and KER$UNLOCK_DEVICE procedures instead of DISABLE_INTERRUPT and ENABLE_INTERRUPT to synchronize the device communication region.

A VAX processor's current IPL is part of its processorwide state. Disabling interrupts of a certain priority also disables other system activities that occur at or below that priority level. If a process raises the processor's IPL to block device interrupts, that process is the only activity (other than ISRs) that can execute on that processor until the process lowers the priority by calling ENABLE_INTERRUPT or KER$UNLOCK_DEVICE.

If the power fails while interrupts are disabled, the kernel sets the IPL to 0 before it raises the exception KER$_POWER_SIGNAL. This exception is handled like other asynchronous exceptions; however, continuing from the exception with interrupts enabled may produce unpredictable results.

6.4 Setting a Driver Job's Processor Eligibility

When a device driver job executes on a VAX 8800 tightly coupled multiprocessing system, calls to the CREATE_DEVICE procedure force the driver job to run on the processor that handles the device's interrupts. This binding lets the driver job raise the processor's IPL with a call to DISABLE_INTERRUPT to synchronize access to the device communication region. If necessary, the driver job can undo this binding by calling the KER$SET_JOB_ELIGIBILITY procedure. Using a call to this procedure, a driver job can dynamically change its processor eligibility and make itself eligible to run on either processor. However, keep in mind that you cannot use an elevated IPL to synchronize access to the device communication region if the driver job is not executing on the processor that handles the device's interrupts.

When calling the KER$SET_JOB_ELIGIBILITY procedure, specify the job's new eligibility mask. The procedure replaces the eligibility mask in the job's job control block (JCB) with the mask you specify. The mask supplies Boolean values that indicate job eligibility for each processor in your target configuration. TRUE means the job is eligible to run on a processor, and FALSE means the job is not eligible to run on a processor. Whether the master process or a subprocess calls the procedure, the call changes the processor eligibility for the entire job.

At least one available processor must be eligible to run the driver job. If the job cannot run on any available processor, the kernel returns the status value KER$BAD_VALUE.

For information about synchronizing access to the device communication region, see Section 6.3.

6.5 Reading and Writing Register Data

Driver programs and ISRs can read data from and write data to device and processor registers by calling the READ_REGISTER, WRITE_REGISTER, MFPR, and MTPR routines.

The READ_REGISTER and WRITE_REGISTER routines operate on device registers. The READ_REGISTER function returns the value of a variable reference, and the WRITE_REGISTER procedure loads a value or group of values into a specified target variable reference. These read and write operations are performed by single machine instructions and are not affected by compiler optimizations. The READ_REGISTER and WRITE_REGISTER routines are the only safe methods for reading data from and writing data to a device register. These routines also can be used safely to read and write a shared variable.

READ_REGISTER and WRITE_REGISTER should always be used, instead of direct assignments, to read and write the fields in a device register. This is required because the VAX architecture does not allow the use of variable-length bit-field instructions to read or write device registers. Using READ_REGISTER and WRITE_REGISTER ensures that the compiler generates only valid instructions.

The MFPR and MTPR routines operate on processor registers. The MFPR function returns the contents of a VAX processor register. The MTPR procedure moves a specified value into a specified VAX internal processor register. To call these routines, a program must be running in kernel mode.

NOTE

Processor registers are a privileged system resource. Changing the contents of processor registers while a system is running may cause an unhandled exception.

6.6 Controlling DMA Devices

The VAXELN Toolkit provides utility procedures that device driver programs can use to perform the following direct memory access (DMA) device operations:

- Allocate, load, and free map registers, Section 6.6.1
- Allocate and free UNIBUS buffered data paths, Section 6.6.2
- Map and unmap memory buffers, Section 6.6.3
- Return a variable's physical address, Section 6.6.4

6.6.1 Allocating, Loading, and Freeing Map Registers

Device driver programs can allocate, load, and free UNIBUS or Q22-bus map registers. The KER$ALLOCATE_MAP procedure allocates a contiguous block of UNIBUS or Q22-bus map registers for use by a program to map VAX memory to UNIBUS or Q22-bus memory addresses, respectively.

The procedure returns a pointer to the first register allocated and returns the starting map register number (0 to 495 for a UNIBUS, 0 to 8175 for a Q22-bus). Optionally, the procedure returns a pointer to the base address of the system page table (SPT). Arguments supply the number of registers to allocate and the DEVICE value that identifies the device for which the registers are to be used.

Once a driver has allocated the appropriate map registers, it can call the ELN$LOAD_UNIBUS_MAP procedure to load the registers for use by a DMA UNIBUS or Q22-bus device.

The ELN$LOAD_UNIBUS_MAP procedure is an alternative to the more commonly used ELN$UNIBUS_MAP procedure.

The procedure assumes that the calling program has called the KER$ALLOCATE_MAP procedure to allocate sufficient map registers. ELN$UNIBUS_MAP allocates them for the caller. ELN$LOAD_UNIBUS_MAP also assumes that an additional map register, beyond the number actually necessary to map the buffer, has been allocated for use as an invalid *wild-transfer-stopper*.

Arguments supply a pointer to the first UNIBUS or Q22-bus map register allocated by KER$ALLOCATE_MAP, the I/O buffer, and the buffer size. An optional argument is a pointer to the SPT; if this argument is not specified, a device communication region (or any system space buffer) cannot be mapped.

Another optional argument supplies a UNIBUS data path for the transfer. If that argument is not supplied, data path 0, the direct data path, is used.

When the map registers are no longer needed, the driver program can free them by calling the ELN$FREE_MAP procedure. Pointers to the freed registers become invalid. Arguments supply the number of contiguous map registers to be freed, the number of the first register, such as the one returned by KER$ALLOCATE_MAP, and the DEVICE value that identifies the device for which the registers are freed.

The KER$ALLOCATE_MAP and KER$FREE_MAP procedures can be called only from programs running in kernel mode.

6.6.2 Allocating and Freeing Buffered Data Paths

A driver program can allocate and free UNIBUS adapter buffered data paths by calling the KER$ALLOCATE_PATH and KER$FREE_PATH procedures. The KER$ALLOCATE_PATH procedure allocates a UNIBUS adapter buffered data path for use by a DMA UNIBUS device.

The procedure returns a pointer to the allocated data path register and the allocated data path register number. An argument supplies the DEVICE value that identifies the device for which the data path is allocated.

A buffered data path can optimize the use of memory by a DMA device that performs strictly sequential address transfers. (For additional information on buffered data paths, see the *VAX Hardware Handbook.*) The VAX–11/750, and VAX 8800, 8700, 8550, 8530, and 8500 processors that are configured with UNIBUS adapters, support UNIBUS buffered data paths. For the VAX–11/750, each UNIBUS adapter has three buffered data paths. For the VAX 8*nnn* processors, each UNIBUS adapter has five buffered data paths.

To use a buffered data path for a DMA transfer, the allocated data path number must be loaded into the UNIBUS map registers being used for the transfer. The ELN$UNIBUS_MAP and ELN$LOAD_UNIBUS_MAP procedures accept an optional data path number for loading into the UNIBUS map registers.

When a UNIBUS buffered data path is used for a DMA transfer, the data path must be *purged* when the transfer has completed. You purge by writing a value of 1 to the data path register, identified by the returned register pointer.

The driver program can free allocated data paths by calling the KER$FREE_PATH procedure. Arguments supply the data path register number, such as the one returned by KER$ALLOCATE_PATH, and the DEVICE value that identifies the device for which the data path is freed.

The KER$ALLOCATE_PATH and KER$FREE_PATH procedures can be called only from programs running in kernel mode.

6.6.3 Mapping and Unmapping Memory Buffers

Device driver programs can map and free memory buffers for DMA operations on UNIBUS and Q22-bus devices by calling the ELN$UNIBUS_MAP and ELN$UNIBUS_UNMAP procedures, respectively. The ELN$UNIBUS_MAP procedure maps a specified buffer into UNIBUS or Q22-bus address space and returns the 18-bit UNIBUS address or the 22-bit Q22-bus address of the mapped buffer.

Arguments supply the DEVICE value identifying the device that will use the mapped memory, the I/O buffer, and the buffer size. An optional argument specifies the UNIBUS data path to use; the default is 0, specifying the direct data path.

NOTE

> The procedure allocates the correct number of map registers by calling KER$ALLOCATE_MAP. The procedure then converts the virtual address of each page of the buffer to a physical address and stores and validates the physical page numbers in the allocated map registers. If a data path other than 0 is specified, it is stored in the map registers as well. Although the map registers are allocated by ELN$UNIBUS_MAP before use, a nonzero data path number is assumed not to be in use by any other device.

When the driver program no longer needs the memory buffers, it can free them by calling ELN$UNIBUS_UNMAP. This procedure unmaps previously mapped memory buffers. The procedure deallocates the correct number of map registers by calling KER$FREE_MAP.

Arguments supply the DEVICE value identifying the device that was using the mapped memory, the I/O buffer and the buffer size, and the 18-bit UNIBUS address or the 22-bit Q22-bus address of the mapped buffer.

6.6.4 Returning a Variable's Physical Address

A device driver program can use the ELN$PHYSICAL_ADDRESS function for DMA devices on MicroVAX processors to return the physical address of an identified variable. Programs using this function must include the module $PHYSICAL_ADDRESS.

6.7 Coding VAXBI Bus Device Drivers

The VAXELN Toolkit provides the utility procedures ELN$BI_NODE_MASK and ELN$BI_STOP for coding device drivers that interface with a VAXBI bus. A VAXBI device driver must call the ELN$BI_NODE_MASK procedure to get the mask identifying the VAXBI node number to which the device should direct its inputs. The driver must load the returned identification into the device's INTR Destination Register. (The alternative, hard-coding the mask, limits the driver's portability.)

For example, in a VAX 8800 or VAX 6000–2*nn* system, the returned mask has a bit set for the VAXBI node number of the NBIB/XBIB bus adapter. In a KA800 system, the mask has a bit set for the processor's VAXBI node number.

The ELN$BI_STOP procedure issues a VAXBI STOP bus transaction to place a device in a stopped node state. The procedure's meaning and usefulness for a device depends on the device.

Pascal and C programs that use these procedures must include the modules $VAXBI and **$vaxelnc**, respectively.

6.8 Executing Routines in Kernel Mode

A number of VAXELN routines must execute in kernel mode. If a program includes a call to one of these routines or a user-declared routine that requires kernel mode, you have two options. You can execute the entire program in kernel mode, or you can use the KER$ENTER_KERNEL_CONTEXT procedure to execute only that routine in kernel mode.

To execute an entire program in kernel mode, select kernel mode when you build the program into your VAXELN system or when you load the program image. Device driver programs are typical examples of entire programs that run in kernel mode.

If it is not desirable to execute an entire program in kernel mode, use calls to KER$ENTER_KERNEL_CONTEXT to execute specific routines in kernel mode; the rest of the program runs in user mode. You specify the KER$ENTER_KERNEL_CONTEXT procedure with the address of the routine that is to be called in kernel mode. You can also specify a status argument and the address of a VAX argument list to be passed to the called routine. The argument list is a block of longwords in standard VAX format: the first byte of the first longword supplies the argument count, and the block contains an additional longword for each of the arguments.

VAXELN routines that require kernel mode include most of the VAXELN driver utility procedures and the following:

ALLOCATE_MEMORY (with the *physical_address* argument)
CREATE_DEVICE
DISABLE_INTERRUPT
ELN$LOAD_UNIBUS_MAP
ENABLE_INTERRUPT
KER$ALLOCATE_MAP
KER$ALLOCATE_SYSTEM_REGION
KER$FREE_MAP
KER$FREE_SYSTEM_REGION
KER$LOCK_DEVICE

KER$UNLOCK_DEVICE
MFPR
MTPR

Example 6–1 uses KER$ENTER_KERNEL_CONTEXT to execute a function that calls DISABLE_INTERRUPT and ENABLE_INTERRUPT.

The call to KER$ENTER_KERNEL_CONTEXT in Example 6–1 establishes the kernel context needed to execute calls to DISABLE_INTERRUPT and ENABLE_INTERRUPT. It replaces a function call that might otherwise appear as follows:

```
return_status := raise_ipl(4);
```

Each argument in the call to KER$ENTER_KERNEL_CONTEXT corresponds to the components of the preceding function call. The first argument in the call to KER$ENTER_KERNEL_CONTEXT, *return_status*, receives the function's completion status, assuming the function is returning an integer status value. (If the KER$ENTER_KERNEL_CONTEXT procedure cannot access a specified argument, the procedure returns the status KER$_NO_ACCESS to *return_status*.) The second and third arguments identify the function and its arguments, respectively.

NOTE

When you call KER$ENTER_KERNEL_CONTEXT, the kernel checks for a completion status. Therefore, you must specify the KER$ENTER_KERNEL_CONTEXT procedure's status argument. If the specified routine is a function, alternatively, that function can explicitly return a status value. If you do not specify the status argument in the call to KER$ENTER_KERNEL_CONTEXT or a function that returns a status value, the call to KER$ENTER_KERNEL_CONTEXT may produce unpredictable results.

Example 6–1: Using the KER$ENTER_KERNEL_CONTEXT Procedure

```
MODULE kernel_context_example;

INCLUDE $KERNEL;

TYPE
   argument_block_type = RECORD
                         argument_count : INTEGER;
                         priority : INTEGER;
                         END;

VAR
   argument_block : argument_block_type;
   return_status  : INTEGER;

PROGRAM change_context(INPUT, OUTPUT);

BEGIN
   .
   .
   .
  argument_block.argument_count := 1;

  argument_block.priority := 4;                   { Priority }
  WRITELN('Entering kernel context...');
  KER$ENTER_KERNEL_CONTEXT(return_status,         { Routine return status }
                           ADDRESS(raise_ipl),    { Routine to execute }
                           ADDRESS(argument_block));  { Routine args }
  WRITELN('Exiting kernel context...');
   .
   .
   .
END.

FUNCTION raise_ipl(priority : INTEGER) : INTEGER;

{ While in kernel mode, raise the processor's IPL to value of priority. }

BEGIN
  WRITELN('In kernel context...');
  DISABLE_INTERRUPT(priority);       { Raise the IPL }
  ENABLE_INTERRUPT;                  { Lower IPL to 0 }
  raise_ipl := 1;                    { Returned in return_status}
END;

END; {MODULE kernel_context_example}
```

6.9 Handling Power-Failure Recovery

Devices normally need special attention following a power failure. When the necessary speed and synchronization requirements cannot be met by the general power-recovery exception (KER$_POWER_SIGNAL), you can specify, in a CREATE_DEVICE call, the name of an ISR that is to be called when the processor enters its power-failure recovery sequence. Such a routine is called before any other process or ordinary ISR is restarted. Typically, for a processor to recover from a power failure, an application must perform the following sequence of operations:

1. Reinitialize the device controller to a known state.
2. Ensure that no partially completed I/O operations are started, since the device has been reinitialized.
3. Signal processes that are waiting for device interrupts, since no interrupts will occur now that the device has been reinitialized.

These operations can be performed by a power-failure recovery routine. Since power-failure recovery occurs at unpredictable times, the ISR and main program must synchronize themselves with the action of the power-failure recovery routine to retry operations that were in progress.

The VAX architecture defines a power-failure interrupt at IPL 30 (see Table 6–1). Therefore, a process can set the processor's IPL to 30 and block the interrupt, allowing the process to synchronize itself with the power-failure recovery routine. Once a power-failure interrupt has been posted, the processor has approximately 4 milliseconds before power is shut down. So the interrupt should not be disabled for more than a few instructions.

Chapter 7

Exception Handling

This chapter discusses VAXELN exceptions and exception-handling procedures. The chapter discusses the following topics:

- VAX stack architecture, Section 7.1
- Exceptions in VAXELN systems, Section 7.2
- Raising exceptions, Section 7.3
- Exception-handling procedures, Section 7.4
- Status codes, Section 7.5
- Using runtime messages in application programs, Section 7.6

For language-specific information concerning exception handling, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual.*

7.1 VAX Stack Architecture

This section contains a brief review of the VAX stack architecture.

Whenever a program is executing on a VAX processor, the stack pointer (SP) and frame pointer (FP) hardware registers describe an active stack environment. The system software always sets up the initial stack environment for a process. Usually the memory for the stack is in the high virtual addresses of the process's memory, the P1 region. (See Chapter 3 for a discussion of VAX memory management and the definition of the P1 region.)

Stacks are good structures to record items in a defined order and then play the items back in the reverse order. Stacks are helpful in performing recursive operations, but in many cases they are best used as a record of the implicit state of a program. The call history of the procedures activated up to a point in the program is a typical application of this stack feature.

The VAX architecture uses the stack environment in the processing of many VAX instructions. The simple cases are instructions such as PUSHAL, which pushes an address on the stack. The action of *pushing* is a 2-step process: subtract a constant from the SP register, then use the new SP value as the address at which to place the data. *Popping* the stack is the reverse: use the value of SP to address the data, then add a constant to the stack.

The *constant* is dependent upon the operation. For PUSHAL, a longword is placed on the stack. In other contexts, different-sized objects are pushed or popped from the stack. VAX stacks grow downward in address as they expand. Nothing can be assumed regarding the alignment of SP on a particular memory-length boundary, although some instructions, such as CALL, implicitly align the stack. Most high-level languages manage the stack environment for the programmer; it is not necessary to manipulate the SP value explicitly.

At any given time, the value in the FP register contains the address of the active stack *call frame*, a small data structure, defined by the VAX hardware, that contains information about the current procedure invocation and the state of the procedure that called it. At the same time, the value of the SP register is equal to or less than (that is, below) the FP value. The memory between the SP and FP values is referred to as the *local storage* of the procedure activation; together, the SP and FP values are referred to as the procedure's *stack frame*, as illustrated in Figure 7–1. Pascal and C use this space to store procedure temporaries or variables.

The VAX CALLS, CALLG, and RET instructions affect the values of SP and FP to dynamically create and destroy the frame structure. For instance, with the stack in the state pictured in Figure 7–1, if a procedure call is performed, the stack would look like Figure 7–2.

Figure 7–1: A Procedure's Stack Frame

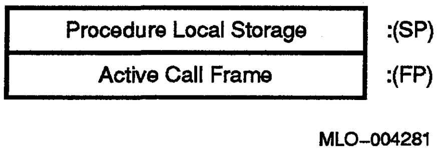
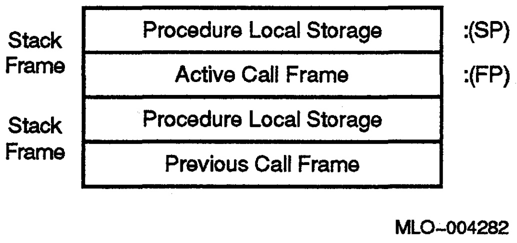


Figure 7–2: A Frame Structure After a Procedure Call

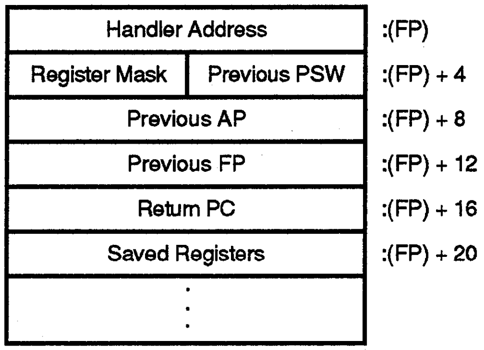


Internally, the call frame block looks like Figure 7–3.

In Figure 7–3, *Return PC* contains the address of the first instruction after the CALL instruction that called this currently active procedure. *Previous FP* contains the address of the previously active frame. The *Handler Address* location is either 0 or the address of an established condition handler procedure. (For a more detailed description of the frame contents, see the *VAX Architecture Reference Manual.*)

By examining the current frame at the FP address, the history of the call sequence can be extracted by following the *Previous FP* values until the top of the stack is reached. This trail of frames is the key to understanding what happens when an exception occurs.

Figure 7–3: Call Frame Block



MLO–004283

7.2 Exceptions in VAXELN Systems

The term *exception* describes programming events that occur during the execution of a program. Exceptions can be either synchronous or asynchronous:

- Synchronous exceptions occur at the same place in the program given a set of circumstances, for example, dividing by 0.
- Asynchronous exceptions are triggered by an event outside the control of the program, for example, power failure.

Some exceptions are generated by hardware events, and some are solely the result of a software event. VAXELN programs can experience these types of exceptions:

- Hardware-detected arithmetic problems, for instance, division by 0 or integer overflow
- Hardware-detected access problems, for instance, nonexistent memory
- Hardware-detected events, for instance, power failure
- Software-detected events, for instance, a signal of a process
- Software-detected conditions, for instance, a Pascal range violation

- Software-detected conditions in the runtime library, for instance, a problem with opening a file
- Software-detected conditions in the VAXELN Kernel when a program has requested a kernel service that must return an error status but the program did not specify a status parameter

When an exception occurs, you have two options: ignore it or handle it. An exception might or might not be important for a program and it might or might not be expected. You must decide if a particular problem or exception condition is important or fatal to the program execution.

The VAXELN Kernel exception-processing software notifies a running program of an exception by temporarily stopping the normal execution of the program and calling a specially defined *exception handler* routine defined by the program. Exception handlers are procedures that are established during the execution of a program to handle one or more of the potential exception conditions that can occur. For example, a programmer might know that an integer overflow could occur during a particular section of code and establish a special handler for that region.

All of the VAX programming languages allow the programmer to dynamically establish exception or condition handlers. For transportability, the VAXELN exception mechanism is almost identical to the VMS exception mechanism.

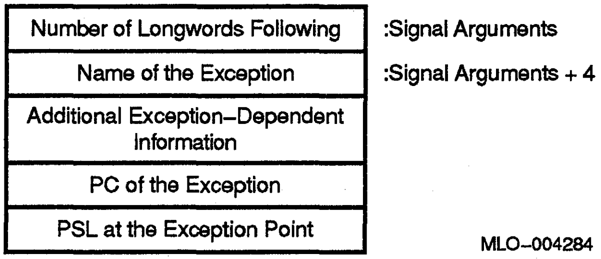
7.2.1 Exception-Handler Arguments

When an exception occurs, the VAXELN Kernel exception logic builds an argument list that describes the exception. The kernel then searches the current list of stack frames to find a frame that contains a nonzero condition handler address. When one is found, the handler procedure is called.

If no handler is found, the kernel takes a default action. If the debugger is present in the system, a special debugger handler is called. The debugger handler acts as the condition handler, giving the programmer a chance to look at the state of the program. If no handler is found and the debugger is not present, the kernel deletes the process.

The argument list for an exception handler routine contains two values. The first argument value is the address of another data block that contains information about the exception that occurred. This block is the *signal* argument block. The signal arguments are illustrated in Figure 7–4.

Figure 7–4: Signal Arguments


Number of Longwords Following
:Signal Arguments
Name of the Exception
:Signal Arguments + 4
Additional Exception–Dependent Information
PC of the Exception
PSL at the Exception Point
MLO–004284

Each exception has a distinct argument list that provides information about the exception. Sometimes, as in the case of division by 0, no additional information is needed or present. Such exceptions have the same names as the corresponding status values, as described in Appendix A, Status Values/Exception Names.

The second argument value is the address of a data block that contains information needed to recover from the exception. This block is called the *mechanism* argument block. The mechanism arguments are illustrated in Figure 7–5.

The frame depth value is the number of frames searched while the system is looking for the exception handler address.

Figure 7–5: Mechanism Arguments

4	:Mechanism Arguments
FP of Established Handler	:Mechanism Arguments + 4
Frame Depth	:Mechanism Arguments + 8
R0 at Exception	:Mechanism Arguments + 12
R1 at Exception	:Mechanism Arguments + 16

MLO–004285

7.2.2 Continue and Resignal Operations

When the exception handler routine is called, it has the responsibility of looking at the exception name value and deciding what to do. The routine then returns a Boolean value to the kernel exception handler logic. If the Boolean value is TRUE (low bit of R0 = 1) the kernel resumes execution of the program at the point of the exception; the condition is handled. If the Boolean value is FALSE (low bit of R0 = 0) the kernel continues to search the stack frame list for another handler to call; the condition is not handled. These two actions are referred to as *continuing* and *resignaling*.

Many high-level languages provide an explicit method for exiting a routine, such as an up-level GOTO in Pascal and the **longjmp** function in C, which you should use to exit an exception handler. When you use a Pascal up-level GOTO or a C **longjmp**, the language runtime library does an implicit continue on behalf of the program.

As explained previously, if no handler is found that handles the exception, the kernel deletes the process and returns the exception name as the status. Each potential exception has an individual status code defined for it (see Section 7.5). The exception name value can be used to associate a descriptive text message with the status code, as explained in Section 7.6.

An exception handler may handle one or more individual exception conditions. Some programs have handlers that handle all exceptions and display a message if something unexpected occurs. Since the stack frame is searched backward in the call history, a handler established in the program's main routine would be the last to be called in the event of an exception and could act as the *catch-all* handler.

In addition to the typical continue or resignal options, the program can also modify the exception state information and continue under different conditions. For instance, if an integer overflow occurs on a statement, the handler can modify the variables involved and continue. As another example, changing the value of the saved PC in the signal argument list has the effect of continuing the program at a different place. Remember, though, that the program continues with the stack state as it was at the exception. This means that the new PC must be in the routine that experienced the exception.

7.2.3 Unwind Operation

As mentioned previously, some languages provide an explicit method for exiting the condition handler. Using such a method has the effect of continuing at a different location and possibly in a different stack environment. The act of exiting cleanly from one stack environment and reestablishing another stack environment is called *unwinding*. Because the stack discipline and modification are complex, a VAXELN Kernel procedure performs the unwinding operation. Normally, the unwind occurs automatically when an application exits an exception handler.

If you use the KER$UNWIND procedure directly, it provides several options. You can specify KER$UNWIND with two parameters: a new frame pointer (FP) and an optional new program counter (PC) . The new FP argument specifies the target FP to which the stack will be unwound, or a value in the range 0 to 32767 that specifies the number of stack frames to be unwound (the *frame depth*). You can use a frame depth value only if you call KER$UNWIND from an exception handler or a routine called by an exception handler. Otherwise, the status value/exception SS$_NOSIGNAL is returned.

When specified as frame depths, the values 0 and −1 have special meanings. The value 0 causes KER$UNWIND to unwind to the frame of the caller of the routine that established the handler. If you specify −1 for the frame depth, KER$UNWIND does not unwind any call frames.

The PC argument specifies the new PC at which execution should resume within the call frame specified by the new FP argument. If you do not specify a new PC, the kernel uses the return PC that is already established for the target call frame.

The KER$UNWIND procedure has the effect of returning back through some number of subroutines without executing any code in the subroutines that are skipped.

Unwinding allows a program to handle the exception by skipping back to a particular call point in the stack history, for instance, the caller of the routine that got the exception.

As an unwind operation takes place, if a frame has a handler established, the handler is called with a special *unwind* exception condition. This exception is to notify the handler that the active frame is being skipped and that any necessary cleanup should be performed. The unwind handler is assumed to complete, returning the Boolean TRUE value that specifies *continue*.

One final feature can be used when an unwind is performed. Most procedures that return a simple value return that value in R0 and R1. Most VAX languages adhere to this standard. You can, therefore, change the value of the saved R0 and R1 in the mechanism argument block and then unwind. The effect is to set the value of a function and return.

The following program calls procedures to a frame depth of three. The procedure at level three establishes an exception handler that unwinds the call stack two frames to the main procedure level.

```
MODULE handler_test;

INCLUDE $KERNEL, $SSMSG;

PROGRAM handler_test;

FUNCTION cond_handler OF TYPE EXCEPTION_HANDLER;

VAR
  status : INTEGER;
  unwind_depth : ^ANYTYPE;
```

```
BEGIN
  IF SIGNAL_ARGS.NAME <> SS$_UNWIND THEN
    BEGIN
      WRITELN ('Inside condition handler.');
      WRITELN ('Unwinding ', mech_args.depth:1, ' frames');
      WRITELN ('to return to the main procedure.');
      unwind_depth::INTEGER := mech_args.depth;
      KER$UNWIND(status,
                 NEW_FP := unwind_depth);
      cond_handler := TRUE
    END
  ELSE
    cond_handler := FALSE
END;

PROCEDURE level_3;

VAR
  i, j, k : INTEGER;

BEGIN
  WRITELN ('Process is in level-3.');
  WRITELN ('Now raising an exception to test KER$UNWIND.');
  RAISE_EXCEPTION (1);
  WRITELN ('Should not execute this statement.');
END;

PROCEDURE level_2;

BEGIN
  WRITELN ('Process is in level-2.');
  WRITELN ('Now calling level-3.' );
  level_3;
  WRITELN ('Should not execute this statement.' );
END;

BEGIN
  WRITELN ('This is the main program -- level 1.');
  WRITELN ('Establishing a condition handler.');
  ESTABLISH (cond_handler);
  WRITELN ('Now calling level-2.' );
  level_2;
  WRITELN ('Control is back in level 1 after the unwind.' );
  WRITELN ('Main routine is done.');
END.
END;
```

In the preceding example, active handlers are called during the unwind operation. Thus, the handler in the example checks whether an unwind operation is already in progress. If not, the handler calls KER$UNWIND. If an unwind operation is in progress, the handler resignals. When the unwind operation is complete, control returns to the main routine.

7.2.4 Multiple Concurrent Exceptions

When an exception signal is in progress, other exceptions can still occur. These exceptions also cause the stack to be searched for an active handler, but a special action takes place. Any frames that were previously tested for having an exception handler are not tested again.

That is, when the exception occurs, the frames from the exception frame through the original condition handler are tested, then the frames between the handler's frame and the frame that activated the handler are skipped. The search resumes with the frame preceding the one that established the handler. This prevents handlers from being recursively entered; once active, a handler cannot be reactivated.

7.3 Raising Exceptions

VAXELN provides the RAISE_EXCEPTION kernel procedure, which can be used to generate exceptions. The result is much like an exception caused by a hardware condition. Sections 7.3.1 and 7.3.2 provide information about kernel procedure failure exceptions and asynchronous exceptions, respectively.

7.3.1 Kernel Procedure Failure Exceptions

Each VAXELN Kernel procedure accepts an optional status variable. The final status of the operation is placed in the variable as one of the last things done by the kernel procedure. If the program does not specify a status variable and the status is some sort of failure, an exception is generated, with the status as the exception name. This feature provides a means of handling unexpected failures for the programmer who expects kernel procedures to succeed.

7.3.2 Asynchronous Exceptions

Asynchronous exceptions do not occur as a result of a program action but as a result of an external event that cannot be predicted. The result of an asynchronous exception is identical to that of any other exception, with one notable difference. While one of these exceptions is signaled, other asynchronous exceptions are prevented from occurring until a handler returns the BOOLEAN TRUE value that specifies *continue*. However, other synchronous exceptions can still occur.

In addition, VAXELN provides two kernel procedures for controlling the occurrence of these exceptions. Normally the exceptions are enabled, but calling DISABLE_ASYNCH_EXCEPTION prevents the delivery of the exceptions to the calling process until ENABLE_ASYNCH_EXCEPTION is called. These procedures mimic the action of having an asynchronous exception signal in progress.

Several types of asynchronous exceptions are generated by VAXELN:

- KER$_POWER_SIGNAL. If a job is specified during system build as desiring power-recovery signals, the kernel generates an exception when the power recovery takes place.
- KER$_QUIT_SIGNAL. Signaling a process object causes the target process to receive this exception.
- KER$_PROCESS_ATTENTION. This exception occurs when a process calls the kernel procedure KER$RAISE_PROCESS_EXCEPTION.

7.4 Exception-Handling Procedures

The kernel procedures relating to exception handling are summarized in Sections 7.4.1 to 7.4.5.

7.4.1 DISABLE_ASYNCH_EXCEPTION Procedure

DISABLE_ASYNCH_EXCEPTION prevents the delivery of asynchronous exceptions to the calling process.

7.4.2 ENABLE_ASYNCH_EXCEPTION Procedure

ENABLE_ASYNCH_EXCEPTION allows the delivery of asynchronous exceptions to the calling process. Asynchronous exceptions are enabled by default and must be reenabled only after being explicitly disabled. They also are disabled while an asynchronous exception is being handled.

7.4.3 RAISE_EXCEPTION Procedure

RAISE_EXCEPTION causes a particular software exception in the calling process. You can specify a list of 0 or more additional exception arguments, which will be made available to the exception handler in the array of additional arguments.

NOTE

Some exception names, such as SS$_ACCVIO, are used to identify specific system or hardware events (in this case, a virtual memory access violation); do not raise one of these exceptions.

7.4.4 KER$RAISE_PROCESS_EXCEPTION Procedure

KER$RAISE_PROCESS_EXCEPTION raises the asynchronous exception KER$_PROCESS_ATTENTION in the specified process.

7.4.5 KER$UNWIND Procedure

The KER$UNWIND procedure unwinds the call stack to a new location. Arguments supply the target frame pointer (FP) and the new program counter (PC) at the new FP.

7.5 Status Codes

Status codes returned by VAXELN routines follow the VAX convention in which odd-numbered integers signify success and even values failure, though not necessarily fatal. The details of the convention are as follows:

- Bits 0 to 2 define the severity: 0 means warning, 1 means success, 2 means error, 3 means informational, and 4 means severe or fatal error.
- Bits 3 to 31 of the integer form a status ID.

Typically, an informational status is similar to success but is qualified in some way. For example, a command interpreter might use it to inform a user that although a delete command was understood and processed successfully, no objects were deleted. Similarly, warning and, sometimes, error severity imply that operation of a system is still possible, whereas fatal severity implies that it is not.

NOTE

> For the exit status of a process, you can return any integer, although Digital recommends that you follow the convention just explained.

The creator of a job has the option of receiving a special *termination* message when the created job completes. This message contains an integer making up the completion, or exit, status of the created job's master process. If the master process specifies no status of its own and completes successfully, the default status code is 1.

NOTE

> The successful completion of a process can be represented by more than one exit status, for example, status code 1 or 3. Therefore, to check for success in your programs, you should check for an odd value (bits 0 to 2 equal 1 for success, or 3 for success with an informational message).

7.6 Using Runtime Messages in Application Programs

The VMS system contains message-processing features that application programs can use to perform error checking and to handle the conversion of status codes into meaningful message text. These features are supported by the VMS Message Utility and the VMS system service $GETMSG. Using the Message Utility, you can construct messages for use with your application programs. The $GETMSG system service extracts message text from system and user-created message data bases generated by the Message Utility.

The VAXELN Toolkit includes message files generated from the VMS Message Utility. You can use the contents of these files in your application programs to check and handle errors. Additionally, the toolkit provides two runtime routines that return message text associated with a status code: the system service SYS$GETMSG, which is similar to the VMS $GETMSG routine, and a high-level language equivalent named ELN$GET_STATUS_TEXT.

Sections 7.6.1 to 7.6.4 identify the VAXELN message files and explain how to create application-specific messages, use message files in application programs, and retrieve message text associated with status codes.

7.6.1 VAXELN Message Files

The VAXELN Toolkit provides message source files and object modules used by the VAXELN software components for error checking and handling. The message source files reside in the general VAXELN runtime library. They consist of message definition statements and directives that define message text, status codes, and message symbols. You may want to examine them before or use them as templates while you create your own message files.

Message object modules reside in the RTLOBJECT.OLB and RTL.OLB object module libraries. These modules are compiled message files. The RTLOBJECT message modules contain message symbols. *Message symbols* are global symbols that provide a convenient way for programs to refer to status codes (see Section 7.6.4). A message symbol consists of a prefix that identifies the facility and a symbol name that is defined in the message definition. An example of a message symbol defined for the VAXELN Kernel is KER$_NO_SUCH_PORT. The prefix is KER$_, and the message symbol is NO_SUCH_PORT.

The RTL.OLB library contains two sets of message modules that are named *facility*$MSGDEF_TEXT and *facility*$MSGDEF. The *facility*$MSGDEF_TEXT modules contain message text. You link these modules with application programs that call the ELN$GET_STATUS_TEXT procedure to access message text at runtime.

The *facility*$MSGDEF modules in the RTL.OLB library define message symbols as linker global values for use with programs written in languages other than VAXELN Pascal.

Table 7–1 summarizes the VAXELN message files.

Table 7–1: VAXELN Message Files

Source File	RTLOBJECT Module	RTL Modules	Description
CMSG.MSG	$CMSG	C$MSGDEF_TEXT C$MSGDEF	Messages generated by the VAXELN C runtime library
ELNDECW_DWTMSG.MSG		ELNDECW_DWT$MSGDEF_TEXT ELNDECW_DWT$MSGDEF	Messages generated by the VAXELN DECwindows XUI Toolkit routines
ELNDECW_XLIBMSG.MSG		ELNDECW_XLIB$MSGDEF_TEXT ELNDECW_XLIB$MSGDEF	Messages generated by the VAXELN DECwindows Xlib routines
ELNMSG.MSG	$ELNMSG	ELN$MSGDEF_TEXT ELN$MSGDEF	Messages generated by the VAXELN Pascal compiler and other runtime components
FORMSG.MSG	$FORMSG	FOR$MSGDEF_TEXT FOR$MSGDEF	FORTRAN-specific messages generated by the VAXELN FORTRAN runtime library
KERNELMSG.MSG	$KERNELMSG	KER$MSGDEF_TEXT KER$MSGDEF	Messages generated by the VAXELN Kernel
LIBMSG.MSG	$LIBMSG	LIB$MSGDEF_TEXT LIB$MSGDEF	General runtime library messages generated by the VAXELN FORTRAN runtime library

Table 7–1 (Cont.): VAXELN Message Files

Source File	RTLOBJECT Module	RTL Modules	Description
MTHMSG.MSG	$MTHMSG	MTH$MSGDEF_TEXT MTH$MSGDEF	Math runtime library messages generated by the VAXELN and VAXELN FORTRAN run-time libraries
OTSMSG.MSG	$OTSMSG	OTS$MSGDEF_TEXT OTS$MSGDEF	Language-independent run-time library messages generated by the VAXELN and VAXELN FORTRAN run-time libraries
PASCALMSG.MSG	$PASCALMSG	PAS$MSGDEF_TEXT PAS$MSGDEF	Messages generated by the VAXELN Pascal runtime library
SSMSG.MSG	$SSMSG	SS$MSGDEF_TEXT SYS$SSDEF	System Service runtime messages generated by the VMS emulation routines and other VAXELN routines
STRMSG.MSG	$STRMSG	STR$MSGDEF_TEXT STR$MSGDEF	String runtime library messages generated by the VAXELN FORTRAN run-time library

In addition to the modules listed in the preceding table, the VAXELN Toolkit includes the message image files ELNDECW_DWTMSG.EXE, ELNDECW_XLIBMSG.EXE, ELNCMSG.EXE, and ELNMSG.EXE and VAXELN Pascal compiler messages. The VAXELN installation procedure places the images ELNDECW_DWTMSG.EXE, ELNDECW_XLIBMSG.EXE, and ELNCMSG.EXE in the VAXELN directory ELN$.

The ELNDECW_DWTMSG.EXE and ELNDECW_XLIBMSG.EXE images provide message text for the VAXELN DECwindows XUI Toolkit and Xlib runtime routines. The ELNCMSG.EXE image file provides message text for the VAXELN C runtime routines.

The image file ELNMSG.EXE and the VAXELN Pascal compiler messages are used by software that runs on a VMS system. The VAXELN installation procedure places the image ELNMSG.EXE in the VMS directory SYS$MESSAGE.

7.6.2 Constructing Messages

To construct application-specific messages, do the following:

1. Create a message source.
2. Compile the source file using the VMS Message Utility.
3. Include the resulting message object module when you link your application program.

A sample message source file follows:

```
.FACILITY       RTAPPLICATION,1 /PREFIX=RTAPP$_
.SEVERITY       ERROR
SYNTAX          <Syntax error in string '!AS'>/FAO=1
ERRORS          <Errors occurred during processing>
.END
```

Consult the message source files that reside in the general runtime library for more elaborate examples.

After you create the source file, use the MESSAGE command to compile it. Specify the command in the following format:

$ MESSAGE *file-spec*[, . . .]

The default file type for message source files is MSG. The following example compiles the message source file RTAPPMSG.MSG and produces the message object module RTAPPMSG.OBJ:

```
$ MESSAGE RTAPPMSG
```

You can then link the message object file with your application program. For example:

```
$ LINK/NOSYSSHR RTAPPLICATION+RTAPPMSG+ELN$:RTLSHARE/LIB+RTL/LIB
```

For more information about the VMS Message Utility, see the *VMS Message Utility Manual*.

7.6.3 Using Message Files with Application Programs

You can use VAXELN and application-specific message symbols in your application programs to check for and handle various conditions at runtime. A program can compare a message symbol with a status value returned by a routine call to check whether an operation completed successfully or whether a particular error occurred.

To use message symbols, a program must import them with a language-dependent include statement. Alternatively, you can include message symbols by specifying a message module, such as KER$MSGDEF_TEXT, from RTL.OLB when you link the application. You can include the same set of object modules for each application program or you can set up the application such that all jobs share the message text shareable image ELN$:SHARED_STATUS_TEXT.EXE. This shareable image contains the status text for the following toolkit components:

- Kernel
- VAXELN runtime library
- C runtime library
- VAXELN Pascal runtime library
- FORTRAN runtime library
- General runtime library (LIB)
- Language-independent runtime library (OTS)
- String runtime library (STR)

For information about building the message text shareable image into a VAXELN system or tailoring the shareable image, see the *VAXELN Development Utilities Guide*.

Example 7–1 imports the message symbols from the message module $KERNELMSG. The program then uses the symbols KER$_SUCCESS and KER$_DISCONNECT to check for success and error conditions.

Example 7–1: Using Message Files

```
MODULE msg_symbol_ex;

INCLUDE $KERNELMSG;    { Import the kernel message symbols. }

PROGRAM use_msg_symbol(INPUT, OUTPUT);

VAR
  one_second : LARGE_INTEGER;
  data_port : PORT;
  dest_port_name : VARYING_STRING(8);
  msg : MESSAGE;
  stat : INTEGER;
   .
   .
   .
BEGIN
   .
   .
   .
  { Create a port and then use that port to establish a connection to
  { another port.  Repeat the connection request until a connection
  { is made. Use the KER$_SUCCESS message symbol to check for success.
  {}

  CREATE_PORT(data_port);
  REPEAT
    WAIT_ANY(TIME := one_second);
    CONNECT_CIRCUIT(data_port,
                    DESTINATION_NAME := dest_port_name,
                    STATUS := stat);
  UNTIL stat := KER$_SUCCESS;

  { Now, send a message over the circuit. }

  SEND(msg, data_port, STATUS := stat),

  { If the send operation failed because the circuit was disconnected
  { by the partner process, reestablish a circuit connection and try
  { to send the message again. Use the KER$_DISCONNECT message symbol
  { to check for this condition.
  {}

  IF stat = KER$_DISCONNECT THEN
    BEGIN
      DISCONNECT_CICUIT(data_port);
      CONNECT_CIRCUIT(data_port,
                      DESTINATION_NAME := dest_port_name,
                      STATUS := stat);
      SEND(msg, data_port, STATUS := stat),
    END;
```

Example 7–1 Cont'd on next page

Example 7–1 (Cont.): Using Message Files

```
  { If the operation failed again, terminate this job. Use the
  { ODD function to check for an odd status code (success or
  { informational).
  {}

  IF NOT(ODD(stat)) THEN
    BEGIN
      WRITELN('Exiting, status is: ', stat:1);
      EXIT(EXIT_STATUS := stat);
    END;
   .
   .
   .
END.
END;
```

If the INCLUDE statement was omitted from this program, you could include the $KERNELMSG module with the following EPASCAL command line:

```
$ EPASCAL/DEBUG MSG_SYMBOL_EX,ELN$:RTLOBJECT/LIB/INCLUDE=$KERNELMSG
```

7.6.4 Retrieving Message Text

The VAXELN runtime libraries provide two message-processing routines: SYS$GETMSG and ELN$GET_STATUS_TEXT. These routines retrieve the message text associated with a specified status code. An application can use these routines to retrieve message text from system message files or user-created message files.

SYS$GETMSG is a system service that is similar to the VMS system service $GETMSG. It locates and returns message text associated with a specified status code into the caller's buffer. You must specify the status code, a longword to receive the message length, and buffer address arguments. Optional arguments let you specify the message components to be returned and the address of a four-byte array that receives other message-specific data. For more information, see the *VMS System Services Reference Manual.*

The VAXELN Toolkit provides the ELN$GET_STATUS_TEXT procedure for easier use with high-level languages. This procedure searches for a specified status code in the message text modules that you include with the program image. If the procedure does not find the status code in the image, the procedure searches for the code in the system's message text shareable image. When the procedure finds the specified status code, it returns the code's message text. If you specify the optional format control string, the procedure returns only the message components identified in the string.

To use the ELN$GET_STATUS_TEXT procedure, you must include the following modules:

Language	**Module**
VAXELN Pascal	$GET_MESSAGE_TEXT from the RTLOBJECT.OLB
C	$GET_MESSAGE_TEXT from ELN$:VAXELNC.TLB
FORTRAN	ELN$:MESSAGES.FOR

The call to ELN$GET_STATUS_TEXT in the following example returns the message text associated with the message symbol KER$_BAD_COUNT to the variable *output_string*:

```
VAR
  output_string
    .
    .
    .
BEGIN
    .
    .
    .
  { Get the message text. }

  ELN$GET_STATUS_TEXT(KER$_BAD_COUNT,
                      [STATUS$FACILITY,STATUS$SEVERITY,STATUS$IDENT,
                      STATUS$TEXT]
                      output_string);

  { Now write it. }

  WRITELN(output_string);
  { %KERNEL-F-BAD_COUNT, Bad parameter count
  {
  { would be written to SYS$OUTPUT
  {}
   .
   .
   .
END.
```

Since the ELN$GET_STATUS_TEXT procedure retrieves message text at runtime, the appropriate *facility*$MSGDEF_TEXT message modules must be linked with your application programs or included as part of the system's message text shareable image. You can include these message modules when you specify the RTL.OLB library module in the LINK command line, or when you select *Yes* for the **Shared status text** entry on the System Builder's System Characteristics Menu.

When using the remote debugger, you can also retrieve runtime message text by using the SHOW MESSAGE debugger command. For information on the SHOW MESSAGE debugger command, see the *VAXELN Development Utilities Guide*.

7.6.5 Displaying VAXELN Message Text on VMS Systems

While developing VAXELN applications on a VMS system, you may want the system to display the message text associated with the hexadecimal values *reason masks* or *reason values*, reported in the context of exceptions. Such values are returned by the local debugger component. To retrieve the message text, specify the appropriate message image files with the DCL command SET MESSAGE. The image files that the toolkit supplies include the following:

ELNDECW_DWTMSG.EXE	Provides message text for the VAXELN DECwindows XUI Toolkit runtime routines
ELNDECW_XLIBMSG.EXE	Provides message text for the VAXELN DECwindows Xlib runtime routines
ELNCMSG.EXE	Provides message text for the VAXELN C runtime routines
ELNMSG.EXE	Provides message text for the VAXELN runtime routines

The following command lines show how to enable message text for the VAXELN and VAXELN C runtime routines, where *hhhhhhhh* is the hexadecimal value of interest:

```
$ SET MESSAGE SYS$MESSAGE:ELNMSG
$ EXIT %xhhhhhhhh

$ SET MESSAGE ELN$:ELNCMSG
$ EXIT %xhhhhhhhh
```

You must enable message text while debugging DECwindows applications. The following command lines enable DECwindows message text, where *hhhhhhhh* is the hexadecimal value of interest:

```
$ SET MESSAGE ELN$:ELNDECW_DWTMSG
$ EXIT %xhhhhhhhh

$ SET MESSAGE ELN$:ELNDECW_XLIBMSG
$ EXIT %xhhhhhhhh
```

Chapter 8

Ethernet/IEEE 802 Datalink Drivers

The VAXELN Toolkit includes Ethernet/IEEE 802 datalink drivers for supported network devices. Each of the datalink drivers supports the the VAXELN Ethernet/IEEE 802 Datagram Service, VAXELN Network Service, and VAXELN Internet Services.

- The Datagram Service provides an interface that VAXELN systems can use to communicate with other types of systems using system-independent communications protocols.
- The Network Service routes messages sent between two network nodes, manages the list of universal names for the network, and provides a runtime interface for managing a DECnet network.
- The Internet Services provide an Ethernet network interface that VAXELN systems can use to communicate with other applications in an Internet network.

The VAXELN datalink drivers support multiple Ethernet controllers. VAXELN systems can include up to eight Ethernet controllers of the same type, and can participate in homogeneous or heterogeneous networking environments. Although DECnet software can run on only one controller at a time, you can implement other private Ethernet protocols that can run on other available controllers. That is, if your system is configured with two Ethernet controllers, one can run DECnet while the other controller runs your private Ethernet protocol.

Using network management routines, applications can start and stop the DECnet software on a controller and can switch the DECnet software from one controller to another (see Section 9.4.3).

The VAXELN Ethernet/IEEE 802 datalink drivers are self-contained program images that perform networking services. Table 8–1 lists the datalink drivers with the devices they support.

Table 8–1: Ethernet/IEEE 802 Datalink Drivers

Driver	Supported Network Devices
ESDRIVER	Integrated Ethernet controllers for the MicroVAX 2000, 3300, and 3400 and the VAXstation 2000 and 3100 series processors (ESA)
ETDRIVER	DEC LANcontroller 200 (BNI)
EZDRIVER	Integrated Ethernet controller for rtVAX 300 processor (EZA)
XBDRIVER	DEBNA VAXBI Ethernet adapter (BNT)
XEDRIVER	DELUA or DEUNA UNIBUS adapter (UNA)
XQDRIVER	DELQA or DEQNA Q-bus adapter (QNA)

You build a datalink driver into a VAXELN system image by selecting appropriate System Builder menu entries on the Network Node Characteristics Menu and the Device Description Menu. To build a datalink driver into a system for using the Network Service or Datagram Service, you must select the following values on the Network Node Characteristics Menu:

- *Enabled* or *Disabled* for the **Network service** entry
- The **Network device** entry value that indicates the type of network device on your system

If you select *Enabled*, the System Builder includes the Network Service in your system such that it runs at system start-up. If you select *Disabled*, the System Builder includes the Network Service in your system, but the service remains idle until the system enables it dynamically at runtime (see Section 9.4.3.1).

When you finish editing the Network Node Characteristics Menu, the System Builder prompts you for a device configuration by displaying the Device Description Characteristics Menu.

Applications that do not require VAXELN DECnet services but require other network services, such as the Datagram Service or Internet Services, must include the appropriate VAXELN datalink driver and device description to support these alternative network protocols. To add Ethernet device support for such an application, you must edit the Network Node Characteristics and Device Description menus as follows:

- Select *No* for the **Network service**, **Name service**, and **File access listener** entries on the Network Node Characteristics Menu.
- Add a device description for the system's Ethernet controller on the Device Description Menu, as follows:
  - Name the device according to the device tables in the *VAXELN Development Utilities Guide*, using the form X*n*B. In other words, do not use the letter A as the third letter in the device name. Instead, use the letter B (or any other letter). For example, instead of specifying the device name XQA, specify the name as XQB; instead of specifying the name XBA, specify XBB.

    Not using a device name in the form X*n*A prevents the System Builder from inappropriately deleting the device description if you reedit the Network Node Characteristics Menu. The System Builder treats device names in the form X*n*A specially.

    If your system contains multiple Ethernet controllers, name subsequent controllers beginning with the letter C, and specify *No* for the **Autoload** entry.
  - Use the appropriate values as specified in the device tables for the **Register address**, **Vector address**, and **Interrupt priority** entries.
  - Select *Yes* for the **Autoload** entry. This selection allows the System Builder to load the correct Ethernet datalink driver into the system image.

After building the system image, you can examine the map file for the system to confirm that the correct device driver was included in the system image.

NOTE

If you build the ESDRIVER into a VAXELN system that is to run on a MicroVAX 3300 or 3400 processor, you must specify an additional 128 pages for the system's system region size.

The System Builder includes the Network Service in an enabled state implicitly when you select the remote debugging option for a system under development.

This chapter provides an overview of the Ethernet/IEEE Datagram Service (see Section 8.1) and explains how an application can use the Datagram Service to do the following:

- Retrieve a CSMA/CD LAN configuration, Section 8.2
- Retrieve Ethernet controller attributes, Section 8.3
- Connect and disconnect an Ethernet/IEEE 802 protocol, Section 8.4
- Transmit and receive messages, Section 8.5
- Set up an Ethernet/IEEE 802 Datagram Service environment, Section 8.6

For more information about building the datalink drivers into VAXELN systems, see the *VAXELN Development Utilities Guide*. For more information about the VAXELN Network Service, see Chapter 9. For more information about the VAXELN Internet Services, see Chapter 10.

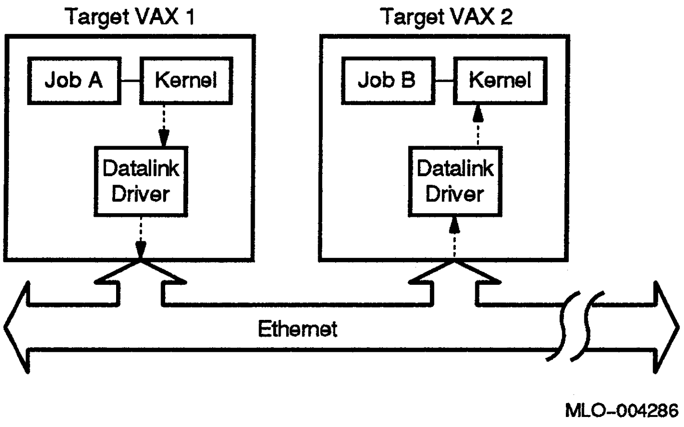
8.1 Ethernet/IEEE 802 Datagram Service

VAXELN systems cannot exchange user-level datagrams transparently with other operating systems. However, two such systems can communicate in a nontransparent manner by using the VAXELN Ethernet/IEEE 802 Datagram Service. This service provides network interface routines that VAXELN application programs can use to communicate over a Carrier Sense Multiple Access/Collision Detect (CSMA/CD) LAN. Using this service, VAXELN systems can communicate with other types of systems using system-independent communications protocols. The systems send messages to and receive messages from the datalink driver without Network Service intervention.

The VAXELN and VMS datalink drivers provide multiplexing through the Ethernet protocol type, IEEE 802 service access points (SAPs), and the IEEE SNAP SAP with a protocol identification (extended version of SAP), that provides access to multiple users.

Figure 8–1 shows a 2-node VAXELN network that is using the Ethernet/IEEE 802 Datagram Service.

Figure 8–1: **A Two-Node VAXELN Network Using the Datagram Service**



Job A transmits a message to Job B by using the datalink driver's network interface routines. The routines retrieve the CSMA/CD LAN configuration, connect to an Ethernet protocol, allocate a buffer for transmitting the message, and then transmit the message. Job B, on Target VAX 2, waits on a dispatch port. When a message arrives, the job calls a routine that receives the message.

Ethernet/IEEE 802 Datagram Service provides the following network interface routines:

Routine	Description
ELN$NI_ALLOCATE_BUFFER	Allocates a buffer for transmitting a message over a CSMA/CD LAN.
ELN$NI_CONNECT	Connects a process to an Ethernet/IEEE 802 protocol.
ELN$NI_DISCONNECT	Disconnects a process from an Ethernet/IEEE 802 protocol.
ELN$NI_GET_ATTRIBUTES	Gets information about the CSMA/CD LAN controller.
ELN$NI_GET_CONFIGURATION	Gets information about the CSMA/CD LAN configuration.
ELN$NI_RECEIVE	Receives a message from a CSMA/CD LAN dispatch port.
ELN$NI_TRANSMIT	Transmits a message over a CSMA/CD LAN.
ELN$NI_TRANSMIT_STATUS	Retrieves a message that was transmitted by a previous call to ELN$NI_TRANSMIT.

The following sections explain how to use the network interface routines to do the following:

- Retrieve the CSMA/CD LAN configuration, Section 8.2
- Retrieve Ethernet controller attributes, Section 8.3
- Connect and disconnect an Ethernet/IEEE 802 protocol, Section 8.4
- Transmit and receive messages over the CSMA/CD LAN, Section 8.5

To use the network interface routines, you must include the appropriate modules from the VAXELN runtime libraries.

Language	Module
VAXELN Pascal	$NI_UTILITY
C	$vaxelnc and $ni_utility
FORTRAN	ELN$:NI_UTILITY.FOR

NOTE

> The network interface routines are in the shareable image NISHR.EXE. If you dynamically load programs that use the network interface routines into a VAXELN system, you should specify ELN$:NISHR.EXE in the **Guaranteed image list** entry on the System Builder's System Characteristics Menu when you build that system.

For descriptions of these routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

Section 8.6 shows how to set up your application environment for Ethernet/IEEE 802 Datagram Service communication.

8.2 Retrieving a CSMA/CD LAN Configuration

To use the Ethernet/IEEE 802 Datagram Service for message communication, you must first retrieve information about the system's Ethernet controller configuration by calling the network interface routine ELN$NI_GET_CONFIGURATION. This routine stores the version number of the network interface routines and the following information for each Ethernet/IEEE 802 controller:

- Device type
- Device name
- Control port value
- Data port value

Once the configuration record is filled in, the application program can access it and retrieve data concerning the active controllers on the system.

A call to ELN$NI_GET_CONFIGURATION must specify a count argument that receives the current number of active controllers and an argument that specifies the configuration record. For example:

```
VAR
  status : INTEGER;
  config_count : INTEGER;
  config_data : ELN$NI_CONFIGURATION;
    .
    .
    .
BEGIN
    .
    .
    .
  ELN$NI_GET_CONFIGURATION(STATUS := status,
                           COUNT := config_count,
                           CONFIG := config_data);

  control_port := config_data.clist[1].control_port;
  data_port := config_data.clist[1].data_port;
    .
    .
    .
END.
```

This section of code fills in the controller configuration record and then accesses the fields containing the controller's control port and data port values. The controller count field can receive a value ranging from 1 to 8.

Sections 8.2.1 to 8.2.4 provide more information about Ethernet controller device types, names, control ports, and data ports.

8.2.1 Ethernet Controller Device Types

The VAXELN datalink drivers categorize the supported Ethernet controller devices by type. Table 8–2 lists these types with the corresponding devices.

Table 8–2: Ethernet Controller Device Types

Device Type	Supported Devices
ELN$K_NI_DEBNA	DEBNA or DEBNT VAXBI Ethernet adapter
ELN$K_NI_DEBNI	DEC LANcontroller 200
ELN$K_NI_DELQA	DELQA Q-bus adapter
ELN$K_NI_DEQNA	DEQNA Q-bus adapter
ELN$K_NI_DEUNA	DELUA or DEUNA UNIBUS adapter

Table 8–2 (Cont.): Ethernet Controller Device Types

Device Type	Supported Devices
ELN$K_NI_LANCE	Integrated Ethernet controllers for MicroVAX 2000, 3300, and 3400 and the VAXstation 2000 and 3100 series systems
ELN$K_NI_SGEC	Second generation Ethernet controller for rtVAX 300 systems

8.2.2 Ethernet Controller Device Names

Each Ethernet controller device in a CSMA/CD LAN configuration has a *device name* that consists of 1 to 32 ASCII characters. The names in a LAN must be unique within the same logical Ethernet. Examples of such names follow:

Device Type	Device Name
ELN$K_NI_DEBNA	XBA0
ELN$K_NI_DELQA	XQA0
ELN$K_NI_DEUNA	XEA0

8.2.3 Ethernet Controller Control Ports

The datalink drivers create a VAXELN control port for each CSMA/CD LAN controller. The control port provides an interface for accessing an Ethernet/IEEE 802 driver process. A program must specify a control port's value in subsequent calls to the ELN$NI_CONNECT, ELN$NI_DISCONNECT, and ELN$NI_GET_ATTRIBUTES routines.

8.2.4 Ethernet Controller Data Ports

The datalink drivers create a VAXELN data port for each CSMA/CD LAN controller. The data port receives messages that are transmitted over the LAN. A program must specify a data port's value in subsequent calls to the ELN$NI_TRANSMIT routine.

8.3 Retrieving Ethernet Controller Attributes

You can also retrieve information about each CSMA/CD LAN controller by calling the network interface routine ELN$NI_GET_ATTRIBUTES. This routine allocates a controller attributes record that stores the version number of the network interface routines and the following controller information:

- Device type
- Name
- Physical address
- Hardware address

Once the attributes record is allocated, the application program can access it and retrieve controller attributes.

A call to ELN$NI_GET_ATTRIBUTES must specify a control port returned in the controller configuration record and a pointer that is to point to the controller attributes record. For example:

```
VAR
  status : INTEGER;
  config_count : INTEGER;
  config_data : ELN$NI_CONFIGURATION;
  attributes_record : ^ELN$NI_ATTRIBUTES;
    .
    .
    .
BEGIN
    .
    .
    .
  ELN$NI_GET_CONFIGURATION(STATUS := status,
                           COUNT := config_count,
                           CONFIG := config_data);

  control_port := config_data.clist[1].control_port;
  data_port := config_data.clist[1].data_port;
    .
    .
    .
  ELN$NI_GET_ATTRIBUTES(CONTROL_PORT := control_port,
                        ATTRIBUTES_PTR := attributes_record);
  WITH attributes_record^ DO
  BEGIN
    WRITELN('Device type = ', DEV_TYPE);
    WRITELN('Device name = ', DEVICE_NAME);
    WRITELN('Physical address = ');
    FOR i := 1 TO 6 DO
      WRITE(HEX(PHYSICAL_ADDRESS::ELN$NI_DATALINK_ADDRESS_BYTE[I],2));
    WRITELN('Hardware address = ');
    FOR i := 1 TO 6 DO
      WRITE(HEX(HARDWARE_ADDRESS::ELN$NI_DATALINK_ADDRESS_BYTE[I],2));
  END;
DISPOSE(attributes_record);
    .
    .
    .
```

This section of code allocates the controller attributes record of the first Ethernet/IEEE 802 controller and then accesses the fields containing the controller's device type, name, physical address, and hardware address.

Deallocate the attributes record when the record is no longer needed.

Sections 8.2.1 and 8.2.2 provide more information about Ethernet controller device types and names. Sections 8.3.1 and 8.3.2 provide more information about Ethernet controller physical and hardware addresses.

8.3.1 Ethernet Controller Physical Addresses

An Ethernet controller's physical address is an Ethernet address that consists of 48 bits (4 bits per hex digit) and has the following format:

nn-nn-nn-nn-nn-nn

This is the format that the ELN$NI_GET_ATTRIBUTES routine uses to store a physical address in a controller's attributes record.

The controller's physical address defaults to the hardware address until DECnet starts. When DECnet software is enabled, the address is the value AA–00–04–00 followed by the DECnet node and area addresses enabled on the controller board. The AA–00–04–00 address resides in the low order 32 bits and the DECnet node and area addresses reside in the high order 16 bits.

An application that starts to run with the DECnet software disabled can set the DECnet node address and start the DECnet software dynamically by calling the ELN$NETMAN_START_NETWORK routine (see Section 9.4.3).

For more information about Ethernet controller physical addresses, see the *VAXELN Development Utilities Guide*.

8.3.2 Ethernet Controller Hardware Addresses

An Ethernet controller's hardware address is the default 48-bit address of the controller hardware. This address resides in the medium access control (MAC) address ROM on the controller. You cannot change this address.

8.4 Connecting and Disconnecting an Ethernet/IEEE 802 Protocol

Before an application program can transmit or receive datagrams over a CSMA/CD LAN, it must connect a process to an Ethernet/IEEE 802 protocol. To make this connection, the application must create a VAXELN message port and pass that port as an argument in a call to the ELN$NI_CONNECT routine. The VAXELN message port serves as a *dispatch port*, receiving data from the datalink driver. You create the dispatch port by calling the CREATE_PORT kernel procedure.

You must also specify the CSMA/CD LAN controller's control port in the call to ELN$NI_CONNECT. The Ethernet/IEEE 802 Datagram Service returns the values of the control ports for all active controllers on the system when you use the ELN$NI_GET_CONFIGURATION procedure to get the CSMA/CD LAN configuration (see Section 8.2). The ELN$NI_CONNECT routine uses the control port to pass the connection request to the datalink driver.

The call to ELN$NI_CONNECT creates a *portal*, which represents the Ethernet/IEEE 802 connection. Once the portal is established, you can transmit and receive datagrams over a CSMA/CD LAN using other network interface routines.

In addition to specifying a control port and dispatch port, a call to ELN$NI_CONNECT must specify an argument that receives an integer identifying the portal and a form argument that specifies the *message format* and the *multiplexing* data to be accepted on behalf of the portal. You can further customize a network interface connection by specifying the following:

- User data value
- Whether promiscuous mode (deliver all messages) is enabled
- Multicast count
- IEEE 802 group service access point (SAP) count
- IEEE 802 logical link control (LLC) sublayer class
- Multicast addresses
- IEEE 802 group LLC SAPs
- Whether the portal is to operate a padded Ethernet protocol

When an application program has finished using an Ethernet/IEEE 802 protocol, the program can disconnect it using a call to the ELN$NI_DISCONNECT routine. A call to ELN$NI_DISCONNECT must specify the connection's portal identification number and the control port. The portal identification must be the value that was returned by a call to ELN$NI_CONNECT. The control port must be the same control port that was used in the call to the ELN$NI_CONNECT for this portal.

The following section of code creates a dispatch port, establishes a portal in promiscuous mode, and disconnects the portal:

```
VAR
  status : INTEGER;
  config_count : INTEGER;
  config_data : ELN$NI_CONFIGURATION;
  dispatch_port : PORT;
  portal_id : INTEGER;
  format_and_mux : ELN$NI_FORMAT_AND_MUX;
  user_data : INTEGER;
  prom : BOOLEAN;
    .
    .
    .
BEGIN
    .
    .
    .
  ELN$NI_GET_CONFIGURATION(STATUS := status,
                           COUNT := config_count,
                           CONFIG := config_data);
  control_port := config_data.clist[1].control_port;
  data_port := config_data.clist[1].data_port;

  CREATE_PORT(dispatch_port);

  prom := TRUE;
  user_data := 12345;

  ELN$NI_CONNECT(STATUS := status,
                 PORTAL_ID := portal_id,
                 CONTROL_PORT := control_port,
                 DISPATCH_PORT := dispatch_port,
                 FORM := format_and_mux,
                 USER_DATA := user_data,
                 PROMISCUOUS := prom);
   .
   .
   .
  ELN$NI_DISCONNECT(STATUS := status,
                    PORTAL_ID := portal_id,
                    CONTROL_PORT := control_port);
END.
```

The following sections provide more information about the following topics:

- Portals, Section 8.4.1
- Dispatch ports, Section 8.4.2
- Message format and multiplexing, Section 8.4.3
- User data, Section 8.4.4
- Promiscuous mode, Section 8.4.5
- Multicast addresses, Section 8.4.6
- Group SAPs, Section 8.4.7
- LLC classes, Section 8.4.8
- Padded Ethernet Protocols, Section 8.4.9

8.4.1 Portals

The ELN$NI_CONNECT routine creates a portal and returns the portal's identification number if your process connects to the specified protocol successfully. The portal represents the Ethernet/IEEE 802 connection, and the unique identification number identifies that connection. You use this value to identify the connection in subsequent transmit and disconnect operations.

8.4.2 Dispatch Ports

A dispatch port is a VAXELN message port that receives messages from a VAXELN datalink driver. You create the dispatch port with a call to the CREATE_PORT kernel procedure prior to calling ELN$NI_CONNECT. If the connection is successful, the datalink driver sends messages that match the multiplexing criteria specified in the connection request to the dispatch port.

The datalink driver sends messages to the dispatch port. To receive the messages, your program must wait on the port and then call the ELN$NI_RECEIVE routine (see Section 8.5.4).

If a dispatch port reaches its message limit, the datalink driver discards new messages until the application removes messages from the port.

8.4.3 Message Format and Multiplexing

The form argument that you specify in a call to the ELN$NI_CONNECT routine is a 2-field structure that identifies the message format and multiplexing data the datalink driver is to use for a portal. You can specify the message format using one of four values: ELNK_NI_PTT, ELNK_NI_SAP, ELN$K_NI_SNAP, and ELN$K_NI_UNUSED. Table 8–3 describes the message formats that these values enable.

Table 8–3: Portal Message Formats

Format	Description
ELN$K_NI_PTT	Ethernet formatted frames. You can use a padded Ethernet protocol by specifying the pad argument in the call to ELN$NI_CONNECT. The PTT (Ethernet Protocol Type) value in the multiplexing field defines the Ethernet protocol type to be used. Only one user can use a particular protocol at any given time. If a user tries to use a busy protocol, the connection fails and the routine returns an error.

Table 8–3 (Cont.): Portal Message Formats

Format	Description
ELN$K_NI_SAP	IEEE 802 formatted frames. The DSAP (destination SAP) value in the multiplexing field identifies the SAP to be accepted. The DSAP value is an 8-bit number of which the low-order bit must be 0. The high-order 7 bits of DSAP identify the SAP. Only one user can use a particular SAP at any given time. If a user tries to use a busy SAP, the connection fails and the routine returns an error.
ELN$K_NI_SNAP	IEEE 802 format with SNAP SAP and protocol identification. This format is an extended version of the IEEE 802 SAP format. It increases the number of allowable protocols in the IEEE 802 frame format by using a 5-byte protocol identification field in addition to the SAP field during frame dispatching. When the datalink driver receives frames addressed to the SNAP SAP, it uses the 5-byte protocol identification as the filtering criteria. The PROTID value in the multiplexing field identifies the protocol identification to be accepted.
ELN$K_NI_UNUSED	No multiplexing field is specified. This format indicates that the multiplexing field is not specified for the connection request. Use this value when you want to do the following: • Use promiscuous mode (see Section 8.4.5) • Enable group SAPs without having to enable an individual SAP (see Section 8.4.7) This is the default.

You specify the type of multiplexing to be used by specifying values for the DSAP, PROTID, PTT, SAP, and SSAP fields of the form argument's multiplexing field. Table 8–4 describes the multiplexing fields.

Table 8–4: **Portal Multiplexing Fields**

Field	Description
DSAP	Destination SAP. An 8-bit value that specifies the SAP. The low-order bit must be 0. You must specify a value for this field if you specify the format ELN$K_NI_SAP.
PROTID	SNAP protocol identification. A 5-byte value that specifies an Ethernet protocol that specifies the filtering criteria. You must specify a value for this field if you specify the format ELN$K_NI_SNAP.
PTT	Ethernet Protocol Type. A value that specifies an Ethernet protocol type. You must specify a value for this field if you specify the format ELN$K_NI_PTT.
SAP	Not applicable.
SSAP	Not applicable.

You must also specify a form argument in calls to the ELN$NI_RECEIVE and ELN$NI_TRANSMIT routines. However, the ELN$K_NI_UNUSED format value does not apply. In the case of ELN$NI_RECEIVE, the argument receives the format and multiplexing information. The ELN$NI_TRANSMIT routine specifies format and multiplexing information for a particular message.

8.4.4 User Data

You can associate a user-defined integer value with a portal by specifying a user data argument in the call to ELN$NI_CONNECT. The integer value is returned with each message sent to the portal's dispatch port. You might use such data to distinguish between messages sent to a dispatch port when the same dispatch port is specified in separate calls to ELN$NI_CONNECT.

8.4.5 Promiscuous Mode

By default, the datalink driver delivers only those messages matching the multiplexing information enabled for a network interface portal. If you want the datalink driver to deliver a copy of each message transmitted on the Ethernet, you must enable *promiscuous mode* for the portal by setting the promiscuous Boolean argument in the call to ELN$NI_CONNECT to TRUE.

Only one user can use promiscuous mode at any given time. If a second user tries to enable this mode, the connection fails and the driver returns an error.

8.4.6 Multicast Addresses

A *multicast address* is a 48-bit CSMA/CD LAN destination address. If you want messages that have such addresses to be dispatched to a portal, you must enable the addresses explicitly in the call to ELN$NI_CONNECT. You can specify a set of up to eight multicast addresses for a portal. If you specify a set of multicast addresses, you indicate the number of addresses in the set by specifying a multicast address count value. By default, the physical address of the node is accepted for each portal.

8.4.7 Group SAPs

In addition to or instead of specifying a single SAP address for a portal, as you do when you use the IEEE 802 formatted frames message format, you can specify up to four *group SAPs*. The datalink driver dispatches the individual SAP and the group SAPs to the same portal.

You specify an 8-bit value for each group SAP. The low-order bit must be 1. The high-order 7 bits specify a SAP address.

If you specify group SAPs, you indicate the number of SAPs being specified by supplying a group SAP count value.

Multiple portals can enable the same group SAP.

8.4.8 LLC Classes

A portal's *LLC class* determines the types of messages that are sent to an enabled SAP. The class can be either ELN$K_NI_CL1 or ELN$K_NI_USER_SUPPLIED. If the class is ELN$K_NI_CL1 class, the datalink driver handles IEEE 802 exchange identification (XID) and test (TEST) messages and sends unnumbered information protocol (UI) messages to the user. If the class is ELN$K_NI_USER_SUPPLIED, the driver sends all messages addressed to the enabled SAP to the user. In this case, you must supply the IEEE 802 control field as part of your user data when you do a transmit operation. The IEEE 802 control field is also returned as message user data in receive operations, and you must process it. The default LLC class is ELN$K_NI_USER_SUPPLIED.

8.4.9 Padded Ethernet Protocols

Portals can operate a *padded Ethernet protocol*. When using a padded format, the datalink driver adds a padding field length of two bytes to each message that it transmits and removes that field from each message that it receives. You can enable the padded format by setting a Boolean argument in the call to ELN$NI_CONNECT to TRUE.

Keep in mind that if one user uses the padded format of a particular protocol, all users using that protocol must also use the padded format. Otherwise, the padding field may be missing or interpreted as user data.

8.5 Transmitting and Receiving Messages

Once your application program establishes a network interface connection, it can use that connection to transmit and receive messages over the CSMA/CD LAN. To transmit a message, you use the network interface routines to do the following:

- Allocate a buffer for transmitting the message (ELN$NI_ALLOCATE_BUFFER)
- Transmit the message (ELN$NI_TRANSMIT)
- Retrieve the transmitted message and status (optional) (ELN$NI_TRANSMIT_STATUS)

To receive messages, a job waits on the dispatch port. When a message arrives, the job calls the ELN$NI_RECEIVE routine to receive the message.

Sections 8.5.1 to 8.5.3 explain how to allocate a message buffer, transmit messages, and retrieve transmitted messages and status values. Section 8.5.4 explains how to receive messages.

8.5.1 Allocating a Message Buffer

Before an application program can transmit a message over the CSMA/CD LAN, the program must call the ELN$NI_ALLOCATE_BUFFER routine to allocate a buffer for the message. A call to ELN$NI_ALLOCATE_BUFFER must specify an integer indicating the number of bytes of user data to be allocated, the MESSAGE variable that is to receive the new message, and a pointer to the first byte of the data to be transmitted.

The size of the buffer that you allocate must be less than or equal to the maximum allowable amount of user data for the format in which the message is to be transmitted. Maximum allowable amounts for various formats are as follows:

Format	Maximum Size
IEEE 802 SNAP	1492 bytes
IEEE 802 with 1-byte control field	1497 bytes
IEEE 802 with 2-byte control field	1498 bytes
Ethernet protocol type (padding enabled)	1498 bytes
Ethernet protocol type (padding disabled)	1500 bytes

The data pointer argument receives a pointer to the first byte of the data to be transmitted in the buffer. The pointer is passed unmodified to ELN$NI_TRANSMIT. (Your program should not modify the pointer, just the buffer to which the pointer points.)

The call to ELN$NI_ALLOCATE_BUFFER in the following example allocates a buffer for a 36-character string:

```
TYPE
  message_1_type = STRING(36);

VAR
  status : INTEGER;
  data_pointer : ^message_1_type;
  user_data_size : INTEGER;
  msg : MESSAGE;
    .
    .
    .
BEGIN
  user_data_size := SIZE(message_1_type);
```

```
ELN$NI_ALLOCATE_BUFFER(STATUS := status,
                       USER_DATA_SIZE := user_data_size,
                       MESSAGE_OBJECT := msg,
                       DATA_POINTER := data_pointer);
  .
  .
  .
```

NOTE

If the portal is enabled with the IEEE 802 user-supplied LLC class, you must allocate space for the IEEE 802 control field in the user buffer area.

8.5.2 Transmitting Messages

After you allocate a message buffer, you can transmit messages over the CSMA/CD LAN by calling the ELN$NI_TRANSMIT routine. A call to ELN$NI_TRANSMIT must specify the following:

- The portal identification received in the call to ELN$NI_CONNECT
- A data port in the configuration record obtained by the call to ELN$NI_GET_CONFIGURATION
- The data pointer received in the call to ELN$NI_ALLOCATE_BUFFER
- The message value that was received in the call to ELN$NI_ALLOCATE_BUFFER
- The size of the message to be transmitted
- The message's destination address
- The form (message format and multiplexing values) of the message being transmitted

NOTE

The value for the size argument can be smaller than the value specified in the call to ELN$NI_ALLOCATE_BUFFER. If the value is larger, the result of the transmit operation is unpredictable.

The destination address must be a 48-bit address. It can be a multicast address or an individual address.

The form argument is a 2-field structure that identifies the message format and type of multiplexing the datalink driver is to use for the portal. As described in Section 8.4.3, you can specify the message format using one of three values: ELNK_NI_PTT, ELNK_NI_SAP, or ELN$K_NI_SNAP. You specify the type of multiplexing (message header information) to be used by specifying values for the DSAP, PROTID, PTT, and SSAP fields of the form argument's multiplexing field. The format values and multiplexing fields are described in Table 8–3 and Table 8–4.

The following section of code shows how you might transmit a message over a CSMA/CD LAN:

```
TYPE
  message_type = STRING(36);

VAR
  status : INTEGER;
  config_count : INTEGER;
  config_data : ELN$NI_CONFIGURATION;
  dispatch_port : PORT;
  portal_id : INTEGER;
  format_and_mux : ELN$NI_FORMAT_AND_MUX;
  user_data : INTEGER;
  prom : BOOLEAN;
  data_pointer : ^message_1_type;
  user_data_size : INTEGER;
  msg : MESSAGE;
  remote_address : ELN$NI_DATALINK_ADDRESS;
  sap_number : INTEGER;
    .
    .
    .
BEGIN
    .
    .
    .
  ELN$NI_GET_CONFIGURATION(STATUS := status,
                           COUNT := config_count,
                           CONFIG := config_data);

  control_port := config_data.clist[1].control_port;
  data_port := config_data.clist[1].data_port;

  CREATE_PORT(dispatch_port);

  format_and_mux.format := ELN$K_NI_SAP;
  format_and_mux.dsap := sap_number * 2;

  format_and_mux.ssap := sap_number * 2;     { Remote address filled }
                                             {   in here also.       }
  class_802 := ELN$K_NI_CL1;
```

```
    ELN$NI_CONNECT(STATUS := status,
                   PORTAL_ID := portal_id,
                   CONTROL_PORT := control_port,
                   DISPATCH_PORT := dispatch_port,
                   FORM := format_and_mux,

                   CLASS_802 := class_802);

    user_data_size := SIZE(message_1_type);

    ELN$NI_ALLOCATE_BUFFER(STATUS := status,
                           USER_DATA_SIZE := user_data_size,
                           MESSAGE_OBJECT := msg,
                           DATA_POINTER := msg_data_pointer);

    data_pointer^ := 'Transmitted message...';

    ELN$NI_TRANSMIT(STATUS := status,
                    PORTAL_ID := portal_id,
                    DATA_PORT := data_port,
                    DATA_POINTER := data_pointer,
                    MESSAGE_OBJECT := msg,
                    USER_DATA_SIZE := user_data_size,
                    DEST_ADDRESS := remote_address,
                    FORM := format_and_mux);
       .
       .
       .
END.
```

8.5.3 Retrieving Transmitted Messages

By default, the datalink drivers delete a message after a transmit operation. However, you can instruct the driver to keep these messages by specifying a reply port in the call to the ELN$NI_TRANSMIT procedure. When you specify a reply port, the datalink driver sends the message to that port upon completion of a transmit operation. You can then extract the message from the reply port with a call to the ELN$NI_TRANSMIT_STATUS routine and use the message in a subsequent call to ELN$NI_TRANSMIT or delete it. For example:

```
TYPE
  message_type = STRING(36);
```

```
VAR
  status : INTEGER;
  config_count : INTEGER;
  config_data : ELN$NI_CONFIGURATION;
  dispatch_port : PORT;
  portal_id : INTEGER;
  format_and_mux : ELN$NI_FORMAT_AND_MUX;
  user_data : INTEGER;
  prom : BOOLEAN;
  data_pointer : ^message_1_type;
  user_data_size : INTEGER;
  msg : MESSAGE;
  remote_address : ELN$NI_DATALINK_ADDRESS;
  reply_port : PORT;
  sap_number : INTEGER;

    .
    .
    .
BEGIN
    .
    .
    .
  ELN$NI_GET_CONFIGURATION(STATUS := status,
                           COUNT := config_count,
                           CONFIG := config_data);

  control_port := config_data.clist[1].control_port;
  data_port := config_data.clist[1].data_port;

  CREATE_PORT(dispatch_port);

  format_and_mux.format := ELN$K_NI_SAP;
  format_and_mux.dsap := sap_number * 2;
  class_802 := ELN$K_NI_CL1;

  ELN$NI_CONNECT(STATUS := status,
                 PORTAL_ID := portal_id,
                 CONTROL_PORT := control_port,
                 DISPATCH_PORT := dispatch_port,
                 FORM := format_and_mux,
                 CLASS_802 := class_802);

  user_data_size := SIZE(message_1_type);

  CREATE_PORT(reply_port);

  ELN$NI_ALLOCATE_BUFFER(STATUS := status,
                         USER_DATA_SIZE := user_data_size,
                         MESSAGE_OBJECT := msg,
                         DATA_POINTER := msg_data_pointer);

  data_pointer^ := 'Transmitted message...';
  ELN$NI_TRANSMIT(STATUS := status,
                  PORTAL_ID := portal_id,
                  DATA_PORT := data_port,
                  DATA_POINTER := data_pointer,
                  MESSAGE_OBJECT := msg,
                  USER_DATA_SIZE := user_data_size,
                  DEST_ADDRESS := remote_address,
                  FORM := format_and_mux
                  REPLY_PORT := reply_port);
  WAIT_ANY(reply_port);
  ELN$NI_TRANSMIT_STATUS(STATUS := status,
                         REPLY_PORT := reply_port,
                         MESSAGE_OBJECT := msg);
  DELETE(msg);
    .
    .
    .
END.
```

The call to ELN$NI_TRANSMIT_STATUS must specify the reply port and message object that were specified in the call to ELN$NI_TRANSMIT. You can also specify an argument that is to receive a pointer to the beginning of the user data portion of the message. You then can use the message object and data pointer values in a subsequent call to ELN$NI_TRANSMIT.

8.5.4 Receiving Messages

To receive a message on a CSMA/CD LAN, a program must wait on the dispatch port that was specified in the call to ELN$NI_CONNECT and then call the ELN$NI_RECEIVE routine. The ELN$NI_RECEIVE routine does the following:

- Receives a message in your program's address space
- Strips the header fields from the message and sends them back to to you as parameters
- Returns a pointer to the beginning of the message's user data
- If requested, returns the size of the message's user data

A call to ELN$NI_RECEIVE must specify the dispatch port that was specified in the call to ELN$NI_CONNECT, the MESSAGE variable that is to receive the message, and a variable that is to receive a pointer that points to the beginning of the message's user data. You can also specify variables that receive the following:

- The size of user data in the message

- The message's destination address
- The message's source address
- The message's form (message format and multiplexing type)
- User data (unique integer established in a call to ELN$NI_CONNECT)
- The portal identification whose format and multiplexing data match that of the received message

The destination and source addresses that the routine receives are 48-bit addresses.

The form argument receives a 2-field structure that identifies the message format and the message's protocol data. As described in Section 8.4.3, the message format can be one of three formats: ELNK_NI_PTT, ELNK_NI_SAP, or ELN$K_NI_SNAP. Values for the DSAP, PROTID, PTT, and SSAP fields of the form argument's multiplexing field indicate the protocol data in the message header. The format values and multiplexing fields are described in Tables 8–3 and 8–4.

The following section of code shows how an application program might receive messages over a CSMA/CD LAN:

```
TYPE
  message_type = STRING(36);

VAR
  status : INTEGER;
  config_count : INTEGER;
  config_data : ELN$NI_CONFIGURATION;
  dispatch_port : PORT;
  portal_id : INTEGER;
  format_and_mux : ELN$NI_FORMAT_AND_MUX;
  user_data : INTEGER;
  prom : BOOLEAN;
  data_pointer : ^message_1_type;
  user_data_size : INTEGER;
  msg : MESSAGE;
  reply_port : PORT;
  dest_address : ELN$NI_DATALINK_ADDRESS;
  src_address : ELN$NI_DATALINK_ADDRESS;
```

```
    .
    .
    .
BEGIN
    .
    .
    .
  ELN$NI_GET_CONFIGURATION(STATUS := status,
                         COUNT := config_count,
                         CONFIG := config_data);

  control_port := config_data.clist[1].control_port;
  data_port := config_data.clist[1].data_port;

  CREATE_PORT(dispatch_port);

  prom := TRUE;
  user_data := 12345;

  ELN$NI_CONNECT(STATUS := status,
               PORTAL_ID := portal_id,
               CONTROL_PORT := control_port,
               DISPATCH_PORT := dispatch_port,
               FORM := format_and_mux,
               USER_DATA := user_data,
               PROMISCUOUS := prom);

  WAIT_ANY(dispatch_port);

  ELN$NI_RECEIVE(STATUS := status,
               DISPATCH_PORT := dispatch_port,
               RECEIVED_MESSAGE := msg,
               DATA_POINTER := data_pointer,
               DATA_SIZE := user_data_size,
               DEST_ADDRESS := dest_address,
               SRC_ADDRESS := src_address,
               FORM := format_and_mux,
               USER_DATA := user_data);
    .
    .
    .
END.
```

8.6 Setting Up an Ethernet/IEEE 802 Datagram Service Environment

This section uses the sample application module *sample_ni_app* and callout text to illustrate the use of the network interface routines. The module consists of a main program that gets the CMSA/CD LAN configuration and calls three procedures: *get_attributes*, *transmit_msg*, and *receive_msg*.

The *get_attributes* procedure calls the ELN$NI_GET_ATTRIBUTES routine to create a controller attributes record. The procedure then accesses the record to extract the controller's device name, physical address, and hardware address.

The *transmit_msg* procedure transmits messages over a CMSA/CD LAN and retrieves the messages that are sent by doing the following:

1. Creating a dispatch port
2. Specifying the message format, multiplexing type, and class
3. Establishing a promiscuous mode portal by passing the dispatch port in a connection request
4. Creating a reply port
5. Allocating a buffer for transmitting the messages
6. Transmitting the messages
7. Waiting on the reply port
8. Retrieving the transmitted messages

The procedure also disconnects the process from the promicuous mode portal when it is finished using the connection.

The *receive_msg* procedure receives messages over a CMSA/CD LAN by doing the following:

1. Creating a dispatch port
2. Specifying the message format
3. Establishing a connection with a CSMA/CD LAN Ethernet protocol by connecting the dispatch port to a controller's control port
4. Waiting on the dispatch port
5. Receiving the messages

This procedure also disconnects the process from the CSMA/CD LAN Ethernet protocol when it is finished using the connection.

Example 8–1 shows a sample network interface application. The example assumes that an Ethernet/IEEE 802 driver is built into the VAXELN system. The discussion that follows is keyed to the numbered callouts in the example.

Example 8–1: Sample Network Interface Application

```
MODULE sample_ni_app;

{
{ This module contains the sender program for the datalink external
{ interface.
{}

INCLUDE $NI_UTILITY, $KERNELMSG, $ELNMSG;

PROGRAM ni_example;

VAR
  config_data : ELN$NI_CONFIGURATION;
  config_count : INTEGER;
  control_port : PORT;
  data_port : PORT;
  status : INTEGER;
  remote_address : ELN$NI_DATALINK_ADDRESS;
  pass_number : INTEGER;
  sap_number : INTEGER;

PROCEDURE get_attributes;

{
{  Get controller attributes.
{}

VAR
  status : INTEGER;
  i : INTEGER;
  controller_attributes_pointer : ^ELN$NI_ATTRIBUTES;

BEGIN
  ELN$NI_GET_ATTRIBUTES(STATUS := status,                        ❶
                   CONTROL_PORT := control_port,
                   ATTRIBUTES_PTR := controller_attributes_pointer);

  WITH controller_attributes_pointer^ DO
    BEGIN
      WRITELN('Device name = ', device_name);
      WRITELN('Physical address = ');
      FOR i := 1 TO 6 DO
        WRITE(HEX(PHYSICAL_ADDRESS::ELN$NI_DATALINK_ADDRESS_BYTE[I],2));
      WRITELN('Hardware address = ');
      FOR i := 1 TO 6 DO
        WRITE(HEX(HARDWARE_ADDRESS::ELN$NI_DATALINK_ADDRESS_BYTE[I],2));
    END;

  DISPOSE(controller_attributes_pointer);
END;
```

Example 8–1 Cont'd on next page

Example 8–1 (Cont.): Sample Network Interface Application

```
PROCEDURE transmit_msg;

{
{  Transmit messages using IEEE 802 formatted frames and a SAP
{  address.
{}

TYPE
  message_1_type = STRING(36);

VAR
  status : INTEGER;
  portal_id : INTEGER;
  sap_port : PORT;
  format_and_mux : ELN$NI_FORMAT_AND_MUX;
  user_data_size : INTEGER;
  msg : MESSAGE;
  data_pointer : ^message_1_type;
  reply_port : PORT;
  i : INTEGER;
  class_802 : ELN$NI_BYTE;

BEGIN
  format_and_mux.format := ELN$K_NI_SAP;                              ❷
  format_and_mux.mux.dsap := sap_number * 2;
  class_802 := ELN$K_NI_CL1;
  CREATE_PORT(sap_port);                                              ❸

  ELN$NI_CONNECT(STATUS := status,                                    ❹
                 PORTAL_ID := portal_id,
                 CONTROL_PORT := control_port,
                 DISPATCH_PORT := sap_port,
                 FORM := format_and_mux,
                 CLASS_802 := class_802);

  CREATE_PORT(reply_port);                                            ❺

  FOR i := 1 TO 100 DO
    BEGIN
      user_data_size := SIZE(message_1_type);
      ELN$NI_ALLOCATE_BUFFER(STATUS := status,                        ❻
                             USER_DATA_SIZE := user_data_size,
                             MESSAGE_OBJECT := msg,
                             DATA_POINTER := data_pointer);

      data_pointer^ := 'This is the message...';
      format_and_mux.mux.ssap := sap_number * 2;
```

Example 8–1 Cont'd on next page

Example 8–1 (Cont.): Sample Network Interface Application

```
      ELN$NI_TRANSMIT(STATUS := status,                                    ❼
                      PORTAL_ID := portal_id,
                      DATA_PORT := data_port,
                      DATA_POINTER := data_pointer,
                      MESSAGE_OBJECT := msg,
                      USER_DATA_SIZE := user_data_size,
                      DEST_ADDRESS := remote_address,
                      FORM := format_and_mux,
                      REPLY_PORT := reply_port);

      WAIT_ANY(reply_port);                                                ❽

      ELN$NI_TRANSMIT_STATUS(STATUS := status,                             ❾
                             REPLY_PORT := reply_port,
                             MESSAGE_OBJECT := msg);

      DELETE(msg);
    END;

  ELN$NI_DISCONNECT(STATUS := status,                                      ❿
                    PORTAL_ID := portal_id,
                    CONTROL_PORT := control_port);
  DELETE(sap_port);
  DELETE(reply_port);
END;

PROCEDURE receive_msg;

{
{ Receive messages that are transmitted over a CSMA/CD LAN.
{}

VAR
  dispatch_port : PORT;
  format_and_mux : ELN$NI_FORMAT_AND_MUX;
  prom : BOOLEAN;
  i : INTEGER;
  user_data_out : INTEGER;
  user_data_in : INTEGER;
  portal_id : INTEGER;
  msg : MESSAGE;
  data_pointer : ^BYTE_DATA(1500);
  data_size : INTEGER;
  da, sa : ELN$NI_DATALINK_ADDRESS;
  status : INTEGER;

BEGIN

  CREATE_PORT(dispatch_port);                                              ⓫
```

Example 8–1 Cont'd on next page

Example 8–1 (Cont.): Sample Network Interface Application

```
  format_and_mux.format := ELN$K_NI_UNUSED;                          ⓬
  prom := TRUE;
  user_data_out := 12345;

  ELN$NI_CONNECT(STATUS := status,                                    ⓭
                 PORTAL_ID := portal_id,
                 CONTROL_PORT := control_port,
                 DISPATCH_PORT := dispatch_port,
                 FORM := format_and_mux,
                 USER_DATA := user_data_out,
                 PROMISCUOUS := prom);

  FOR i := 1 to 100 DO

    BEGIN
      user_data_in := 0;

      WAIT_ANY(dispatch_port);                                        ⓮

      ELN$NI_RECEIVE(STATUS := status,                                ⓯
                     DISPATCH_PORT := dispatch_port,
                     RECEIVED_MESSAGE := msg,
                     DATA_POINTER := data_pointer,
                     DATA_SIZE := data_size,
                     DEST_ADDRESS := da,
                     SOURCE_ADDRESS := sa,
                     FORM := format_and_mux,
                     USER_DATA := user_data_in);
    END;

  ELN$NI_DISCONNECT(STATUS := status,                                 ⓰
                    PORTAL_ID := portal_id,
                    CONTROL_PORT := control_port);

  DELETE(dispatch_port);

END;

{
{  Main
{}

BEGIN

  pass_number := 0;

  REPEAT

    pass_number := pass_number + 1;
```

Example 8–1 Cont'd on next page

Example 8–1 (Cont.): Sample Network Interface Application

```
    sap_number := (sap_number + 1) MOD 128;
    IF (sap_number = ((ELN$_SNAP_SAP DIV 2) DIV 2))
      OR (sap_number = (ELN$_SNAP_SAP DIV 2))
      OR (sap_number = 0)
    THEN
      sap_number := sap_number + 1;
    WRITELN('SAP number =', sap_number);

    WRITELN('Get the configuration...');

    {
    {  Get the CSMA/CD LAN configuration and assign the control
    {  port and data port values to control_port and data_port,
    {  respectively.
    {}

    ELN$NI_GET_CONFIGURATION(STATUS := status,                          ⓱
                             COUNT := config_count,
                             CONFIG := config_data);

    control_port := config_data.clist[1].control_port;                  ⓲
    data_port := config_data.clist[1].data_port;

    WRITELN('Get the controller attributes...');
    get_attributes;

    WRITELN('Transmit messages...');
    transmit_msg;

    WRITELN('Receive messages...');
    receive_msg;

    WRITELN('Pass =', pass_number);

  UNTIL FALSE;
END.
END;
```

❶ **Get the controller attributes.** Get the controller attributes by calling the ELN$NI_GET_ATTRIBUTES routine. The routine stores the version number of the network interface routines being used and the device type, name, physical address, and hardware address of the controller whose control port value is *control_port* (see step 18). The *get_attributes* procedure then extracts the controller's device name, physical address, and hardware address. For more information about retrieving Ethernet controller attributes, see Section 8.3.

❷ **Create a dispatch port for the SAP.** Use the CREATE_PORT procedure to create a dispatch port for the SAP. The sample module creates the dispatch port *sap_port*. The module will connect the dispatch port to the first controller's control port. For more information about dispatch ports, see Section 8.4.2.

❸ **Specify the message format, multiplexing type, and LLC class.** If necessary, specify the message format, type of multiplexing, and LLC class. The sample module specifies IEEE 802 formatted frames for the message format by assigning the value ELN$K_NI_SAP to the variable *format_and_mux.format*. The value calculated from the expression *sap_number* * 2 is assigned to the DSAP multiplexing field *format_and_mux.mux.dsap*. Thus, the datalink driver will deliver messages addressed to the SAP identified by *format_and_mux.mux.dsap* to the dispatch port *dispatch_port*.

The sample module specifies that the LLC class ELN$K_NI_CL1 is to be used. This indicates that the datalink driver is to handle IEEE 802 XID and TEST messages and send UI messages to the user.

❹ **Establish the dispatch port for the portal.** Establish the dispatch port by specifying it in a call to the ELN$NI_CONNECT routine. A call to this routine must specify a portal identification number, control port, dispatch port, and message form.

The call to ELN$NI_CONNECT in the sample module establishes the dispatch port *sap_port*. The variable *portal_id* receives an integer value identifying the connection.

The *format_and_mux* and *class_802* arguments specify the message format, multiplexing type, and LLC class to be accepted on behalf of the specified portal. These values were specified in step 2.

❺ **Create a reply port.** Use the CREATE_PORT procedure to create a reply port. This port is specified in the calls to ELN$NI_TRANSMIT and ELN$NI_TRANSMIT_STATUS for returning transmitted messages and their status values.

❻ **Allocate the message buffer.** Allocate a message buffer for transmitting the messages by calling the ELN$NI_ALLOCATE_BUFFER routine. A call to this routine must specify the number of bytes of data to be allocated, a variable of type MESSAGE that is to receive the messages, and a pointer variable that receives a pointer to the first byte of the message's user data.

The sample module allocates space for a 36-character string. The arguments *msg* and *data_pointer* receive the message data and data pointer, respectively.

❼ **Transmit a message.** Transmit a message by calling the ELN$NI_TRANSMIT routine. The call to ELN$NI_TRANSMIT must specify the portal's identification number, the controller's data port, the message's data pointer and object value, the user data size, the destination address, and the form argument.

The call to ELN$NI_TRANSMIT specifies the portal identification number that the *portal_id* argument received in step 4 and the controller's data port value (see step 18). The *data_pointer* points to the beginning of the message's user data. The message data is transmitted using the SAP address that results from the computation *sap_number* * 2. The computation shifts the SAP number left by one bit, making the low-order bit 0. This indicates that the SAP is an individual DSAP.

The optional *reply_port* argument lets you retrieve the messages that are transmitted and their status values.

For more information about transmitting messages, see Section 8.5.2.

❽ **Wait on the reply port.** Wait on the reply port by specifying the port in a call to the WAIT_ANY procedure. The sample module waits on *reply_port*.

❾ **Retrieve the transmitted message.** Retrieve a message on the reply port by specifying the port in a call to the ELN$NI_TRANSMIT_STATUS routine. You must also specify the message object. The call to ELN$NI_TRANSMIT_STATUS in the sample module retrieves the message *msg* from *reply_port*.

❿ **Disconnect the portal.** Disconnect the portal after all messages are transmitted. The sample module disconnects the connection identified by *portal_id*.

⓫ **Create a dispatch port.** Use the CREATE_PORT procedure to create a dispatch port. The sample module creates the dispatch port *dispatch_port*. The module will establish the dispatch port for the created portal. For more information about dispatch ports, see Section 8.4.2.

⓬ **Specify the message format.** If necessary, specify the message format, multiplexing information, and LLC class to be used. The sample module specifies the unused message format so that promiscuous mode can be used. When in promiscuous mode, the datalink driver delivers a copy of each message transmitted on the Ethernet.

⓭ **Establish the dispatch port for the portal.** Establish the dispatch port by calling the ELN$NI_CONNECT routine. A call to this routine must specify a portal identification number, control port, dispatch port, and message form.

The call to ELN$NI_CONNECT in the sample module establishes the dispatch port *dispatch_port* for the specified portal. The variable *portal_id* receives an integer value identifying the connection.

The *format_and_mux* argument specifies the message format to be used. This value was specified in step 12.

The *user_data_out* and *prom* arguments specify the integer value that is to be returned in each call to ELN$NI_RECEIVE and a Boolean that enables promiscuous mode.

⓮ **Wait on the dispatch port.** Wait on the dispatch port by specifying the port in a call to the WAIT_ANY procedure. The sample module waits on *dispatch_port*.

⓯ **Receive a message.** Receive a message by calling the ELN$NI_RECEIVE routine. The call to ELN$NI_RECEIVE must specify the dispatch port, a variable of type MESSAGE to receive the message, and a pointer variable to receive a pointer to the beginning of the message's user data. The call to ELN$NI_RECEIVE in the sample module includes these arguments and arguments that receive the size of the message's user data, destination address, source address, the message format, and the user data value that was defined in the call to ELN$NI_CONNECT.

⓰ **Disconnect the dispatch port from the control port.** Disconnect the dispatch port from the control port after all messages are received. The sample module disconnects the connection identified by *portal_id*.

⓱ **Get the system's CSMA/CD LAN configuration.** Get the system's CSMA/CD LAN configuration by calling the ELN$NI_GET_CONFIGURATION routine. The call to ELN$NI_GET_CONFIGURATION stores the version number of the network interface routines and the device types, device names, control port values, and data port values for all active CSMA/CD LAN controllers. You must retrieve this information before you can use the other network interface routines. The sample module returns the configuration record to *config_data*.

⓲ **Extract the control port and data port values from the controller configuration record.** Once you retrieve the controller configuration record, extract the control and data port values. You must specify a control port in calls to ELN$NI_CONNECT, ELN$NI_DISCONNECT, and ELN$NI_GET_ATTRIBUTES. You must specify a data port in calls to ELN$NI_TRANSMIT. The sample module extracts the port values for the first controller and assigns them to *control_port* and *data_port*, respectively.

Chapter 9

DECnet Network Services

The VAXELN Toolkit's DECnet Network Service routes messages sent between two DECnet network nodes, manages the list of universal names for the network, and provides a runtime interface for managing local system DECnet software. The Network Service calls the datalink driver to transmit messages; in turn, the datalink driver calls the Network Service to dispatch received messages.

When a process gets a value for a port that is not on the process's node, the kernel and the Network Service on the local node cooperate to route the message to the destination, through the Network Service on the receiver's node. Once received at the destination, the message has the same format as any message. The methods for receiving a message and replying to it are always the same.

When a process attempts to translate a universal name, the Network Service and the kernel cooperate to obtain the translation. The Network Service also provides for communication with other DECnet network nodes and implements functions for managing nodes in the network.

You configure multinode VAXELN systems with a Network Service at each node. However, the methods by which a program sends and receives messages are the same whether jobs communicate between nodes or within a single node. Data transmission between network nodes is transparent.

NOTE

The processors configured for a closely coupled symmetric multiprocessing system constitute one Ethernet node.

This chapter describes the protocols that the Network Service employs (see Section 9.1). The rest of the chapter provides information about the following DECnet Network Service services:

- Message transmission services, Section 9.2
- Name service, Section 9.3
- Network management services, Section 9.4
- Services for communicating with VMS Nodes, Section 9.5
- Remote Terminal Utility, Section 9.6

9.1 Network Service Protocols

The Network Service employs the following Phase IV DECnet protocols:

- Routing protocol, Version 2.0
- Network services protocol (NSP), Version 4.0
- Session control protocol (SCP), Version 1.0
- Data access protocol (DAP), Version 7.1

The routing protocol routes system-level datagrams between VAXELN nodes and other DECnet nodes. The protocol provides Ethernet *end-node routing*. Although end-node routing limits a VAXELN system to only one Ethernet datalink controller, such as a DEUNA or DEQNA, the routing capabilities let the VAXELN system communicate directly over the Ethernet with any DECnet node on the same Ethernet. If a full routing system is present on the Ethernet, for example, a VMS system, the VAXELN system can communicate through the routing system to any node on the entire network.

NSP and SCP support transparent application-level circuits that are connected to remote nodes. Such circuits are also known as *logical links*. They connect two remote application- or session-level ports. Therefore, a call to the VAXELN CONNECT_CIRCUIT procedure that specifies a remote destination port causes the VAXELN Network Service to create an NSP logical link with the specified destination. Likewise, the ACCEPT_CIRCUIT procedure lets the calling program accept logical links from remote destination ports. Once the circuits (logical links) are established, the NSP uses the routing protocol to deliver messages to remote systems.

VAXELN uses DAP in all communications tasks within an application, not just for message-passing. For example, console and disk I/O use DAP as their highest-level interface. All VAXELN drivers have DAP front ends to facilitate transparent multiprocessing in local area network configurations.

In addition, the VAXELN Toolkit uses direct device access (DDA) to perform local disk and terminal I/O functions that the DAP architecture does not define. DDA provides an interface for disk and serial-line read and write operations. This protocol also provides an interface for dynamically setting serial-line characteristics, setting serial lines to the spacing state, monitoring the use of out-of-band characters, and controlling modem signals.

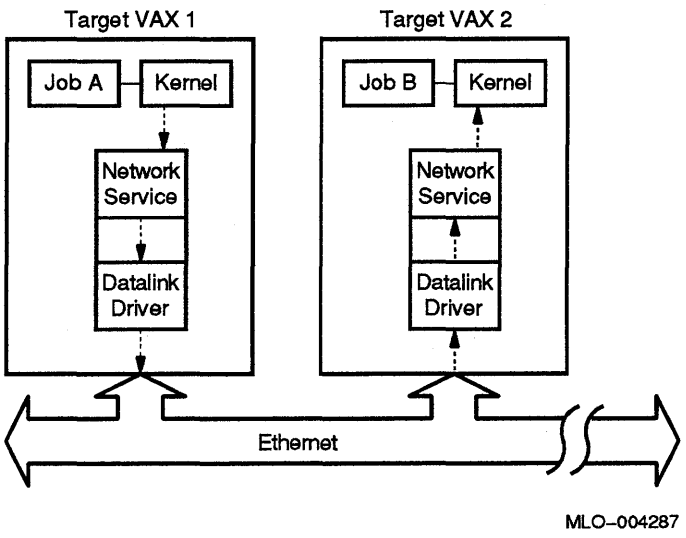
9.2 Message Transmission Services

The Network Service uses Phase IV DECnet protocols to add transparent network extensions to the message-passing kernel procedures ACCEPT_CIRCUIT, CONNECT_CIRCUIT, DISCONNECT_CIRCUIT, RECEIVE, and SEND. When an application uses these procedures to pass messages between two network nodes, the kernel and Network Service on each node cooperate to ensure message delivery. When a process sends a message, the kernel checks whether the specified port value is known to the executing node. If it is not, the kernel and Network Service route the message through the receiving node's Network Service to the destination port. The receiving process receives and replies to the message as though executing on the same node as the sending process.

Applications that include the Network Service can pass messages between nodes explicitly by using the SEND procedure and implicitly through I/O operations that use services, drivers, or hardware on a different node. The system that runs on each node in such an application must include the Network Service.

Figure 9–1 shows a 2-node VAXELN network. When Job A sends a message to Job B, the Network Service on Target VAX 1 delivers a formatted message to the datalink driver on that system.

Figure 9–1: A Two-Node VAXELN Network Using the Network Service



Part of the formatted message is the 48-bit Ethernet address of the destination node, Target VAX 2. The datalink driver on Target VAX 2 recognizes its Ethernet address in the message and forwards the message to the Network Service on its machine. The Network Service then delivers the message to a message port in the destination job, Job B.

Neither the sending nor the receiving job communicates directly with the Network Service. Instead, the kernel on each node determines whether an outgoing message is destined for a message port on the local node or a remote node.

NOTE

When jobs on the same target processor send messages to each other, the Network Service is not involved. Therefore, you can omit the Network Service from such systems.

The use of circuits is recommended, especially in network applications. However, the Network Service functions the same way when messages are sent between two unconnected message ports on different nodes.

Circuits are recommended because, whether or not you use a network, they guarantee message delivery if the physical connection is intact. (The Ethernet does not guarantee intact connections.) Circuits also guarantee that messages are delivered in the correct sequence and that messages of any length will be split, or *segmented,* into messages of the maximum size supported by the hardware. The message segments are reassembled into messages of the original size.

Generally, these guarantees are especially important in networks. If your application requires communication without circuits, you probably will have to program guarantees, such as message delivery, yourself.

An alternative to using circuits is to send data as a datagram remote port. However, the Ethernet and general DECnet networks impose a limit on message size. This limit restricts the size of datagrams that you can send to a remote port to 1500 bytes: the maximum Ethernet message size (1514 bytes) minus the size of the message header (14 bytes).

9.3 Name Service

When you build the Network Service into a VAXELN system, you can also include the Name Service. The Name Service adds network extensions to the CREATE_NAME, TRANSLATE_NAME, and DELETE procedures. These extensions let jobs access and maintain a table of universal port names (port names that are known to all nodes in a VAXELN local area network). Using universal port names, jobs can identify message destinations without having to know or maintain other jobs' PORT values.

If you include the Name Service in a VAXELN system, your application programs can do the following:

- Use the CREATE_NAME procedure to create universal names
- Use the TRANSLATE_NAME procedure to translate universal names
- Use the DELETE procedure to delete universal names

Universal port names are the key to distributed applications. By using universal port names, a VAXELN system can move a job or disk file to another node without your having to modify code. The Network Service ensures the validity of the communications path. Thus, a job running on one node can open, read, and write files that are located on another node, while the use of multiple nodes remains transparent to the user.

Each target system in a VAXELN network application retains a list of the universal names it creates and sends a copy of those names to the universal Name Service. One of these target systems acts as a *name server* and manages the universal name table.

NOTE

> The set of universal names in a VAXELN local area network is known only to the VAXELN nodes in that network. That set of names is not known to nodes running other systems, such as VMS, nor to other VAXELN nodes not directly connected to the local area network's Ethernet.

9.3.1 Name Server

A VAXELN network's name server is the VAXELN system that is responsible for managing the network's universal name table. The name server is elected from the pool of VAXELN systems that include the Name Service. If your VAXELN system includes the VAXELN Command Language Utility (ECL), you can display the name of the current name server node by issuing the SHOW NAME_SERVER command. If the name server is a remote node, this command displays the node's Ethernet address and DECnet area and node number. If the local node is the name server, the command displays a message to inform you.

9.3.2 Kernel and Name Service Interaction

The kernel and Name Service on each node in a VAXELN network communicate with each other and with the name server. Messages are sent between the Name Service on each node and name server until a valid reply is received. Kernel and Name Service interaction can be summarized as follows:

- Name Creation

When a job creates a universal name, the kernel on its node sends a message to its node's Name Service. The Name Service then sends the name and its PORT value to the name server. The name server enters the universal name in its table and sends an acknowledgment back to the Name Service. The Name Service waits for the acknowledgment from the name server (a message indicating the success or failure of the name creation) and forwards the reply back to its local kernel. The completion status is returned to the program that initiated the name creation.

- Name Deletion

  When a job deletes a universal name, the kernel informs its local Name Service of the deletion, and the Name Service informs the name server. The name server removes the name from the table, unless it has been already deleted, and replies to the Name Service. The completion status is returned to the program that initiated the name deletion.

- Name Translation

  When a job translates a universal name, the local kernel sends a message to each node's Name Service. The Name Service forwards the translation request to the name server. The name server translates the name to a PORT value, which the name server returns to the Name Service in its reply. The completion status is returned to the program that initiated the name translation.

9.3.3 Name Server Election

The Name Service preserves universal names if the current name server shuts down and at least one other system in the VAXELN network has the Name Service. Each VAXELN system in a VAXELN network that includes the Name Service is eligible to serve as the network's name server. Each of these nodes can nominate itself as the name server, but it will not necessarily be one.

The protocol for electing a name server is as follows:

- The current name server periodically broadcasts its Ethernet address to inform the other nodes that it is the current name server.
- If you build the Name Service into a system, the service retains a list of the universal names that the system creates.

- Nodes that have the Name Service listen for the name server's periodic broadcast. If a timeout interval elapses with no broadcast, another node is elected as the current name server, and each system that includes the Name Service sends its list of universal names to the new server.

Assuming that a name server is elected, the preceding protocol ensures that a system's universal port names are available to the other nodes. Thus, the failure of one node does not prevent other nodes from using universal names.

To ensure the integrity of a network's universal name table, include the Name Service in a sufficient number of systems. If necessary, you can include the service in systems that use only local names. However, keep in mind that as the size of a network and the number of systems that include the Name Service increase, the amount of overhead that results from the election process increases.

9.4 Network Management Services

The Network Service provides services for managing VAXELN DECnet nodes. The Network Service supports a subset of the Phase IV network management protocol (NMP). Thus, you can manage VAXELN DECnet nodes from a VMS host system by using the DECnet–VAX network control program (NCP). You can use the NCP to invoke functions of the following facilities:

- Network management listener (NML), Version 4.0. Monitors the network and controls DECnet systems.
- Loopback Mirror. Tests the Network Service and its ability to communicate with other nodes on the network.

The Network Service also provides the following services for managing VAXELN DECnet nodes from VAXELN target systems:

- Network Management Service. Provides a routine interface for dynamically starting and stopping DECnet software at runtime.
- Down-Line Load Service. Provides a runtime routine interface for down-line loading VAXELN system images to other VAXELN target nodes.

Section 9.4.1 explains how to use NML to manage VAXELN DECnet systems from a VMS host system. Section 9.4.2 explains how to use the Loopback Mirror to test the Network Service. Sections 9.4.3 and 9.4.4 explain how to use the Network Management Service and Down-Line Load Service.

9.4.1 Managing VAXELN DECnet Systems from a VMS Host System

You can monitor and control DECnet systems from a VMS host system by using NML. You invoke NML functions by using NCP. For information about using NCP, see the *DECnet–VAX System Manager's Guide*. This section explains NCP features that the VAXELN Toolkit supports remotely.

To use the NCP to invoke the VAXELN NML, you must first define the VAXELN system's node name and address in the VMS system's network node data base. This definition is usually established when the network is installed, but you should check that each node in your network has a unique address and name. The following VMS commands define a VAXELN system for use by the network management services:

```
$ RUN SYS$SYSTEM:NCP
NCP> DEFINE NODE FRED ADDRESS 5
NCP> SET NODE FRED ALL
```

Once you define the node, you can verify its existence in the network node data base by using the NCP SHOW NODE and SHOW CIRCUIT commands. (The circuit that you specify in the NCP command is a datalink-level circuit between nodes, not the application-level circuit referred to in VAXELN programs.)

```
NCP> SHOW NODE FRED

Node Volatile Summary as of 8-MAR-1990 12:44:41

Node      State      Active  Delay  Circuit  Next Node
                     Links

5 (FRED)  reachable                 UNA-0    5 (FRED)
```

To invoke the VAXELN NML through the NCP, use the NCP SET EXECUTOR command or the TELL prefix. The following example shows how to use the SET EXECUTOR command:

```
NCP> SET EXECUTOR NODE FRED
NCP> SHOW EXECUTOR

Node Volatile Summary as of  8-MAR-1990 10:48:00

Executor node = 5 (FRED)

State = on
Identification = VAXELN V4.1
```

The VAXELN NML supports the following NCP commands and options. Brackets identify optional items, which in most cases are mutually exclusive.

- LOOP NODE *node-id* [WITH *block-type*] [COUNT *count*] [LENGTH *length*]
- SHOW EXECUTOR [SUMMARY] [STATUS] [CHARACTERISTICS] [COUNTERS]
- SHOW KNOWN CIRCUIT [SUMMARY] [COUNTERS]
- SHOW KNOWN LINE [SUMMARY] [COUNTERS]
- SHOW NODE *node-id* [SUMMARY] [COUNTERS]
- ZERO EXECUTOR
- ZERO KNOWN CIRCUIT
- ZERO KNOWN LINE
- ZERO NODE *node-id*

9.4.2 Testing the Network Service

You can test a VAXELN system's Network Service from a VMS system or another VAXELN system by using the Loopback Mirror facility. The mirror passively loops messages sent to it, using the NCP LOOP NODE command.

The mirror is a good test of the Network Service and its ability to communicate with other nodes on the network. Therefore, you should use the LOOP NODE command whenever communication between systems is in doubt. For example, to test the communication between a remote VAXELN system and the local VMS system, use a command similar to the following:

```
NCP> LOOP NODE ENODE COUNT 100
```

To test communication between two VAXELN systems, use a command similar to the following:

```
NCP>  TELL ENODE LOOP NODE ENODE2 COUNT 100
```

9.4.3 Using the Network Management Service

The Network Service provides network management routines that an application can use to modify the state of its DECnet software. An application can start and stop DECnet software at runtime by calling the ELN$NETMAN_START_NETWORK and ELN$NETMAN_STOP_NETWORK routines.

Calls to the ELN$NETMAN_START_NETWORK routine can specify a node name, node address, and line name. A fields argument points to an aggregate that identifies which arguments you are specifying. If you are using Pascal or C, you can identify the arguments that are to be used individually, using Boolean values, or collectively, using a bit mask value. For FORTRAN applications you must use a bit mask value. If you choose the Boolean method, you set the Boolean value for each argument that you are going to specify to TRUE. When using the bit mask method, you specify the sum of the appropriate mask values. The argument fields and mask values are defined as follows:

Argument	Field Name	Mask Name	Mask Value
Node name	*node_name_field*	NETMAN$NODE_NAME_MASK	1
Node address	*node_address_field*	NETMAN$NODE_ADDRESS_MASK	2
Line name	*line_name_field*	NETMAN$LINE_NAME_MASK	4

For each field that you set, you must specify a value for a corresponding argument. For example, to specify a line name, you must set the bit for the line name in the fields argument and specify a line name for the line name argument. For the fields that you choose not to set, you can specify a null string for the corresponding argument.

The line name argument identifies the Ethernet controller over which the DECnet software is to run. If you do not specify a line name, the Network Service starts DECnet on the default Ethernet controller. If the call to ELN$NETMAN_START_NETWORK starts DECnet for the first time, the default controller is the Ethernet controller that you specified on the Network Node Characteristics Menu when you built the system. Otherwise, the default controller is the last controller on which DECnet successfully started.

If a system image is built with DECnet software disabled and that system calls ELN$NETMAN_START_NETWORK for the first time, the routine call can also specify the node address and node name that the DECnet software is to use. The values that you specify in the routine call override the node address and node name that may have been specified previously when the system was built or down-line loaded.

- If the call to ELN$NETMAN_START_NETWORK is starting DECnet for the first time and you do not specify a node address and node name, DECnet uses the address and name that were specified when the system was built or down-line loaded.
- If a call to ELN$NETMAN_START_NETWORK specifies a node address and node name and the call is not starting DECnet for the first time, the routine returns an error.

The universal Name Service is not available to systems on which DECnet software is disabled. If the Name Service was built into a system, the universal Name Service becomes available when DECnet starts on that system.

When an application calls the ELN$NETMAN_STOP_NETWORK routine to stop the DECnet software, the Network Service aborts existing network logical links, shuts down the universal name service, and stops all DECnet operations. Although DECnet operations stop, the Ethernet/IEEE 802 Datagram Service continues to run. Thus, applications can continue to use the Datagram Service's network interface routines for networking operations.

To use the Network Management Service routines, a program must include the appropriate include files. The include modules vary for each language. For Pascal programs you must include the modules $NETMAN_UTILITY and $NET_DEFINITIONS. If you are programming in C, you must include the modules $vaxelnc and $netman_utility. For FORTRAN programs, you must include the definition file ELN$:NETMAN_UTILITY.FOR.

For descriptions of Down-Line Load Service routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

The sections that follow explain how to use the ELN$NETMAN_START_NETWORK and ELN$NETMAN_STOP_NETWORK routines to do the following:

- Initialize DECnet addresses at runtime, Section 9.4.3.1
- Start and stop DECnet to temporarily reduce network overhead, Section 9.4.3.2
- Switch the Ethernet controller on which DECnet is to run, Section 9.4.3.3

9.4.3.1 Initializing DECnet Node Addresses at Runtime

You can use the ELN$NETMAN_START_NETWORK routine to initialize the local node's DECnet node address at runtime. A VAXELN application configuration that requires the loading of the same VAXELN system image onto multiple targets could benefit from such initialization. Some ROM-based applications use such configurations. You can load a VAXELN system image that has DECnet disabled onto multiple targets and then start DECnet at runtime on each system, specifying the appropriate DECnet node address. This may be more appropriate than hard-coding DECnet node addresses into multiple versions of a system image that is the same otherwise.

If the system images are to be down-line loaded or if the DECnet node addresses were supplied when the systems were built, you can specify the calls to ELN$NETMAN_START_NETWORK without the node addresses and node names. When you omit these arguments, the routine uses the address and name that were specified when the system was built or down-line loaded.

If the image is not down-line loaded and the DECnet node information is not supplied when a system is built, the application must retrieve the information. The task of retrieving the node address and node name that needs to be specified in the call to ELN$NETMAN_START_NETWORK is left to the application designer. Three approaches are as follows:

- Create a user-defined Ethernet/IEEE 802 protocol that determines a system's node address.
- Include a table in the application that maps DECnet node addresses with CPU identification numbers or Ethernet controller hardware addresses.
- Prompt for the information at the console at system start-up.

9.4.3.2 Stopping and Starting DECnet Software to Reduce Network Overhead

VAXELN networking applications can use the ELN$NETMAN_STOP_NETWORK and ELN$NETMAN_START_NETWORK routines to temporarily shut down DECnet operations to reduce network overhead for time-critical tasks. You can eliminate the following types of overhead by stopping DECnet:

- Connection requests
- End-node routing announcement messages
- Periodic network timer
- Universal Name Service operations

Although applications retain the overhead incurred from user-defined Ethernet/IEEE 802 protocols and datalink-level system identification messages (sent by ESA, EZA, and QNA device drivers), the reduction in overhead that you gain from shutting down DECnet operations may provide a significant contribution to system performance when it is most needed.

A VAXELN application that collects time-critical data and sends that data to other systems for processing is an example of an application that can benefit by temporarily shutting down DECnet operations. Such an application might stop DECnet, collect the time-critical data, and then start DECnet again to transmit the collected data to another system for processing.

To use ELN$NETMAN_START_NETWORK and ELN$NETMAN_STOP_NETWORK in such an application, you do the following:

1. Build the system image with DECnet enabled.
2. At a time-critical point in the application, call the ELN$NETMAN_STOP_NETWORK routine to stop DECnet.
3. Perform the time-critical task.
4. Call the ELN$NETMAN_START_NETWORK routine to start DECnet.
5. Communicate with other systems over the network.

The following code shows how you might program this in an application:

```
MODULE stop_n_start_decnet;

INCLUDE $NETMAN_UTILITY, $NET_DEFINITIONS;

PROGRAM stop_n_start(INPUT, OUTPUT);

VAR
  stat : INTEGER;
  specified_fields : NETMAN$INFORMATION_FIELDS;
  node_address : NET$NODE_ADDRESS;
  node_name : NET$NODE_NAME;
  line_name : VARYING_STRING(32);
    .
    .
    .
BEGIN
    .
    .
    .
  {
  {  Stop DECnet
  {}

  ELN$NETMAN_STOP_NETWORK;

  {
  {  Perform the time-critical task and then restart DECnet using the
  {  default node address, node name, and line name.
  {}

  {
  {  Set specified_fields to zero to use the current settings for the
  {  node name, node address, and line name.
  {}

  specified_fields.mask_value := 0;

  ELN$NETMAN_START_NETWORK(stat,
                           specified_fields,
                           node_address,
                           node_name,
                           line_name);
    .
    .
    .
```

9.4.3.3 Switching DECnet Software Between Ethernet Controllers

The VAXELN datalink drivers can support up to eight Ethernet controllers, only one of which can run DECnet software at a given time. If a VAXELN networking application employs a multiple Ethernet controller configuration, it can use the ELN$NETMAN_STOP_NETWORK and ELN$NETMAN_START_NETWORK routines to switch DECnet from one controller to another.

An application might use multiple controllers to maintain DECnet networking integrity if a communications path is broken. If an application can switch DECnet to another controller dynamically at runtime, another controller can take over if the communications path to the controller running DECnet fails.

The task of preparing a mechanism that determines whether a communications path has failed is left to the application designer. One approach is to program an application to do the following:

1. Check for the KER$_DISCONNECT status value. The SEND and RECEIVE procedures return this status value to notify their callers when a port was disconnected.
2. Call the DISCONNECT_CIRCUIT procedure to disconnect the partner port.
3. Call ELN$NETMAN_STOP_NETWORK to stop DECnet. The Network Service aborts existing logical links, shuts down the universal name service, and stops all network operations.
4. Close all connections to the datalink driver (established with the ELN$NI_CONNECT routine) running on the controller to which the DECnet software is to be switched.
5. Restart DECnet on the other controller with a call to ELN$NETMAN_START_NETWORK, specifying the controller's line name.
6. Reestablish connections to the datalink driver.
7. Reestablish the circuit.

The following section of code shows this approach:

```
MODULE switch_controllers;

INCLUDE $NETMAN_UTILITY, $NET_DEFINITIONS;

PROGRAM switch(INPUT, OUTPUT);
```

```
VAR
  data_port : PORT;
  stat : INTEGER;
  specified_fields : NETMAN$INFORMATION_FIELDS;
  node_address : NET$NODE_ADDRESS;
  node_name : NET$NODE_NAME;
  line_name : VARYING_STRING(32);
   .
   .
   .
{ Check for and handle a disconnected circuit. }

IF stat = KER$_DISCONNECT THEN
  BEGIN

    { Disconnect the port before trying to reestablish the connection. }

    DISCONNECT_CIRCUIT(data_port);

    { Stop DECnet. }

    ELN$NETMAN_STOP_NETWORK;

    { Start DECnet on another controller. }

    specified_fields.mask_value := NETMAN$LINE_NAME_MASK;

    node_address.area := 0;
    node_address.node := 0;
    node_name := '';
    line_name := 'XQB0';

    ELN$NETMAN_START_NETWORK(stat,
                             node_address,
                             node_name,
                             line_name);

    { Reestablish the connection. }

    CONNECT_CIRCUIT(data_port,
                    DESTINATION_NAME := dest_port_name,
                    STATUS := stat);
    SEND(msg, data_port, STATUS := stat);
  END;
   .
   .
   .
```

For more information about programming circuits, see Section 5.3.6.

Another approach is to program an application to implement a user-defined Ethernet/IEEE 802 protocol that periodically multicasts datagrams to other nodes on a LAN. Based on information that the application gathers from sending the datagrams, it can determine which nodes are available on the LAN. If nodes are not available on that LAN, the application can use ELN$NETMAN_STOP_NETWORK and ELN$NETMAN_START_NETWORK to switch DECnet operations to another controller (and LAN) that provides communications paths to all the necessary nodes.

9.4.4 Using the Down-Line Load Service

The VAXELN Down-Line Load Service handles VAXELN system load requests and provides runtime interface routines. Using the interface routines, VAXELN applications can configure, manage, and monitor a memory-resident down-line load data base. Applications can also use routines to trigger boot or down-line load VAXELN systems to remote VAXELN target nodes.

You build the Down-Line Load Service into a VAXELN system as a program image, and it runs as a system job. When the job starts executing, it creates and starts a process for each Ethernet controller on the local system. The process for a controller handles all load requests sent to and from that controller. The master process waits for data base, trigger, and load requests.

When you build the Down-Line Load Service into a VAXELN system, you can specify that it is to start when the system begins executing by selecting *Yes* for the **Run** entry on the System Builder's Program Description Menu. Alternatively, you can activate the Down-Line Load Service at runtime by using one of the following:

- A call to the CREATE_JOB procedure from an application program
- The EXECUTE/WAIT ECL command
- The RUN ECL command
- The CREATE JOB debugger command

The down-line load data base can store information about remote VAXELN target nodes and physical lines known to the local node. An application can set up the data base for a local node, using the supplied routines. Alternatively, you can create a down-line load data base script file and specify the file as a program argument for the Down-Line Load Service job. You can specify the script file when you build the service into the system image or when you activate the service at runtime.

The data base can contain an aggregate for each node and line, with the aggregate fields identifying specific node or line characteristics. The Down-Line Load Service uses the characteristics in the data base as defaults for information not supplied in load requests.

An application can keep the data base current by adding new entries, modifying existing entries, clearing duplicate information, and so on. An application can also obtain current information from the data base.

An application communicates with the Down-Line Load Service by using the following routines:

Routine	Description
ELN$DLL_CLEAR_LINE	Clears or resets down-line load data base line entries.
ELN$DLL_CLEAR_NODE	Clears down-line load data base node entries.
ELN$DLL_GET_LINE	Returns line information from the down-line load data base.
ELN$DLL_GET_NODE	Returns node information from the down-line load data base.
ELN$DLL_LOAD	Loads a VAXELN system onto another VAXELN target node.
ELN$DLL_SET_LINE	Adds information to or modifies information in down-line load data base line entries.
ELN$DLL_SET_NODE	Adds information to or modifies information in down-line load data base node entries.
ELN$DLL_TRIGGER	Trigger boots a VAXELN system.

To use these routines, a program must include the appropriate include files and establish a connection with the Down-Line Load Service's control port. Before calling some of the routines, you must also get the line name for an Ethernet device.

The include modules vary for each language. For Pascal programs you must include the modules $DLL_UTILITY and $NET_DEFINITIONS. If you are programming in C, you must include the modules $vaxelnc and $dll_utility. For FORTRAN programs, you must include the definition file ELN$:DLL_UTILITY.FOR.

Sections 9.4.4.1 to 9.4.4.6 explain how to use the runtime interface. Section 9.4.4.1 explains how to establish a circuit for Down-Line Load Service communication. Sections 9.4.4.2 to 9.4.4.6 explain how to use the Down-Line Load Service routines to do the following:

- Manage and monitor data base node entries
- Manage the monitor data base line entries
- Trigger boot VAXELN target nodes

- Down-line load VAXELN system images

For descriptions of Down-Line Load Service routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

For information about building the Down-Line Load Service into VAXELN systems, activating the service at runtime, and setting up script files, see the *VAXELN Development Utilities Guide*.

For information about down-line loading system images from VMS host nodes, see the *VAXELN Development Utilities Guide*.

9.4.4.1 Establishing Circuits for Down-Line Load Service Communication

An application program communicates with the Down-Line Load Service using a VAXELN virtual circuit. A program must establish the circuit connection by getting its job port value and connecting that port with the Down-Line Load Service control port $DLL_CONTROL. The following example shows how to establish such a connection:

```
MODULE test_dll;

INCLUDE $NI_UTILITY, $ELNMSG,
        $DLL_UTILITY, $NET_DEFINITIONS;

PROGRAM do_down_line_load(INPUT,OUTPUT);

VAR
  application_job_port, dll_port : PORT;
  stat : INTEGER;
    .
    .
    .
BEGIN
  TRANSLATE_NAME(dll_port, '$DLL_CONTROL', NAME$LOCAL, STATUS := stat);

  { Get job port. }

  JOB_PORT(application_job_port);

  { Connect the job port to the Down-Line Load Service. }

  CONNECT_CIRCUIT(application_job_port, DESTINATION_PORT := dll_port);
    .
    .
    .
END.
END;
```

Once the connection between *application_job_port* and the Down-Line Load Service port is established, the program can call the Down-Line Load Service routines, specifying *application_job_port* as a circuit argument.

9.4.4.2 Managing and Monitoring Data Base Node Entries

An application can use Down-Line Load Service routines to set up the local host's initial data base node configuration or update an existing configuration. The data base should identify remote VAXELN nodes that the local node might trigger boot or to which the local node might down-line load a VAXELN system image.

The data base node entries include the node's name and DECnet address. Optionally, an entry can specify the hardware address, a line name, a VAXELN system image file, a secondary load file, and a tertiary load file. The line name identifies the controller on the local node to be used for trigger boot and down-line load operations. All other node entry information pertains to a remote target node. Table 9–1 summarizes these characteristics.

Table 9–1: Down-Line Load Data Base Node Characteristics

Characteristic	Description
Node name	The name of the target node.
Node address	The DECnet node address of the target node.
Hardware address	The hardware address of the Ethernet controller on the target node.
Line name	The name of the line device on the local node to be used for down-line load operations for the target node.
Image file name	The name of the VAXELN system image file to be down-line loaded to the target node.
Secondary loader file name	The name of the secondary loader file. The secondary loader is a small image that a system's primary loader may request. In turn, the secondary loader may request a tertiary loader.

Table 9–1 (Cont.): Down-Line Load Data Base Node Characteristics

Characteristic	Description
Tertiary loader file name	The name of the tertiary loader file. The tertiary loader is a larger image that a system's secondary loader may request. In turn, the tertiary loader may request the VAXELN system image file.

When you trigger boot or down-line load an image, the Down-Line Load Service uses the data that you specify in the routine call and gets any missing information from the data base. Thus, if a call to ELN$DLL_LOAD specifies only a node name, the Down-Line Load Service looks for a line name, image file, secondary loader file, and tertiary loader file in the data base. If necessary, the service also tries to derive a physical address from the DECnet node address stored in the data base.

An application can modify and monitor the data base by calling the ELNDLL_SET_NODE, ELNDLL_CLEAR_NODE, and ELN$DLL_GET_NODE routines. ELN$DLL_SET_NODE adds or modifies data base entries. ELN$DLL_CLEAR_NODE clears the information stored in the entries. ELN$DLL_GET_NODE returns target node information from the data base.

Calls to these routines must specify the port connected in a circuit to the $DLL_CONTROL port. The routines use the connected circuit to communicate with the Down-Line Load Service.

Calls to ELN$DLL_CLEAR_NODE and ELN$DLL_GET_NODE must also identify the node for which information is to be cleared or returned. The node identifier must be a string representing the node's name (for example, BNODE) or DECnet node address (for example, 12.2). You can specify a wildcard string ('*') to indicate that the routine is to clear or return information from all node entries in the data base.

To set or clear a node's characteristics, an application must also specify a fields argument. This argument indicates that the corresponding value in the data base aggregate is to be set or cleared. You can set or clear the node name, node address, hardware address, line name, image file, secondary loader file, and tertiary loader file.

You can specify the aggregate fields to be set or cleared individually, using Boolean values, or collectively, using a bit mask value. If you choose the Boolean method, you set the Boolean value for each characteristic that you want to set or clear to TRUE. If you choose the bit mask method, you specify the sum of the appropriate mask values in the mask value field. The characteristic fields and mask values are defined as follows:

Characteristic	Field Name	Mask Name	Mask Value
Node name	*node_name_field*	DLL$NODE_NAME_MASK	1
Node address	*node_address_field*	DLL$NODE_ADDRESS_MASK	2
Hardware address	*hardware_address_field*	DLL$HARDWARE_ADDRESS_MASK	8
Line name	*line_name_field*	DLL$LINE_NAME_MASK	16
System image file name	*image_file_field*	DLL$IMAGE_FILE_MASK	64
Secondary load file name	*sec_loader_file_field*	DLL$SEC_LOADER_FILE_MASK	128
Tertiary load file name	*tert_loader_file_field*	DLL$TERT_LOADER_FILE_MASK	256

NOTE

The mask values 4 and 32 are for use with the ELN$DLL_LOAD and ELN$DLL_TRIGGER routines; the values do not represent node characteristics that you can set or clear. The value 4 indicates that you specified a node identifier (name or DECnet address). The value 32 indicates whether you specified a physical address. For more information, see Sections 9.4.4.5 and 9.4.4.6.

You can set or clear fields without changing other fields in the aggregate. However, for each field that you set, you must specify a value for a corresponding argument. For example, to set a node name, you must set the bit for the node name field in the fields argument and specify the name of a target node for the node name argument. For the fields that you choose not to set, you can specify a null string for the corresponding argument.

If a node entry that you want to modify contains a node name but no address and you specify only the node address in the call to ELN$DLL_SET_NODE, the routine creates a new entry, using the node address as the node's identification. When this happens, you should combine the information in the two entries for that node to avoid conflicts between the original and updated information. To combine the entries, clear the data in one entry and set the information from the cleared entry in the second entry.

To return node information from the data base, an application must specify the name of a user-defined show node routine. ELN$DLL_GET_NODE invokes the show node routine if it finds the specified node in the data base. If you specify the string '*' for the node identifier, ELN$DLL_GET_NODE calls your routine once for each node in the data base and returns all user-specified node information.

Example 9–1 adds a node entry to the down-line load data base, clears the hardware address for that entry, and then returns the information stored in the entry.

Example 9–1: Managing and Monitoring Down-Line Load Data Base Node Entries

```
MODULE manage_dll_node_entries;

INCLUDE $NI_UTILITY, $ELNMSG,
        $DLL_UTILITY, $NET_DEFINITIONS;

PROGRAM manage_nodes(INPUT,OUTPUT);

VAR
  app_job_port, dll_port : PORT;
  set_status, clear_status, get_status : INTEGER;
  new_fields, clear_fields : DLL$NODE_INFORMATION_FIELDS;
  node_name : NET$NODE_NAME;
  node_address : NET$NODE_ADDRESS;
  node_identifier : NET$NODE_NAME_ADDRESS;
  hardware_address : NET$ETHERNET_ADDRESS;
  line_name : VARYING_STRING(32);
  image_file : VARYING_STRING(255);
  sec_loader_file : VARYING_STRING(255);
  tert_loader_file : VARYING_STRING(255);
  dll_msg : MESSAGE;

BEGIN

  {  Get the values for the Down-Line Load Service control port and the
  {  program's job port and establish a connection.
  {}

  JOB_PORT(app_job_port);                                        ❶

  CONNECT_CIRCUIT(app_job_port,
                DESTINATION_NAME := '$DLL_CONTROL');
```

Example 9–1 Cont'd on next page

Example 9–1 (Cont.): Managing and Monitoring Down-Line Load Data Base Node Entries

```
{  Add an entry to the down-line load data base.
{}

new_fields.mask_value := 0;                                          ❷

new_fields.node_name_field := TRUE;
new_fields.node_address_field := TRUE;
new_fields.hardware_address_field := TRUE;
new_fields.line_name_field := TRUE;
new_fields.image_file_field := TRUE;

node_name := 'BNODE';
node_address.area := 12;
node_address.node := 1;
line_name := 'XQA0';
hardware_address.address := ''(%X00, %X00, %X11, %X00, %X22, %X33);
image_file := 'BNODE::DUA0:[DLL]IMAGEFILE.SYS';
sec_loader_file := '';
tert_loader_file := '';

ELN$DLL_SET_NODE(set_status,                                         ❸
                 app_job_port,
                 new_fields,
                 node_name,
                 node_address,
                 hardware_address,
                 line_name,
                 image_file,
                 sec_loader_file,
                 tert_loader_file);

WRITELN('Status of data base set operation: ', set_status);

{  Clear the new entry's hardware address.
{}

clear_fields.mask_value := 0;                                        ❹

clear_fields.hardware_address_field := TRUE;

ELN$DLL_CLEAR_NODE(clear_status,                                     ❺
                   app_job_port,
                   clear_fields,
                   'BNODE');

WRITELN('Status of data base clear operation: ', clear_status);

{  Display the information in the data base entry.
{}
```

Example 9–1 Cont'd on next page

Example 9–1 (Cont.): Managing and Monitoring Down-Line Load Data Base Node Entries

```
  ELN$DLL_GET_NODE(get_status,                                    ❻
                   app_job_port,
                   '12.1',
                   show_node);

  WRITELN('Status of data base get operation: ', get_status);

DISCONNECT_CIRCUIT(app_job_port);                                 ❼
END;

PROCEDURE show_node of type ELN$DLL_SHOW_NODE_ROUTINE;            ❽

BEGIN

  {  Return information from the fields that are set.
  {}

  WITH node_info_ptr::^DLL$NODE_INFORMATION^ DO
  BEGIN
    WRITELN('***********************************')
    WRITELN('        Version = ', version);
    IF node_flag_set.node_name_field THEN
      WRITELN('        Node name =', node_name);
    IF node_flag_set.node_address_field THEN
      BEGIN
        WRITELN('        Node address area =', node_address.area);
        WRITELN('        Node address node =', node_address.node);
      END;
    IF node_flag_set.hardware_address_field THEN
      WRITELN('        Hardware address =', hardware_address.address);
    IF node_flag_set.line_name_field THEN
      WRITELN('        Line name =', line_name);
    IF node_flag_set.image_file_field THEN
      WRITELN('        System image file =', image_file);
    IF node_flag_set.sec_loader_file_field THEN
      WRITELN('        Secondary loader file =', sec_loader_file);
    IF node_flag_set.tert_loader_file_field THEN
      WRITELN('        Tertiary loader file =', tert_loader_file);
    WRITELN('***********************************')
  END;

END;
END.
```

➊ **Connect to the Down-Line Load Service control port.** Get the value of the program's job port and connect that port in a circuit to the Down-Line Load Service control port. The sample module connects the job port *app_job_port* in a circuit to the local control port $DLL_CONTROL. For information about connecting to the Down-Line Load Service control port, see Section 9.4.4.1.

➋ **Set up the data base aggregate for a new node entry.** Set up the data base aggregate for a new node entry by clearing the new fields mask value, setting the appropriate fields in the aggregate, and assigning values to the corresponding arguments. The sample module uses the Boolean method to set the node name, node address, line name, hardware address, and image file fields to TRUE. Alternatively, the module could have set the fields by assigning a mask value of 91 (the sum of the mask values 1, 2, 8, 16, and 64) to the mask value field. The module then assigns appropriate values to the *node_name*, *node_address*, *line_name*, *hardware_address*, and *image_file* arguments. Null strings are assigned to the *sec_loader_file* and *tert_loader_file* arguments because those fields are not being set. Although, you do not have to specify values for fields that are not being set; the arguments are ignored.

➌ **Add a new node entry to the data base.** Add a new node entry to the down-line load data base by calling ELN$DLL_SET_NODE. A call to this routine must specify the port connected in a circuit to the $DLL_CONTROL port and a value for the new fields argument. The call must also specify values for arguments representing the node's name, DECnet address, and hardware address; the name of the line on the local node to be used for load operations; and the name of the system image file, secondary load file, and tertiary load file to be loaded. The *new_fields* argument identifies the node characteristics the application will be setting. The routine call in the sample application sets the node name, node address, line name, hardware address, and image file name.

➍ **Set up the data base aggregate for a clear operation.** Set up the data base aggregate for a clear operation by clearing the clear fields mask value and setting the appropriate fields in the aggregate. The sample module uses the Boolean method to set the hardware address field to TRUE. Alternatively, the module could have assigned a mask value of 8 to the mask value field.

❺ **Clear node information from the data base.** Clear node information from the down-line load data base by calling ELN$DLL_CLEAR_NODE. You must specify the port connected in a circuit to the $DLL_CONTROL port, a value for the *clear_fields* argument, and a node identifier. The routine call in the sample module specifies that the hardware address for node BNODE be cleared. Alternatively, the node identifier could have been specified as '12.1'.

❻ **Return node information from the data base.** To check the data that is in the down-line load data base, issue a call to ELN$DLL_GET_NODE. The call must specify the port connected in a circuit to the $DLL_CONTROL port, a node identifier, and the name of a user-defined show node routine. The call to ELN$DLL_GET_NODE in the sample module returns information about the node whose DECnet node number is 12.1. Alternatively, the node identifier could have been specified as 'BNODE'.

❼ **Disconnect the application job port from the Down-Line Load Service control port.** Disconnect the application job port from the Down-Line Load Service control port when the circuit is no longer needed. The sample module disconnects the connection identified by *app_job_port*.

❽ **Define the show node routine to be invoked by ELN$DLL_GET_NODE.** The user-defined routine *show_node* displays information that is in the aggregate fields that are set. For BNODE, the routine will display the node name, node address, line name, and system image file name.

9.4.4.3 Managing and Monitoring Data Base Line Entries

An application can use Down-Line Load Service routines to update the local host's data base line configuration. The data base should identify the physical Ethernet lines on the local node that can be used for trigger boot and down-line load operations.

The data base line entries include the line name. Optionally, an entry can specify the line's state, retry count, and service timer. Table 9–2 summarizes these characteristics.

Table 9–2: Down-Line Load Data Base Line Characteristics

Characteristic	Description
Down-line load enabled	Flag that specifies whether the line is enabled for trigger boot and down-line load operations.
Retry count	The number of times the Down-Line Load Service is to send an unacknowledged load request to the target node. The service abandons the load attempt when the number of tries exceeds the count value.
Service timer	The amount of time the Down-Line Load Service is to wait for a response message from a target node during a down-line load operation before resending the load message.

When you trigger boot or down-line load an image, the Down-Line Load Service uses the line characteristics that are in the data base for the specified line. Thus, if a call to ELN$DLL_LOAD specifies the line name XQA0, the Down-Line Load Service looks for the characteristics for XQA0 in the data base.

An application can modify and monitor the data base by calling the ELNDLL_SET_LINE, ELNDLL_CLEAR_LINE, and ELN$DLL_GET_LINE routines. ELN$DLL_SET_LINE modifies data base entries. ELN$DLL_CLEAR_LINE resets the information stored in the entries. ELN$DLL_GET_LINE returns local node line information from the data base.

Calls to ELNDLL_SET_LINE, ELNDLL_CLEAR_LINE, and ELN$DLL_GET_LINE must specify the port connected in a circuit to the $DLL_CONTROL port and a line name. The routines use the connected circuit to communicate with the Down-Line Load Service. The line name is a string that identifies a physical line on the local node (for example, XQA0). You can get the names of available lines from controller configuration aggregates returned by ELN$NI_GET_CONFIGURATION (see Section 8.2) or by using the ECL command SHOW DEVICES.

When calling ELN$DLL_CLEAR_LINE or ELN$DLL_GET_LINE you can specify a wildcard string ('*') for the line name argument. The wildcard string indicates that the routine is to reset or return information from all line entries in the data base.

To set or clear a line's characteristics, an application must also specify a fields argument. This argument identifies the fields in the data base for the line that you intend to set or clear. You can set or clear the down-line load enabled flag, the retry count, and service timer. When you clear the retry count or service timer, ELN$DLL_CLEAR_LINE resets the values to 5 and 4000 milliseconds, respectively.

You can specify the aggregate fields to be set or cleared individually, using Boolean values, or collectively, using a bit mask value. If you choose the Boolean method, you set the Boolean value for each characteristic that you want to set or clear to TRUE. If you choose the bit mask method, you specify the sum of the appropriate mask values in the mask value field. The characteristics fields and mask values are defined as follows:

Characteristic	Field Name	Mask Name	Mask Value
Down-line load enabled	*dll_enabled_field*	DLL$DLL_ENABLED_MASK	1
Retry count	*retry_count_field*	DLL$RETRY_COUNT_MASK	2
Service timer	*service_timer_field*	DLL$SERVICE_TIMER_MASK	4

You can set or clear fields without changing other fields in the aggregate. However, for each field that you set, you must specify a value for a corresponding argument. For example, to set the retry count, you must set the bit for the retry count field in the fields argument and specify a count value for the retry count argument. For the fields that you choose not to set, you can specify a null string for the corresponding argument.

NOTE

The fields argument is ignored if you specify a wildcard string for the line name in a call to ELN$DLL_CLEAR_LINE.

To return node information from the data base, an application must specify the name of a user-defined show line routine. ELN$DLL_GET_LINE invokes the show line routine if it finds the specified line in the data base. If you specify a wildcard string for the line name, ELN$DLL_GET_LINE calls your routine once for each line in the data base and returns the name of the line, state of the down-line load enabled flag, retry count value, service timer value, and the hardware address of the local node's Ethernet controller with which the line is associated.

Example 9–2 resets the retry count and service timer for a line entry that is in the down-line load data base and returns the information stored in that entry.

Example 9–2: Managing and Monitoring Down-Line Load Data Base Line Entries

```
MODULE manage_dll_line_entries;

INCLUDE $NI_UTILITY, $ELNMSG,
        $DLL_UTILITY, $NET_DEFINITIONS;

PROGRAM manage_lines(INPUT,OUTPUT);

VAR
  app_job_port, dll_port : PORT;
  status : INTEGER;
  set_status, clear_status, get_status : INTEGER;
  new_fields, clear_fields : DLL$LINE_INFORMATION_FIELDS;
  dll_enabled : BOOLEAN;
  retry_count : INTEGER;
  service_timer : LARGE_INTEGER;
  line_name : VARYING_STRING(32);
  hardware_address : NET$ETHERNET_ADDRESS;
  config : ELN$NI_CONFIGURATION := ZERO;
  line_count : INTEGER;

BEGIN

  {  Get the values for the Down-Line Load Service control port and
  {  the program's job port and establish a connection.
  {}

  JOB_PORT(app_job_port);                                          ❶

  CONNECT_CIRCUIT(app_job_port,
                DESTINATION_NAME := '$DLL_CONTROL');

  {  Set the retry count and service timer for line XQA0.
  {}

  ELN$NI_GET_CONFIGURATION(status,                                 ❷
                         line_count,
                         config);

  IF ODD(status) THEN
    line_name := config.clist[1].name;

  new_fields.mask_value := 0;                                      ❸
```

Example 9–2 Cont'd on next page

Example 9–2 (Cont.): Managing and Monitoring Down-Line Load Data Base Line Entries

```
  new_fields.retry_count_field := TRUE;
  new_fields.service_timer_field := TRUE;

  retry_count := 10;
  service_timer := TIME_VALUE('0 00:00:20');

  ELN$DLL_SET_LINE(set_status,                        ❹
                   app_job_port,
                   new_fields,
                   line_name,
                   dll_enabled,
                   retry_count,
                   service_timer);

  WRITELN('Status of data base set operation: ', set_status);

  {  Reset the entry's retry count and service timer.
  {}

  clear_fields.mask_value := 0;                       ❺

  clear_fields.retry_count_field := TRUE;
  clear_fields.service_timer_field := TRUE;

  ELN$DLL_CLEAR_LINE(clear_status,                    ❻
                     app_job_port,
                     clear_fields,
                     line_name);

  WRITELN('Status of data base clear operation: ', clear_status);

  {  Display the information in the data base entry.
  {}

  ELN$DLL_GET_LINE(get_status,                        ❼
                   app_job_port,
                   line_name,
                   show_line);

  WRITELN('Status of data base get operation: ', get_status);

  DISCONNECT_CIRCUIT(app_job_port);                   ❽
END;
```

Example 9–2 Cont'd on next page

Example 9–2 (Cont.): Managing and Monitoring Down-Line Load Data Base Line Entries

```
PROCEDURE show_line of type ELN$DLL_SHOW_LINE_ROUTINE;  ❾

BEGIN

  WITH line_info_ptr^ DO
  BEGIN
    WRITELN('***********************************');
    WRITELN('        Version = ', version);
    WRITELN('        Line name =', line_name);
    WRITELN('        Down-line load enabled =', dll_enabled);
    WRITELN('        Retry count =', retry_count);
    WRITELN('        Service timer =', TIME_STRING(service_timer));
    WRITELN('        Hardware address =', hardware_address.address);
    WRITELN('***********************************')
  END;

END;
END.
```

❶ **Connect to the Down-Line Load Service control port.** Get the value of the program's job port and connect that port in a circuit to the Down-Line Load Service control port. The sample module connects the job port *app_job_port* in a circuit to the local control port $DLL_CONTROL. For information about connecting to the Down-Line Load Service control port, see Section 9.4.4.1.

❷ **Get the name of a line.** Use a call to the ELN$NI_GET_CONFIGURATION routine to get the controller aggregates that describe the Ethernet controller on the local node. From the aggregates, you can extract the names of the controllers and specify the names for the line name argument in calls to ELNDLL_SET_LINE, ELNDLL_CLEAR_LINE, and ELN$DLL_GET_LINE. The sample module extracts the name of the first controller and assigns that name to *line_name*.

❸ **Set up the data base aggregate for the line entry.** Set up the data base aggregate for the line entry by clearing the new fields mask value, setting the appropriate fields in the aggregate, and assigning values to the corresponding arguments. The sample module uses the Boolean method to set the retry count and service timer fields to TRUE. Alternatively, the module could have set the fields by assigning a mask value of 6 (the sum of the mask values 2 and 4) to the mask value field. The module then assigns the values 10 and 5000 to the *retry_count* and *service_timer* arguments, respectively.

❹ **Modify a line entry in the data base.** Modify a line entry in the down-line load data base by calling ELN$DLL_SET_LINE. A call to this routine must specify the port connected in a circuit to the $DLL_CONTROL port and a value for the *new_fields* argument. The call must also specify values for arguments representing the line's down-line load enabled flag, retry count, and service timer. The *new_fields* argument identifies the line characteristics the application will set. The routine call in the sample application sets the retry count and service timer.

❺ **Set up the data base aggregate for a clear operation.** Set up the data base aggregate for a clear operation by clearing the clear fields mask value and setting the appropriate fields in the aggregate. The sample module uses the Boolean method to set the retry count and service timer fields to TRUE. Alternatively, the module could have assigned a mask value of 6 to the mask value field.

❻ **Clear line information from the data base.** Clear line information from the down-line load data base by calling ELN$DLL_CLEAR_LINE. You must specify the port connected in a circuit to the $DLL_CONTROL port, a value for the *clear_fields* argument, and a line name. The routine call in the sample module specifies that the retry counter and service timer for *line_name* be cleared.

❼ **Return line information from the data base.** To check the data that is in the down-line load data base, issue a call to ELN$DLL_GET_LINE. The call must specify the port connected in a circuit to the $DLL_CONTROL port, a line name, and the name of a user-defined *show_line* routine. The call to ELN$DLL_GET_LINE in the sample module returns information about *line_name*.

❽ **Disconnect the application job port from the Down-Line Load Service control port.** Disconnect the application job port from the Down-Line Load Service control port when the circuit is no longer needed. The sample module disconnects the connection identified by *app_job_port*.

❾ **Define the show line routine to be invoked by ELN$DLL_GET_LINE.** The user-defined routine *show_line* displays information that is in the aggregate fields. The *show-line* routine in the sample module displays the version number of the Down Line Load Service and the name, down-line load flag value, retry count, service timer, and hardware address for *line_name*.

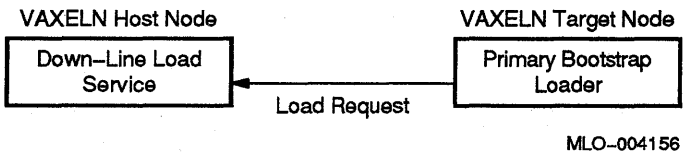
9.4.4.4 Managing Target-Initiated Down-Line Load Requests

A VAXELN system can set up a down-line load data base for servicing load requests by calling routines to set, clear, and retrieve information from data base entries. The VAXELN Down-Line Load Service services both application- and target-initiated down-line load requests. Application-initiated requests are trigger boot and down-line load requests that result from calls to the ELN$DLL_TRIGGER and ELN$DLL_LOAD routines. Trigger booting and down-line loading are discussed in Sections 9.4.4.5 and 9.4.4.6, respectively.

A target-initiated load request is an unsolicited request that a target node transmits in response to a console BOOT command. A target node might direct the load request to a specific node or to a multicast address.

Figure 9–2 shows the load request message flow that might result from a console BOOT command.

Figure 9–2: Target-Initiated Down-Line Load Request



The Down-Line Load Service listens for load requests on lines for which it is enabled. After receiving such a request, the service creates a process that reads and processes the request.

The Down-Line Load Service extracts the data in the load request and determines whether a request is directed to a multicast address or the local node. If the request is for a multicast address, the service volunteers to perform the load if a node entry in the down-line load data base matches the target node's hardware address.

The service volunteers by sending a message to the requesting node. If the target node does not respond, the service drops the initial load request. Otherwise, it continues to service the request.

When servicing the remainder of a multicast load request or a request directed to the local node, the Down-Line Load Service checks for other information that may have been supplied in the load request. If the request includes additional information, it is used to service the request. If the load request does not supply all the necessary information, the Down-Line Load Service searches the down-line load data base for information it needs. When the service has all the required information, it performs the load operation.

The typical load sequence for a target-initiated load request begins with the primary bootstrap loader running on the target node. Typically, this program executes directly from the target node's bootstrap ROM, or it is in the microcode of the load device. After the primary loader is triggered, the target node sends a message requesting a program load to an eligible host node. (The host node may be a specific node defined by the target node, or any node on the Ethernet.) Usually, the primary loader requests a secondary loader program, which may request a tertiary loader. The tertiary loader may then request a VAXELN system image file.

9.4.4.5 Trigger Booting a VAXELN Target Node

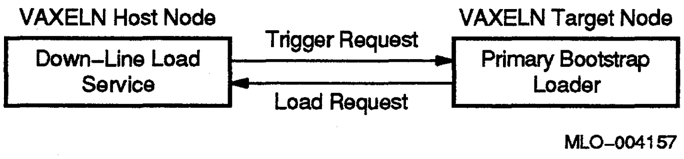
A VAXELN application can trigger boot another VAXELN target node by calling the ELN$DLL_TRIGGER routine. This routine triggers a remote node's bootstrap ROM, causing the target node to issue a request for a load operation. Depending on how the target node's primary bootstrap loader is programmed, a system image is loaded from a specific host, any host on the LAN, or a local disk.

To trigger boot a target node, an application sends a trigger request message to that node. The trigger request must specify the name of the line over which the trigger and load requests are to be sent and the target node's Ethernet address.

Depending on how a target node's primary bootstrap loader is programmed, a target node might respond to a trigger request by sending a load request message back to the host VAXELN system. The Down-Line Load Service running on the host system extracts the load information it needs from the message; if information is missing, the service searches the down-line load data base for the missing information. The host system then uses the information to down-line load (copy) a system image over the Ethernet to the target node.

Figure 9–3 shows the message flow that might result from a trigger boot operation.

Figure 9–3: Trigger Boot Request



An application initiates a trigger boot with a call to ELN$DLL_TRIGGER. A call to this routine must specify the port connected in a circuit to the $DLL_CONTROL port. The routine uses the connected circuit to communicate with the Down-Line Load Service.

A call to ELN$DLL_TRIGGER must also identify the name of the line to be used for the operation and the Ethernet address of the target node. You can identify the line name and Ethernet address by specifying a node identifier or a line name and physical address. If you specify a node identifier, the Down-Line Load Service searches for the required information in the down-line load data base.

The Down-Line Load Service uses the node identifier to get information needed to derive a physical Ethernet address. The service derives a physical Ethernet address from the DECnet node address that you specify as a node identifier or that is stored in the data base. In addition to trying to derive the physical address, the service searches the node's data base entry for a hardware address. If both addresses are available, both are used to trigger the target node. The target node responds to the address that is appropriate for the target node's state (running or not running). If only one of the addresses is available, the service tries to trigger boot the target node using that address.

The information that you specify in the routine call overrides the data in the data base. Thus, if you specify a node identifier, line name, and physical address, the Down-Line Load Service uses the line name and physical address specified in the routine call; corresponding information in the data base is ignored. Specifically, if you specify a physical address in the call to ELN$DLL_TRIGGER, the Down-Line Load Service uses only that address to trigger the target node. An available hardware address will not be used.

A fields argument points to an aggregate that identifies which arguments you are specifying. You can identify the arguments that are to be used individually, using Boolean values, or collectively, using a bit mask value. If you choose the Boolean method, you set the Boolean value for each argument that you will be specifying to TRUE. If you choose the bit mask method, you specify the sum of the appropriate mask values in the mask value field. The argument fields and mask values are defined as follows:

Status	**Field Name**	**Mask Name**	**Mask Value**
Node identifier	*node_name_address_field*	DLL$NODE_NAME_ADDRESS_MASK	4
Ethernet address	*physical_address_field*	DLL$PHYSICAL_ADDRESS_MASK	16
Line name	*line_name_field*	DLL$LINE_NAME_MASK	32

For each field that you set, you must specify a value for a corresponding argument. For example, to specify a line name and Ethernet address, you must set the bits for the line name and Ethernet address fields in the fields argument and specify a line name and Ethernet address for the line name and physical address arguments. For the fields that you choose not to set, you can specify a null string for the corresponding argument.

Example 9–3 shows how a VAXELN application might trigger boot another VAXELN target node.

Example 9–3: Trigger Booting a VAXELN Target Node

```
MODULE manage_dll_line_entries;

INCLUDE $NI_UTILITY, $ELNMSG,
        $DLL_UTILITY, $NET_DEFINITIONS;

PROGRAM manage_lines(INPUT,OUTPUT);

VAR
  app_job_port, dll_port : PORT;
  status : INTEGER;
  specified_fields : DLL$NODE_INFORMATION_FIELDS;
  node_identifier : NET$NODE_NAME_ADDRESS;
   .
   .
   .
BEGIN

  {  Get the values for the Down-Line Load Service control port and
  {  the program's job port and establish a connection.
  {}

  JOB_PORT(app_job_port);

  CONNECT_CIRCUIT(app_job_port,
                  DESTINATION_NAME := '$DLL_CONTROL');

  specified_fields.mask_value := 0;

  specified_fields.node_name_address_field := TRUE;

  node_identifier := 'BNODE';
  line_name := '';
  physical_address := '';

  ELN$DLL_TRIGGER(status,
                  app_job_port,
                  specified_fields,
                  node_identifer,
                  line_name,
                  physical_address);

  WRITELN('Status of trigger operation: ', status);
  .
  .
  .
  DISCONNECT_CIRCUIT(app_job_port);
END;
END.
```

The call to ELN$DLL_TRIGGER in Example 9–3 boots the target node identified by the node name BNODE. The Down-Line Load Service uses the specified node identifier, BNODE, to find the appropriate line name and Ethernet address for the trigger operation in the down-line load data base. If the service does not find a line name and Ethernet address for BNODE, the trigger request is dropped.

NOTE

If a VAXELN system is to be trigger booted with the Down-Line Load Service, you must build that system with trigger booting enabled. To enable trigger booting, you must select *Yes* for the **Node triggerable** entry on the Network Node Characteristics Menu. trigger booting on) The target node does not need to be running the Down-Line Load Service. You can specify *Disk*, *ROM*, or *Downline* for **Boot method.**

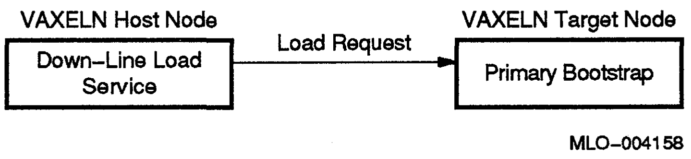
9.4.4.6 Down-Line Loading VAXELN Systems

A VAXELN application can down-line load a VAXELN system image to another VAXELN target node by calling the ELN$DLL_LOAD routine. This routine tries to down-line load a system image to a remote target node. You supply the load information as arguments in the routine call. The Down-Line Load Service gets any unspecified information from the down-line load data base. If the service cannot find all the information it needs to service the request, an error status value is returned.

A down-line load request must specify the name of the line over which the load requests are to be sent, the target node's Ethernet address, and the name of the file to be loaded.

Figure 9–4 shows the message flow that might result from a down-line load operation.

Figure 9–4: Down-Line Load Request



The Down-Line Load Service can accept multiple load requests. However, the service can perform only one load operation for a particular target at a given time. A load request to a particular target node overrides preceding requests to that node. However, if the service receives a request to trigger a node during a load operation of that node, the service honors the trigger request, overriding the load operation.

An application initiates a down-line load operation with a call to ELN$DLL_LOAD. A call to this routine must specify the port connected in a circuit to the $DLL_CONTROL port. The routine uses the connected circuit to communicate with the Down-Line Load Service.

A call to ELN$DLL_LOAD must also identify the name of the line to be used for the operation, the Ethernet address of the target node, and the name of the system image file to be loaded. Some target nodes also require secondary and tertiary loader files. You can identify this data by specifying it in the routine call or by specifying a node identifier. If you specify a node identifier, the Down-Line Load Service searches for the required information in the down-line load data base.

The Down-Line Load Service uses the node identifier to get information needed to derive a physical Ethernet address. The service derives a physical Ethernet address from the DECnet node address that you specify as a node identifier or that is stored in the data base. In addition to trying to derive the physical address, the service searches the node's data base entry for a hardware address. If both addresses are available, both are used to load the system image. The target node responds to the address that is appropriate for the target node's state (running or not running). If only one of the addresses is available, the service tries to down-line load the system image using that address.

The information that you specify in the routine call overrides the data in the data base. Thus, if you specify a node identifier, line name, and physical address, the Down-Line Load Service uses the line name and physical address specified in the routine call; corresponding information in the data base is ignored. Specifically, if you specify a physical address in the call to ELN$DLL_LOAD, the Down-Line Load Service uses only that address to down-line load the system image. An available hardware address will not be used.

A fields argument points to an aggregate that identifies which arguments you are specifying. You can identify the arguments that are to be used individually, using Boolean values, or collectively, using a bit mask value. If you choose the Boolean method, you set the Boolean value for each argument to be specified to TRUE. If you choose the bit mask method, you specify the sum of the appropriate mask values in the mask value field. The argument fields and mask values are defined as follows:

Argument	Field Name	Mask Name	Mask Value
Node identifier	*node_name_address_field*	DLL$NODE_NAME_ADDRESS_MASK	4
Ethernet address	*physical_address_field*	DLL$PHYSICAL_ADDRESS_MASK	16
Line name	*line_name_field*	DLL$LINE_NAME_MASK	32
Load file	*image_file_field*	DLL$IMAGE_FILE_MASK	64
Secondary loader file	*sec_loader_file_field*	DLL$SEC_LOADER_FILE_MASK	128
Tertiary loader file	*tert_loader_file_field*	DLL$TERT_LOADER_FILE_MASK	256

For each field that you set, you must specify a value for a corresponding argument. For example, to specify a line name and Ethernet address, you must set the bits for the line name and Ethernet address fields in the fields argument and specify a line name and Ethernet address for the line name and physical address arguments. For the fields that you choose not to set, you can specify a null string for the corresponding argument.

Example 9–4 shows how a VAXELN application might down-line load a VAXELN system image to another VAXELN target node.

Example 9–4: Down-Line Loading a VAXELN System Image

```
MODULE manage_dll_line_entries;

INCLUDE $NI_UTILITY, $ELNMSG,
        $DLL_UTILITY, $NET_DEFINITIONS;

PROGRAM manage_lines(INPUT,OUTPUT);

VAR
  app_job_port, dll_port : PORT;
  status : INTEGER;
  specified_fields : DLL$NODE_INFORMATION_FIELDS;
  node_identifier : NET$NODE_NAME_ADDRESS;
  image_file : VARYING_STRING(255);
  sec_loader_file : VARYING_STRING(255);
  tert_loader_file : VARYING_STRING(255);
   .
   .
   .
BEGIN

  {  Get the values for the Down-Line Load Service control port and
  {  the program's job port and establish a connection.
  {}

  JOB_PORT(app_job_port);

  CONNECT_CIRCUIT(app_job_port,
                  DESTINATION_NAME := '$DLL_CONTROL');

  specified_fields.mask_value := 0;

  specified_fields.node_name_address_field := TRUE;
  specified_fields.image_file_field := TRUE;

  node_identifier := 'BNODE';
  line_name := '';
  physical_address := '';
  image_file := 'HNODE::DUA0:[DLL]IMAGEFILE.SYS';
  sec_loader_file := '';
  tert_loader_file := '';

  ELN$DLL_LOAD(status,
               app_job_port,
               specified_fields,
               node_identifer,
               line_name,
               physical_address,
               image_file,
               sec_loader_file,
               tert_loader_file);
```

Example 9–4 Cont'd on next page

Example 9–4 (Cont.): Down-Line Loading a VAXELN System Image

```
  WRITELN('Status of down-line load operation: ', status);
  .
  .
  .
  DISCONNECT_CIRCUIT(app_job_port);
END;
END.
```

The call to ELN$DLL_LOAD in Example 9–4 down-line loads the image IMAGEFILE.SYS to the target node identified by the node name BNODE. The Down-Line Load Service uses the specified node identifier, BNODE, to find the appropriate line name and Ethernet address for the trigger operation in the down-line load data base. If the service does not find a line name and Ethernet address for BNODE, the load request is dropped.

NOTE

> A VAXELN system that the Down-Line Load Service is to load must be built with down-line loading enabled. To enable down-line loading, select *Downline* for the **Boot method** entry on the System Characteristics Menu. down-line loading on)

9.5 Services for Communicating with VMS Nodes

VAXELN and VMS systems can communicate transparently or non-transparently. Using *transparent communication*, VAXELN and VMS programs can exchange information with standard I/O statements over the network as if the programs were running on the same system. Transparent communication offers the basic mechanism for establishing a connection, exchanging messages, and breaking the connection.

Nontransparent communication allows VMS programs to use network-specific features to handle the message exchange. The features available are a superset of those available in the transparent case but require more knowledge of DECnet operation and more sophisticated programming. (For example, a VAXELN program can use the

ACCEPT_DATA and CONNECT_DATA parameters of the kernel's circuit procedures to exchange up to 16 bytes of data with a remote VMS program as part of the NSP connection requests and acceptances.)

When using nontransparent communication, VMS systems use mailboxes to handle multiple connection requests and the $QIO function codes IO$_ACPCONTROL and IO$_ACCESS to establish names and accept connections from multiple VAXELN processes. (For more information and examples, see the *DECnet–VAX User's Guide.*)

A complete explanation of VMS network I/O is beyond the scope of this manual. The following sections provide information specific to the VAXELN Toolkit about the following:

- Specifying nodes, Section 9.5.1
- Requesting connections from VAXELN systems, Section 9.5.2
- Accepting connections on VMS systems, Section 9.5.3
- Requesting connections from VMS systems, Section 9.5.4
- Accepting connections on VAXELN systems, Section 9.5.5
- Using object numbers in connection requests, Section 9.5.6

For more information, see the *DECnet–VAX User's Guide.*

9.5.1 Specifying Nodes

When nodes running VAXELN systems and nodes running other operating systems are connected to the same network, you need to be able to identify them to each other. This allows VAXELN systems to operate on files stored on all systems, to establish circuits to the other system, and so on.

You do not need to use node specifications to identify VAXELN nodes to each other. For example, a VAXELN program on one node can use a file stored on another node without giving a node identifier in the file specification. The network locations of VAXELN jobs are transparent to one another. Node specifications are needed only for communication between VAXELN nodes and nodes running other operating systems.

In DECnet networks, nodes are identified by node name and by node number; either is a unique identification of a node. A node name has a maximum of six characters, and a node number is an integer. The VMS command SHOW NETWORK displays both the name and number of the nodes known to the DECnet–VAX software.

VAXELN systems use node numbers to access a remote node. Other operating systems can use either node numbers or node names (assuming node names are supported on the particular system) to access a VAXELN node.

You can use the network control program (NCP) on a VMS system to assign node names and numbers to VAXELN nodes as usual. (For a brief introduction to NCP, see Section 9.4.1.)

Once you use NCP to establish the VAXELN node in the DECnet–VAX data base, you can use the SHOW NETWORK command to display the node any time the node is running and its system image includes the Network Service.

9.5.1.1 Using Node Names and Node Numbers in VMS

You can use a VAXELN node name or number from another operating system to display directories, perform other directory- or file-related operations on File Service volumes, and perform network management operations. Suppose you want to display a directory that resides on a VAXELN file-server node named ENODE. If you enter the SHOW NETWORK command on the VMS system, the system might display something like the following:

```
$  SHOW NETWORK

  Node       Links    Cost    Hops    Next Hop to Node
  10 RVAXAA  0        0       0       (Local)
  12 ENODE   1        3       1       UNA-0
```

Here, RVAXAA is the node from which you want to access a file on ENODE. You can use either of the following VMS commands to display the directory [ANALOG.DATA] on disk volume DISK$A on node ENODE (12):

```
$  DIR ENODE::DISK$A:[ANALOG.DATA]

$  DIR 12::DISK$A:[ANALOG.DATA]
```

If you have used this feature on VMS before, you will be familiar with this syntax for network file and directory operations. The name or number preceding the double colon (::) is the node containing the specified directory or file.

9.5.1.2 Using Node Numbers in VAXELN

When working from a VAXELN node, you specify a remote node (such as a VMS node) by number. Suppose you want to open a file on the VMS node RVAXAA. The SHOW NETWORK command on the VMS system might display something like the following:

```
$  SHOW NETWORK

  Node        Links    Cost     Hops     Next Hop to Node
  10 RVAXAA   0        0        0        (Local)
   3 ELN1     1        3        1        UNA-0
```

Here, ELN1 is the node from which you want to access a file on RVAXAA. You can open the file as usual, with the OPEN procedure appropriate to the language, and the node number of RVAXAA in the file specification. For example, you specify a call to the Pascal OPEN procedure as follows:

```
OPEN(pasvar, FILE_NAME := '10::SYS$LIBRARY:DIGITAL.DAT');
```

You specify the C equivalent as follows:

```
#include stdio
FILE *file_ptr;
file_ptr=fopen("10::SYS$LIBRARY:DIGITAL.DAT","r");
```

The FORTRAN equivalent would look like the following:

```
OPEN(FILE = '10::SYS$LIBRARY:DIGITAL.DAT', TYPE = 'NEW', UNIT = 100);
```

9.5.2 Requesting Connections from VAXELN Systems

You can use the CONNECT_CIRCUIT procedure to request a connection with a VMS program on the same DECnet network by specifying the *destination_name* argument in the following format:

'nodenumber::objectname'

The *nodenumber* is a network node number (as described in Section 9.5.1), and *objectname* is the name of the object on the VMS system that will handle the connection.

Set up a command procedure that runs the desired VMS program image, name the procedure *objectname*.COM, and place it in the default DECnet directory on the VMS system. The command procedure executes when the DECnet–VAX software gets a request for a connection to the specified object. The VMS image then handles the connection.

9.5.3 Accepting Connections on VMS Systems

A VMS program image has two ways of waiting for and accepting a connection from a VAXELN system:

- You can use an operation that is comparable to using the VAXELN ACCEPT_CIRCUIT procedure.
- You can specify the name SYS$NET in a high-level language OPEN procedure (or equivalent).

(In a VAX MACRO program, you can use the $ASSIGN system service.)

You can break the connection by calling the DISCONNECT_CIRCUIT procedure from your VAXELN program or by performing a close operation in the VMS program.

9.5.4 Requesting Connections from VMS Systems

A VMS program can request a connection with a VAXELN program by using a high-level language OPEN procedure or the $ASSIGN system service with a name of the form:

nodename::"TASK=*portname*"

The *nodename* is the name of the VAXELN network node, and the *portname* is the character string name of the port created by the VAXELN program.

9.5.5 Accepting Connections on VAXELN Systems

The VAXELN program does nothing special to accept a connection from a remote VMS program. The VAXELN program needs only to create a PORT object and a NAME object for the port and then call the ACCEPT_CIRCUIT procedure to await the connection request.

9.5.6 Using DECnet Object Numbers in Connection Requests

A VAXELN program can connect and accept connections using requests that specify DECnet object numbers instead of names. This feature is useful only for compatibility with existing DECnet applications.

To connect to a port or object by number, specify a string with this format for the DESTINATION_NAME parameter of CONNECT_CIRCUIT:

'*nodenumber::objectnumber*'

To accept a connection for an object by number, create a port name of the form:

'NET$OBJECT_*objectnumber*'

Here, *objectnumber* is the object number in ASCII. Once the name is created, connections can be accepted as usual.

9.6 Remote Terminal Utility

The Network Service provides a Remote Terminal Utility that lets you connect to a remote computer system from a terminal on another computer system by using a SET HOST command. For example, you can connect to a VAXELN system from a VMS system terminal by using the DCL SET HOST command, or you can connect to a VMS system from a VAXELN system terminal by using the ECL SET HOST command. Once connected to a remote system, you can log in, use operating system commands (such as DCL and ECL commands), receive messages, and interact with programs that run on that system.

To use the Remote Terminal Utility, you must build it into your VAXELN system with the outbound, inbound, or outbound/inbound capability. The outbound capability lets you connect to computer systems from your VAXELN system. The inbound capability lets you connect to your VAXELN system from other systems.

For more information about the Remote Terminal Utility and the ECL SET HOST command, see the *VAXELN Development Utilities Guide*.

Chapter 10

Internet Services

You can use the VAXELN Internet Services for VAXELN applications that need to communicate between two computer hosts that reside on the same or on different networks. The *hosts* are the sources and destinations of transferred data. The Internet Services provide the protocols necessary for VAXELN applications to transfer data over an Internet.

An *Internet* is a set of connected networks. Higher-level software hides the underlying Internet architecture and makes a collection of networks appear as a single large network. The hosts on a network are physically connected and networks on the Internet are physically connected. Applications can communicate across intermediate networks even though the networks are not connected to the source or destination host. The hosts that connect and transfer messages between networks are called *gateways*.

NOTE

Although VAXELN systems can use gateways for Internet communication, they cannot function as gateways.

The VAXELN Internet Services provide the following:

- Connectionless or end-to-end connection-oriented packet delivery service
- Packet delivery service that is independent of the communications medium over which data is transmitted
- Communications environment that supports a variety of computer platforms
- Communications protocol standards

This chapter explains Internet Service concepts in Section 10.1, how to configure a VAXELN system that uses the Internet Services in Section 10.2, and how an application can use runtime routines to do the following:

- Control the Internet Services, Section 10.3
- Convert the byte order of Internet and host physical addresses, Section 10.4
- Manipulate Internet addresses, Section 10.5
- Communicate over the Internet, Section 10.6
- Retrieve and set socket characteristics, Section 10.7

NOTE

The VAXELN Internet Services currently support a C language runtime interface only.

10.1 Internet Service Concepts

Before using the VAXELN Internet Services, you should understand the following Internet Service concepts:

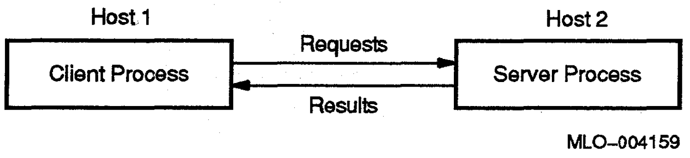
- Client-server model
- Internet architecture
- Internet addresses
- Ports as Internet communication endpoints
- Sockets
- Routing
- Fragmentation

Sections 10.1.1 to 10.1.7 explain these concepts.

10.1.1 Client-Server Model

The hosts in a network environment communicate through processes. A process that offers a service over the network to another process is known as a *server*. Servers accept requests from other processes known as clients. A *client* sends requests and waits for the results from the server. Figure 10–1 represents a client-server model.

Figure 10–1: Client-Server Model



A process name on a host cannot be used as the destination for message communication for the following reasons:

- Heterogeneous operating systems define processes differently.
- Not all processes that send data have enough information to identify a process on another host.
- Process IDs can change.

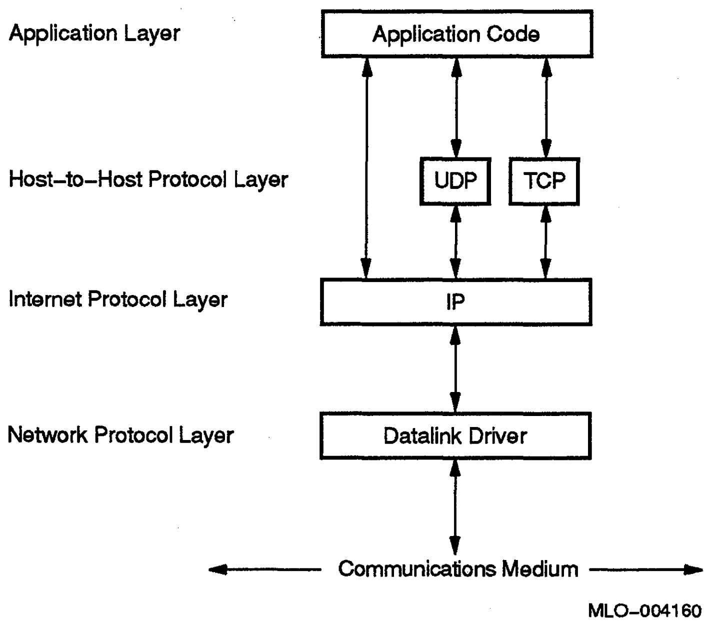
Therefore, the hosts on the Internet identify communication endpoints using ports (see Section 10.1.4). Internet protocols that comprise the Internet architecture allow communication between the client and server endpoints.

10.1.2 Internet Architecture

The Internet architecture consists of four layers of protocol that allow two-way interprocess data flow between hosts, gateways, and networks. The architecture includes an application layer, host-to-host protocol layer, Internet Protocol (IP) layer, and network protocol layer. Figure 10–2 illustrates the Internet layers.

The host-to-host layer supports two protocols: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

Figure 10–2: Internet Layers



Processes on a host transmit data by passing it to the lower protocol layers. A process at the application layer passes the data to the host-to-host protocol layer. The host-to-host protocol layer then packages the data according to protocol functions. For example, TCP adds a header that ensures reliable communication. Then the protocol sends the packaged data to the IP layer. The IP also adds a header and sends the data to the local datalink driver.

Sections 10.1.2.1, 10.1.2.2, and 10.1.2.3 describe IP, UDP, and TCP, respectively.

10.1.2.1 Internet Protocol

The *Internet Protocol (IP)* is a protocol that is used for data communication in a packet-switched computer network. IP implements mechanisms for connecting networks and gateways into a system that can deliver network packets from source to destination.

IP saves applications from addressing network specifics by doing the following:

- Routing packets to destinations through networks
- Keeping track of routes for hosts and networks
- Accounting for incompatibilities

The protocol packages message data and a header in blocks called *datagrams*. The header provides fixed-length source and destination Internet addresses, a protocol number that identifies the host-to-host protocol being used, and a checksum value. The datagrams are encapsulated in the network packets that are delivered between the source and destination hosts. IP can fragment and reassemble datagrams if necessary to accommodate requirements of smaller packet networks.

IP is specifically limited to delivering datagrams, without provisions for reliability, flow control, sequencing, or other services found in host-to-host protocols.

In addition to handling datagram fragmentation, IP implements address mapping, and transmits control and error messages by using the following protocols:

- **Address Resolution Protocol (ARP).** Dynamically maps Internet addresses to physical Ethernet addresses and stores the address pairs in an ARP cache. Using this protocol, an application can determine a target host's physical (built-in) Ethernet address. Section 10.1.3 provides more information about Internet addresses. For more information about managing the ARP cache, see Section 10.3.1.
- **Internet Control Message Protocol (ICMP).** Transmits error and control messages to a destination host's IP when an IP datagram delivery fails. ICMP provides routing information and notifies hosts when a datagram cannot reach its destination or when a datagram's keep-alive time reaches zero.

- **Reverse Address Resolution Protocol (RARP).** Determines a diskless host's Internet address at start-up so that the host can operate in an Internet network. A host can broadcast a message that specifies its physical Ethernet address to all hosts in a local area network (LAN). A host running an RARP server searches its address data base and responds by returning the appropriate Internet address. See Section 10.1.3 for more information about Internet addresses.

10.1.2.2 User Datagram Protocol

The *User Datagram Protocol (UDP)* is layered on IP and provides host-to-host datagram communication for applications that do not require streamed communication. UDP adds multiplexing to IP, letting multiple processes use the protocol to send and receive data independently. The protocol achieves mutliplexing by using ports to identify the processes executing on a host.

UDP lets application programs send messages to programs running on other hosts in a network using minimal protocol. The protocol is transaction oriented, and it does not guarantee delivery or duplicate protection.

UDP accepts a message from an application, places the message in a datagram, and tries to deliver the datagram. The datagrams may not arrive at the destination or may arrive out of order. Because UDP does not provide a reliable service, applications generally add reliability by including error and sequence control.

Table 10–1 summarizes UDP characteristics:

Table 10–1: UDP Characteristics

Protocol Characteristic	UDP Specifics
Initial setup	Not required
Transmission path	Datagram
Error handling	Done by application
Remote address	Remote address may be specified on each transmission
End-to-end flow control	Not provided
Data sequencing	Passed in order of arrival

Table 10–1 (Cont.): UDP Characteristics

Protocol Characteristic	UDP Specifics
Checksum computation	Provided

The VAXELN Toolkit provides a Boot Protocol (BOOTP) that is based on UDP. Like RARP, BOOTP determines a diskless host's Internet address at start-up so that the host can operate in an Internet network. A host can broadcast a message that specifies its physical address to all hosts in a LAN. A host running a BOOTP server searches its address data base and responds by returning the appropriate Internet address.

10.1.2.3 Transmission Control Protocol

The *Transmission Control Protocol (TCP)* is layered on IP and provides host-to-host, connection-oriented communication in a network environment. TCP adds multiplexing, checksum computations, connectivity, and reliability to IP. TCP provides for reliable interprocess communication between pairs of processes executing on host computers attached to distinct but interconnected networks. Although TCP is layered on IP, TCP does not require reliability of the underlying IP and datalink driver.

TCP uses virtual circuits for data transmission. The virtual circuits provide automatic sequencing, error control, and flow control.

Applications that use TCP must establish a virtual circuit connection before transferring data. Once an application establishes the connection, the application can use data transfer calls to send data to a destination without specifying a destination address. When the connection is no longer needed, the application must explicitly shut it down.

TCP provides the following functions:

- Transfers a continuous stream of bytes in each direction between a source and destination. TCP breaks up a message into bytes, packages the bytes into segments for transmission through the Internet, and reassembles the message at the destination. TCP ensures that all data is transferred.

- Recovers lost, duplicated, or out-of-order data by assigning a sequence number to each *octet* (eight bits) transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. The receiver's TCP uses the sequence numbers to reorder segments that are received out of order and to eliminate duplicates. TCP handles damaged data by adding a checksum to each transmitted segment, checking it at the receiver end, and discarding damaged segments.
- Handles flow control. TCP controls data flow by returning a *sliding window* (message buffer size) with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window identifies the number of octets that the sender can transmit before receiving an ACK.
- Provides for multiplexing. Using ports, multiple processes running on a host can use TCP simultaneously.
- Establishes connections using unique device interfaces that specify connection-related information, such as status information, sequence numbers, and window sizes.

  TCP establishes a connection by using a handshaking mechanism with initial sequence numbers to avoid connection initialization errors. An application should terminate a connection and free resources when the connection is no longer needed.

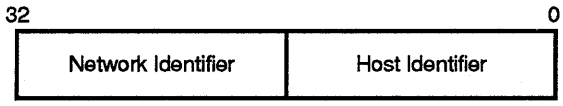
Table 10–2 summarizes TCP characteristics:

Table 10–2: TCP Characteristics

Protocol Characteristic	TCP Specifics
Initial setup	Required
Transmission path	Virtual circuit
Error handling	Transparent to application
Remote address	Remote address is required at setup
End-to-end flow control	Provided
Data sequencing	Passed in order sent
Checksum computation	Provided

10.1.3 Internet Addresses

For a source host to communicate with a destination host, it must know the *Internet address* of the destination host. An Internet address is a 32-bit (four octets) address that identifies a network and a host.



MLO–004161

The network identifier must be the same for all hosts connected to the same network, and no two networks can have the same network identifier if they are connected in any way.

No two hosts on the same network can have the same host identifier.

The notation used to represent a 32-bit Internet address consists of four decimal integer fields separated by periods. The value in each field can range from 0 to 255. A sample Internet address might be represented as 5.0.2.10.

An Internet address can fall into one of three network classes and can identify subnetworks (see Section 10.1.6). A network mask informs a system which bits of an Internet address to interpret as the network, subnetwork, and host addresses. A broadcast mask interprets an Internet address as a broadcast address. Sections 10.1.3.1, 10.1.3.2, and 10.1.3.3 provide more information about network classes, network masks, and broadcast masks, respectively.

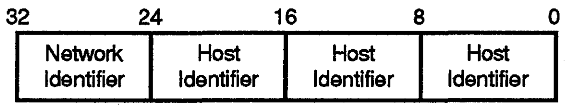
10.1.3.1 Network Classes

In addition to providing a network identifier, the network part of an Internet address identifies a network class. The Internet supports three network classes: Class A, Class B, and Class C. The network configuration determines a network's class type.

The four Internet address fields are used in different ways to specify the network class, network number, and host number. The high-order bits in an Internet address designate the network class of the address. The first high-order bits for each class are defined as follows:

Class	High-Order Bits
A	0
B	10
C	110

For a Class A network, the first field specifies the network number and class and the remaining three fields specify a subnet number, if subnetworks are being used (see Section 10.1.6), and the host number. The following figure shows such an Internet address:



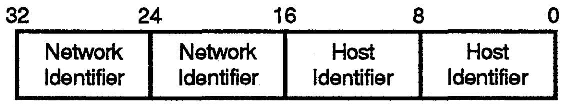
MLO–004162

The value in the first field can range from 1 to 126, inclusive. By convention, 127 is reserved as the loopback address. Loopback is used for testing the connectivity to a specific host in the network.

NOTE

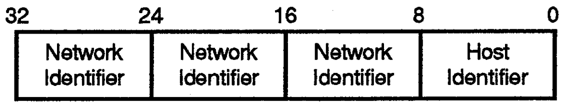
Currently, the VAXELN Internet Services do not use 127 as the loopback address.

For a Class B network, the first two fields specify the network number and class, and the remaining two fields specify a subnet number, if subnetworks are being used, and the host number. The value in the first field can range from 128 to 191 and the value in the second field can range from 1 to 254. The following figure shows the Internet address format for a Class B network:



MLO–004163

For a Class C network, the first three fields specify the network number and class, and the remaining field specifies the host number, as shown in the following figure:



MLO–004164

The value in the first field can range from 192 to 223, the value in the second field can range from 0 to 255, and the value in the third field can range from 1 to 254. Subnet routing is not generally used with a Class C network because there are only eight bits in the host field. Table 10–3 lists the ranges of the network numbers for the three network classes.

Table 10–3: Network Class Number Ranges

Class	Number
A	1.—126.
B	128.1—191.254
C	192.0.1—223.255.254

To determine which network class to use, you must consider the number of network hosts and the number of Internet networks.

The Class A network is best suited for sites with a few networks but numerous hosts, because it has 24 bits in the host part of its Internet address. The 24 bits allow for the most host-number combinations. In this case, the network part of the Internet address consists of seven usable bits, leaving 126 usable network-number combinations (0 and 127 are reserved).

The Class B network is best suited for sites where the number of networks is about equal to the number of hosts, because the 32 bits of the Internet address are evenly divided between the network and the host part of the address. The network part uses 16 bits and the host part uses 16 bits.

The Class C network is best suited for sites with numerous networks but few hosts, because the network part of the Internet address has 21 usable bits. The 21 bits allow up to 2,097,152 network-number combinations, while the eight bits of the host part of the Internet address can have only up to 254 host-number combinations.

If you are planning to set up a LAN, you should obtain a registered Internet address. This way, if you choose to connect your network with another network, you will not have to change your Internet addresses. You can obtain a registered Internet address by calling the Network Information Center at 1–800–235–3155 from inside the United States.

10.1.3.2 Network Mask

A *network mask* is a 32-bit number that informs the system which bits of the Internet address to interpret as the network, subnetwork, and host addresses. A one-to-one correspondence exists between the 32 bits in the network mask and the 32 bits in the Internet address.

For each bit in the network mask that is set (binary 1), the corresponding bit position in the Internet address is interpreted as part of the network and subnetwork address.

The decimal number 255 is 11111111 in binary notation. The value 255 means that an entire 8-bit field is set because each bit position is a 1. Generally, an 8-bit field is either set (255) or cleared (0). Values other than 255 and 0 can be used, but by using 255 or 0 you make it easier to differentiate between the network, subnetwork, and host fields.

If the network mask bit position is part of the host field and is set, the corresponding bit in the Internet address is interpreted as part of the subnetwork address. If the network mask bit position is part of the host field and is cleared, the corresponding bit in the Internet address is interpreted as part of the host address.

Each bit in the first (leftmost) field of the network mask must be set (decimal value of 255, binary value of 11111111), because the first field of the Internet address must always be interpreted as the network address regardless of whether subnetworks exist. If a bit in the first field of the network mask is cleared, part of the network field of the Internet address is interpreted as part of the host address. This may cause errors.

The second and third fields are usually 255 or 0, depending on how the Internet address is to be interpreted. The fourth field is usually 0, indicating that it represents the host address.

A Class A network mask is usually 255.255.0.0 or 255.255.255.0. When the network mask is 255.255.0.0, the first octet is the network address, the second octet is the subnet address, and the third and fourth octets are the host address. If the network mask is 255.255.255.0, the first octet is the network address, the second and third octets are the subnet address, and the fourth octet is the host address.

If a Class B network uses 255.255.255.0 for a network mask, the first and second octets are the network address, the third octet is the subnet address, and the fourth octet is the host address.

Normally, Class C networks do not have subnetworks, because only eight bits are allocated for the host part of the Internet address. Eight bits may not be enough to divide between a subnetwork address and a host address.

The default network masks for each class are as follows:

Class	Default Network Mask
A	255.0.0.0
B	255.255.0.0
C	255.255.255.0

10.1.3.3 Broadcast Mask

A *broadcast mask* interprets an Internet address as a broadcast address. Using the broadcast address, a process can send messages to all hosts on the network that have the same Internet broadcast address at the same time.

The format of the broadcast address consists of the network number followed by all ones (1).

NOTE

Some operating systems, such as UNIX BSD 4.2 and ULTRIX–32 prior to Version 1.2, require that the Internet broadcast address be the network number followed by all zeros (0). Currently, the VAXELN Internet Services support only the default format.

The network number includes the subnet, if there is one.

If you know the Internet address and the network mask for a particular host, you can calculate the broadcast mask by using the following formula:

$$(NOT\ networkmask)\ OR\ (internetaddress)$$

For example, if a host has an Internet address of 128.50.100.100 and the network mask 255.255.0.0 (the default), the host's broadcast mask is 128.50.255.255. The *NOT* of the host's network mask is 0.0.255.255. You then substitute the first two fields of the Internet address for the two zeros to get the broadcast mask.

Table 10–4 lists examples of broadcast addresses.

Table 10–4: Broadcast Addresses

Host Internet Address	Host Number	Network Class	Network Number	Network Mask (Subnet Mask)	Broadcast Address
3.0.0.10	10	A	3.	255.0.0.0	3.255.255.255
11.1.0.12	12	A	11.1.	255.255.0.0	11.1.255.255
129.39.0.15	15	B	129.39.	255.255.0.0	129.39.255.255
128.45.2.8	8	B	128.45.2.	255.255.255.0	128.45.2.255
192.0.1.8	8	C	192.0.1.	255.255.255.0	192.0.1.255
192.0.1.223	223	C	192.0.1.	255.255.255.0	192.0.1.255

10.1.4 Ports as Internet Communication Endpoints

While Internet addresses identify source and destination hosts, ports represent the endpoints of a communications link between two processes. Like the messages sent to a VAXELN port, Internet messages sent to a port are queued until another process extracts them. Processes that are waiting for messages are blocked until a message arrives.

To send data to a port on another host, a process uses a destination host's Internet address and a port number. The Internet address identifies a network and a host. The port number identifies a particular destination on the host. A process also specifies a source port when it sends a message. The process that receives the message can use the source port to return a reply.

Integers identify the communications ports. The source and destination ports are not necessarily identified with the same port number. TCP/IP and UDP/IP use port numbers that range from 1 to 65535.

Port numbers ranging from 1 to 1023 identify privileged ports. Privilege means something different for each operating system. In general, when a host receives a message from a privileged port, you can assume that the destination host has done some level of checking against the application using the port.

The port numbers ranging from 1 to 255 are reserved to provide a service contact point to known callers. Digital honors these assigned ports as implemented in the Department of Defense (DoD) and Defense Advanced Research Projects Agency (DARPA) Internet communities.

Before an application can use UDP/IP or TCP/IP for communication, a process must be bound to a port. An application binds a process to a port by specifying an Internet address and port number in a call to the **bind** function (see Section 10.6.2).

NOTE

To bind a process to a privileged port, the calling program must be authorized with a system group UIC (that is, a UIC less than or equal to %X0008FFFF or [10, 177777]).

10.1.5 Sockets

A *socket* is a communication endpoint abstraction that allows two peers to communicate. The peers can be entities such as two programs, two processes within a program, or a program itself.

Sockets have the following properties:

- Communication domain
- Protocol type
- Protocols

A *communication domain* is the collective common properties of processes communicating through sockets. One such property would be the naming scheme of the sockets. The VAXELN Internet Services support the Internet (AF_INET) domain.

Protocol types are the communication properties that are visible to the user. Normally, processes communicate only between sockets of the same protocol type. Three protocol types are available as defined in Table 10–5.

Table 10–5: Socket Protocol Types

Protocol Type	Description
Stream	Provides bidirectional, reliable, sequenced, and unduplicated data flow without record boundaries. The receiving processes are guaranteed to receive messages, in order, without duplication.
Datagram	Provides bidirectional data flow that does not guarantee that messages will be received in sequence, without duplication, or at all. The record boundaries of the data are preserved.
Raw	Provides access to underlying communications protocols that support sockets. Raw sockets are not intended for the general user; they are mainly available for developing new communications protocols.

The stream, datagram, and raw protocol types map to the protocols TCP, UDP, and IP, respectively. These protocols are described in Section 10.1.2.

Before a process can use a socket, the process must bind a name (Internet destination) to the socket. A *socket name* consists of an Internet address (network and host) and port number (process on the host). Once a socket has a name, an application can use the socket for connection or connectionless communication. Sections 10.1.5.1 and 10.1.5.2 provide more information about these two modes of communication.

10.1.5.1 Connection Socket Communication

After a process binds a name to a socket, the process can use that socket to establish a connection and communicate with another process over the Internet. One process can function as a client and the other as a server. Once a socket is created, the server listens to its socket for service requests. The client requests services from the server by initiating a connection request.

If the client process's socket is unnamed at the time of a connection request, the Internet software assigns a name to the socket. If the connection is successful, the socket is associated with the server and data can be transmitted. If the connection is unsuccessful, an error is returned (the name that the system binds to the socket remains).

A connection may be unsuccessful for one of the following reasons:

- A lack of resources on the source or destination host
- An application problem such as:
  - Conventions not being followed
  - The incorrect port number being specified
  - A privileged port number being required

After binding the socket, the server can receive a client's connection request if the following conditions exist:

- Server is listening for the connection request
- Maximum number of outstanding connections that can be queued to the server's port has not been reached

If a client requests a connection when the queue is full, the messages that comprise the request are ignored and the client retries the request. Once a connection is established, data can be exchanged between the two sockets.

For communication to take place between the source and destination hosts, the socket at each endpoint must be bound to a name. The application program on the source host must provide its Internet address and the destination socket name. The source port number is optional. If the application program omits the port number, the Internet software on the source host selects a port number.

10.1.5.2 Connectionless Socket Communication

Sockets can also support connectionless communication typical of datagram facilities found in packet-switched networks. While processes are still likely to have a client-server relationship, applications do not need to establish connections. Instead, each message includes a destination address.

You create datagram sockets the same way that you create sockets for connection-oriented communication. However, you must bind a name to each datagram socket to identify the message sender and receiver.

For source and destination hosts to communicate, applications must specify the source and destination socket names. The application program on the source host must provide its Internet address and the destination socket name. The source port number is optional. If the application program omits the port number, the Internet software on the source host selects a port number.

10.1.6 Routing

A *route* is the path over the Internet that information takes to get from one host to another. A route can be a path to either a host or a network. IP uses routes to hosts for sending packets to a remote host and uses routes to networks for sending packets to any host in a remote network.

A *subnetwork* is a set of hosts within a network that are organized into a logical group. A network can be made up of several subnetworks. A host on another network can access a host on a subnetwork if a gateway connects the networks. The data from the host on the other network is routed through the gateway to the network and onto the appropriate subnetwork, where the destination host ultimately receives the data. A subnet mask identifies the bits in an Internet address to be used for the network and subnet addressing.

The VAXELN Internet Services support static routing. This method of routing employs a table that pairs destination Internet addresses with Internet addresses that specify routes. Each table entry also contains flags that specify the following:

- Whether IP should use only the network portion of an Internet address or an entire Internet address when searching for a matching destination Internet address in the routing table
- Whether the route for a destination Internet address is to a host on the local network or to a gateway on the local network
- Whether the route is locked to prevent ICMP from updating the route with redirect messages

The destination Internet address identifies a host or network. The Internet address that specifies a route can identify a host or gateway on the local network. The Internet address for a gateway is an intermediate destination for datagrams being sent to the network identified in the table entry. Figure 10–3 shows the routing table format.

Figure 10–3: Routing Table

Flags	Destination	Route
Network or Local? Locked?	Destination Internet Address	Gateway or Local Internet Address
⋮	⋮	⋮

MLO–004165

IP uses the routing table to determine the appropriate path for a datagram. The protocol extracts the destination Internet address from the datagram and searches for a matching destination Internet address in the routing table, extracting the network portion of the address as necessary.

Figure 10–4 provides an overview of the routing algorithm.

Figure 10–4: Routing Algorithm

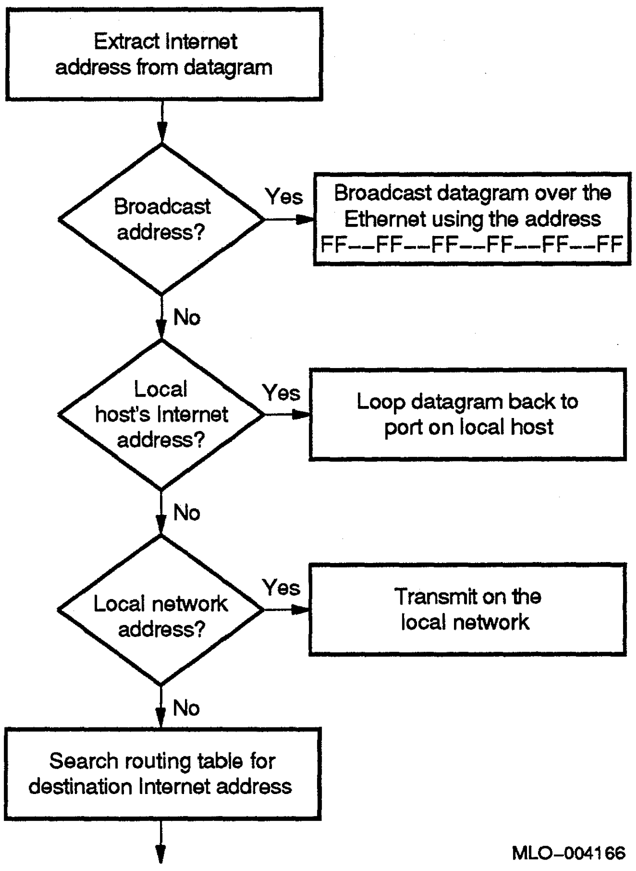
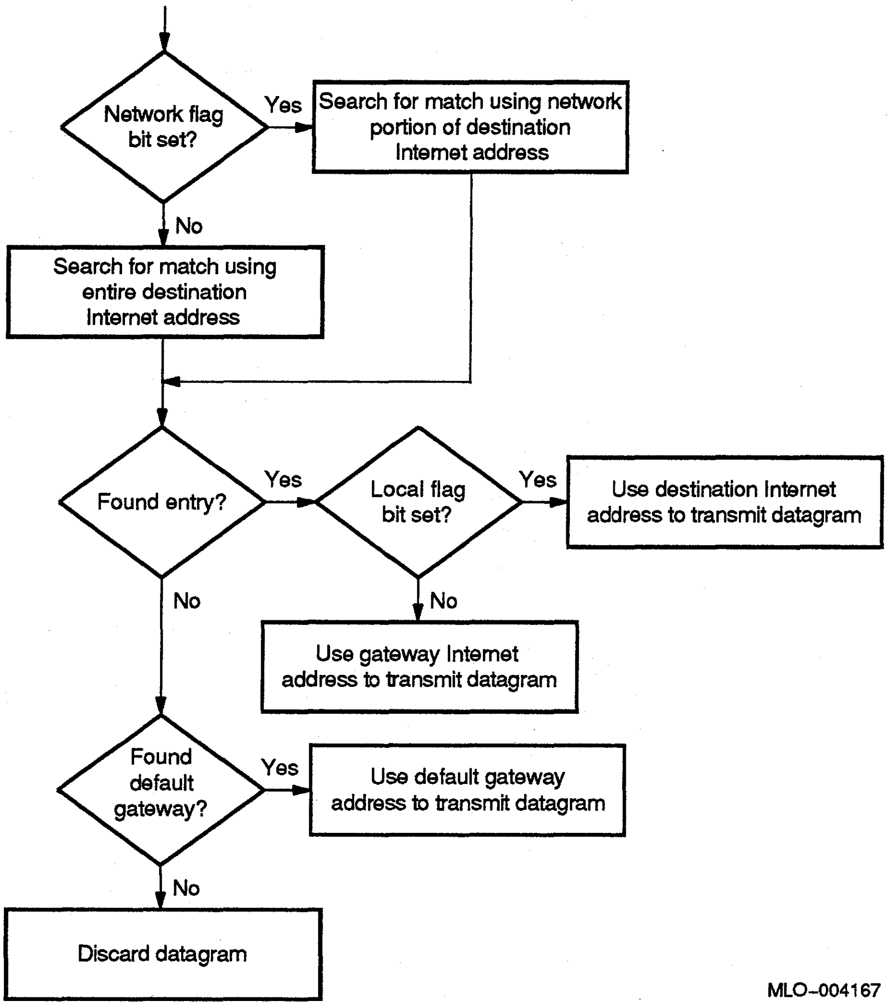

Extract Internet address from datagram
Broadcast address?
Yes
Broadcast datagram over the Ethernet using the address FF--FF--FF--FF--FF--FF
No
Local host's Internet address?
Yes
Loop datagram back to port on local host
No
Local network address?
Yes
Transmit on the local network
No
Search routing table for destination Internet address
MLO–004166

Figure 10–4 Cont'd on next page

Figure 10–4 (Cont.): Routing Algorithm


Network flag bit set?
Yes
Search for match using network portion of destination Internet address
No
Search for match using entire destination Internet address
Found entry?
Yes
Local flag bit set?
Yes
Use destination Internet address to transmit datagram
No
Use gateway Internet address to transmit datagram
No
Found default gateway?
Yes
Use default gateway address to transmit datagram
No
Discard datagram
MLO–004167

IP first checks whether the destination Internet address equals a broadcast address, or the local host's Internet address. If the address is a broadcast address, IP broadcasts the datagram over the Ethernet using the FF–FF–FF–FF–FF–FF Ethernet address. A destination Internet address is considered a broadcast address if one of the following conditions applies:

- The address is 255.255.255.255
- The network part of the address matches the network part of the local host's Internet address and the logical OR of the destination address and the network mask equals 255.255.255.255
- The address is local and the logical OR of the address and the subnet mask equals 255.255.255.255

NOTE

IP cannot check for a broadcast address if it has not yet determined the local Internet address.

If the destination Internet address is equal to the local host's Internet address, IP loops the datagram back to a port on the local host.

If the destination Internet address is not a broadcast address or the local host's Internet address, IP checks whether the address is local and if so, sends the datagram to a node on the local network. The destination Internet address is local if one of the following applies:

- The subnet mask is 0
- The logical AND of the destination Internet address and the subnet mask equals the logical AND of the local host's Internet address and the subnet mask

If the destination Internet address is not local, IP searches the routing table for a matching address. If the network flag is set, IP uses only the network portion of the address when checking for a match. Otherwise, IP uses the entire address for the search.

If IP finds an entry for the address, the protocol checks the state of the local flag. If the flag is set, IP uses the destination Internet address to route the datagram to a host on the local network. Otherwise, IP uses the gateway address in the table entry to route the datagram to a gateway on the local network.

If IP does not find an entry in the routing table for the destination address, IP checks the table for a default gateway. If IP does not find a default, the protocol discards the datagram.

An application can manage a routing table at runtime by using Internet service routines. For more information, see Section 10.3.2.

10.1.7 Fragmentation

IP fragments a datagram when a datagram originates in a local network that allows a large packet size and must traverse a local network that limits packets to a smaller size to reach its destination, IP may also fragment a datagram when no gateway exists and applications send messages that are greater in length than the network layer supports.

A gateway can fragment an Internet datagram into smaller Internet datagrams. The gateway produces a set of Internet datagrams, each carrying a fragment. If necessary, subsequent gateways can break down the fragments into smaller fragments.

The fragment format is designed so that the destination IP can reassemble fragments into datagrams.

10.2 Configuring Internet Services

To use the Internet Services, you must build the appropriate datalink driver and the Internet Services into your VAXELN system. You configure the Internet Services for a system by selecting the **Edit Internet Service Characteristics** entry on the System Builder's Main Menu. When you select this entry, the System Builder displays two menu options: **Edit Internet Characteristics** and **Edit Internet Network Description**. The Internet Characteristics Menu lets you define systemwide Internet characteristics. You must use the Internet Network Description Menu to provide an Internet network description for the Ethernet controller that is to use the Internet Services.

You include the Internet Services in a VAXELN system by selecting *Yes* for the **Internet Services** entry on the Internet Characteristics Menu. This menu defines the following general systemwide Internet characteristics:

- Maximum number of ARP cache entries
- Maximum number of routing table entries
- Maximum number of bytes in an Internet datagram

- Default gateway

You also can use the Internet Characteristics Menu to define the following systemwide TCP characteristics:

- Maximum number of octets in a segment
- Default number of octets in the sliding window
- Maximum number of octets in the sliding window
- Number of seconds to wait for a connection
- Number of seconds a connection should linger after it is closed
- Number of seconds to wait for a connection acknowledgment
- Number of seconds to wait for message acknowledgments
- Maximum number of message resends

You provide an Internet network interface description for an Ethernet controller in your system by editing the Internet Network Description Menu. Using this menu, you specify the following controller information:

- Name
- Internet address
- Internet network mask
- Broadcast mask
- Address resolution method
- Whether the Internet Services are to determine the network mask
- Number of seconds to wait for the Internet address before timing out (0 indicates no timeout)
- Number of seconds to wait for the Internet network mask before timing out (0 indicates no timeout)

If you include the Internet Services in a VAXELN system, an application program can use runtime routines to control the Internet Services, convert byte order Internet and host physical addresses, communicate over the Internet, and retrieve and set socket characteristics.

For descriptions of the Internet Service routines, see the *VAXELN C Reference Manual* and *VAXELN C Runtime Library Reference Manual*. For more information about building the Internet Services into VAXELN systems, see the *VAXELN Development Utilities Guide*.

10.3 Controlling Internet Services

A VAXELN application can use Internet Service control routines to manage the ARP cache, routing table, and Internet network interfaces dynamically at runtime. Control routines also provide a means for retrieving IP, UDP, and TCP statistics and connection information.

10.3.1 Managing the ARP Cache

ARP maps Internet addresses to Ethernet addresses and stores the address pairs in an ARP cache. A host searches its ARP cache for an Internet address binding. If the host does not find the binding, the host broadcasts the target host's Internet address to all hosts on the network. The target host recognizes its Internet address and responds by returning its physical address to the requesting host.

The VAXELN Internet Services provide the following Internet network control routines for managing a host's ARP cache:

Routine	**Description**
ELN$INET_DELETE_ARP_ENTRY	Deletes an entry from the ARP cache.
ELN$INET_FIND_ARP_ENTRY	Returns an Ethernet address from the ARP cache.
ELN$INET_SET_ARP_ENTRY	Adds an entry to the ARP cache.
ELN$INET_SHOW_ARP_ENTRIES	Returns the entries currently stored in the ARP cache.

For information about Internet addresses, see Section 10.1.3. Sections 10.3.1.1 to 10.3.1.3 explain how to use the Internet control routines to do the following:

- Add and delete ARP cache entries
- Retrieve Ethernet addresses from the ARP cache
- Retrieve ARP cache entries

10.3.1.1 Adding and Deleting ARP Cache Entries

An application can add entries to and delete entries from a host's ARP cache by calling the ELN$INET_SET_ARP_ENTRY and ELN$INET_DELETE_ARP_ENTRY routines.

A call to ELN$INET_SET_ARP_ENTRY maps an Internet address to an Ethernet address and places the mapping in the cache. The call must specify an Internet address, Ethernet address, and an ARP option. The Internet address must be the Internet address of the host on which the Internet interface resides. The Ethernet address is the target interface address that the routine maps to the Internet address. The Ethernet address cannot be a multicast address.

The option argument specifies whether an entry is permanent. A permanent entry can be deleted only with a call to the ELN$INET_DELETE_ARP_ENTRY routine. However, ARP requests can continue to update entries marked with this option.

You can set or clear the permanent option using a Boolean or bit mask value. If you choose the Boolean method, set the permanent field of the INET$SET_ROUTE_OPTIONS aggregate to TRUE. When using the bit mask method, specify the mask name INET$ARP_PERMANENT_MASK for the aggregate's mask value field.

You should delete an ARP cache entry when the entry is no longer needed. To delete an entry from the ARP cache, specify the host Internet address of the entry to be deleted in a call to the ELN$INET_DELETE_ARP_ENTRY routine. If ARP does not find a cache entry for the specified Internet address, the routine returns an error.

NOTE

The Internet address that you specify in a call to ELN$INET_SET_ARP_ENTRY or ELN$INET_DELETE_ARP_ENTRY cannot be the Internet address of a network interface.

The following function adds and deletes an ARP cache entry:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void add_and_delete_arp_entry()
{
  long int status;
  INET$INTERNET_ADDRESS internet_address;
  INET$ETHERNET_ADDRESS ethernet_address;
  INET$SET_ARP_OPTIONS options;

  /* Get an Internet address, Ethernet address, and options. */

  internet_address.S_un.S_addr = get_ia("Internet address: ");
  get_epa("Ethernet address: ", &ethernet_address);
  options.mask_value = get_ulong("Options (1=NODELETE): ");

  /* Add the input to the ARP cache. */

  eln$inet_set_arp_entry(&status,
                         &internet_address,
                         &ethernet_address,
                         &options);
  if (!(status & 1))
    disp_status(status);

  /* When the entry is no longer needed, delete it. */

  eln$inet_delete_arp_entry(&status,
                            &internet_address);

  if (!(status & 1))
    disp_status(status);
}
```

10.3.1.2 Retrieving Ethernet Addresses from the ARP Cache

An application can retrieve the Ethernet address that corresponds to an Internet address by calling the ELN$INET_FIND_ARP_ENTRY routine. A call to this routine must specify an Internet address and the variable that is to receive the corresponding Ethernet address. If ARP does not find a cache entry for the specified Internet address, the routine returns an error.

The following function retrieves the Ethernet address that corresponds to a specified Internet address:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void find_arp_entry()
{
  long int status;
  INET$INTERNET_ADDRESS internet_address;
  NET$ETHERNET_ADDRESS  ethernet_address;

  /* Get an Internet address. */

  internet_address.S_un.S_addr = get_ia("Internet address: ");

  /* Find the entry for the specified address. */

  eln$inet_find_arp_entry(&status,
                          &internet_address,
                          &ethernet_address);

  if (!(status & 1))
    disp_status(status);
  else
    {
      printf ("\nArp entry for %s", format_ia(internet_address));
      printf (" is %s\n\n", format_epa (&ethernet_address));
    }
```

10.3.1.3 Retrieving ARP Cache Entries

An application can retrieve all the cache entries currently stored in a host's ARP cache by calling the ELN$INET_SHOW_ARP_ENTRIES routine. A call to this routine must specify the name of a user-defined routine that returns ARP entry information. ELN$INET_SHOW_ARP_ENTRIES invokes the user-defined routine once for each entry in the cache. If the ARP cache is empty, ELN$INET_SHOW_ARP_ENTRIES returns an error.

The user-defined routine returns the cache data to an aggregate called INET$ARP_ENTRY. A program can then extract the following information:

- Internet address
- Ethernet address
- ARP status information

The ARP status information is returned to a flag field of bits that indicate whether the entry is permanent, in use, and complete. An application can set the permanent bit in calls to ELN$INET_SET_ARP_ENTRY. ARP sets the in use bit when the host broadcasts an entry's Internet address. When a target host returns its physical address to the requesting host, ARP sets the complete bit.

Once the ARP entry data is returned, a program can examine and manipulate the data using the field names *internet_address*, *ethernet_address*, and *arp_status*. You can examine the ARP status bits individually using Boolean values, or collectively, using bit mask values. If you choose the Boolean method, examine the bits using aggregate field names. When using the bit mask method, specify one or more mask values. The status fields and mask values are defined as follows:

Status	**Field Name**	**Mask Name**	**Mask Value**
Permanent	*permanent_field*	ARP_PERMANENT_MASK	1
In use	*inuse_field*	ARP_INUSE_MASK	2
Complete	*complete_field*	ARP_COMPLETE_MASK	4

You can also manipulate groups of status values by specifying the sum of the appropriate mask values for the *mask_value* field.

The following code shows an example of how an application might use ELN$INET_SHOW_ARP_ENTRIES:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void show_arp_entries()
{
  char ch;
  long int status;
  FUNCTION_DESCRIPTOR fn_desc;
  void show_arp_entry();

  version_displayed = FALSE;

  /* Show the entries that are in the ARP cache. */

  eln$inet_show_arp_entries(&status,
           ELN$PASS_FUNCTION_DESCRIPTOR(fn_desc, show_arp_entry));
  if (!(status & 1))
    disp_status(status);
  else
    ch = get_char ("\nPress <RETURN> to continue.\n");
}

INET$SHOW_ARP_ENTRY(show_arp_entry)
{
  BOOLEAN parenthesis_displayed = FALSE;

  if (!version_displayed)
    {
      version_displayed = TRUE;
      printf("ARP Information version number is: %d\n\n", version);
    }

  printf ("%s", format_ia(entry->internet_address));
  printf (" => %s", format_epa (&entry->ethernet_address));

  if (entry->arp_status.mask_value)
    {
      if (entry->arp_status.fields.permanent_field)
        {
          parenthesis_displayed = TRUE;
          printf(" (PERM");
        }

      if (entry->arp_status.fields.inuse_field)
        {
          if (parenthesis_displayed)
            printf(",INUSE");
          else
            {
              parenthesis_displayed = TRUE;
              printf(" (INUSE");
            }
        }

      if (entry->arp_status.fields.complete_field)
        {
   if (parenthesis_displayed)
            printf(",COMPL");
          else
            {
              parenthesis_displayed = TRUE;
              printf(" (COMPL");
            }
        }
      printf (")");
    }
  printf ("\n");
}
```

10.3.2 Managing the Internet Routing Table

IP maps Internet routes (addresses) to host and network addresses and stores the address pairs in a routing table. The VAXELN Internet Services provide the following Internet network control routines for managing the Internet routing table:

Routine	Description
ELN$INET_CHECK_ROUTE	Searches for a route to a specified Internet address.
ELN$INET_DELETE_ROUTE	Deletes an entry from the routing table.
ELN$INET_SET_ROUTE	Adds an entry to the routing table.
ELN$INET_SHOW_ROUTES	Returns the entries currently stored in the routing table.

For more information about routing, see Section 10.1.6. Sections 10.3.2.1 to 10.3.2.3 explain how to use the Internet control routines to do the following:

- Add and delete routing table entries
- Checking the status of routing table entries
- Retrieve routing table entries

10.3.2.1 Adding and Deleting Routing Table Entries

An application can add entries to and delete entries from an Internet routing table by calling the ELN$INET_SET_ROUTE and ELN$INET_DELETE_ROUTE routines.

A call to ELN$INET_SET_ROUTE maps a routing path to a host or network and places the mapping in the table. The call must specify an Internet address, gateway address, and route options. The Internet address must be the host or network destination address. The gateway address is the Internet address of the gateway host.

The options argument is an aggregate of bit fields that specify the following:

- Whether the entry is for a network or host route
- Whether the route is to a host or gateway

- Whether the route is locked (can be updated by ICMP redirect messages)

You can set or clear the route entry options individually, using Boolean values, or collectively, using bit mask values. If you choose the Boolean method, you set the Boolean value for the appropriate aggregate fields to TRUE or FALSE, as appropriate. When using the bit mask method, specify the sum of the appropriate mask values in the mask field. The option fields and mask values are defined as follows:

Option	**Field Name**	**Mask Name**	**Mask Value**
Search for network address	*network_field*	INET$ROUTE_NETWORK_MASK	1
Local route	*local_field*	INET$ROUTE_LOCAL_MASK	2
Lock route	*lock_field*	INET$ROUTE_LOCK_MASK	4

If you do not specify options, the entry identifies a host route.

You can specify multiple route options by specifying the sum of the mask values for the desired options in the *mask_value* field. For example, to specify a network route that cannot be updated by ICMP redirect messages, use a mask value of 5 (the sum of mask values INET$ROUTE_NETWORK_MASK and INET$ROUTE_LOCK_MASK).

NOTE

You cannot add a route to the static routing table until the Internet address and Internet network mask are known.

You should mark a routing table entry for deletion when the entry is no longer needed. Once an entry is marked, the Internet software can delete the entry when it is no longer in use. The Internet software uses a reference count to determine whether an entry is being used. A reference count is maintained for each entry in the routing table.

To mark an entry for deletion, specify the Internet address of the entry to be deleted in a call to the ELN$INET_DELETE_ROUTE routine. If the Internet software does not find an entry for the specified Internet address, the routine returns an error.

You must also specify an option argument in calls to ELN$INET_DELETE_ROUTE. The route option indicates whether the Internet software is to delete a network or host route.

The following function adds a route to the routing table and then deletes the route when it is no longer needed:

```
#include $vaxelnc
#include $internet_utility
   .
   .
   .
void set_route()
{
  long int                   status;
  INET$INTERNET_ADDRESS      internet_address;
  INET$INTERNET_ADDRESS      gateway_address;
  INET$SET_ROUTE_OPTIONS     set_options;
  INET$DELETE_ROUTE_OPTIONS  del_options;
   .
   .
   .
  /* Get an Internet address, gateway address, and the options to */
  /* be set.                                                      */

  internet_address.S_un.S_addr = get_ia("Internet address: ");
  gateway_address.S_un.S_addr = get_ia("Gateway address: ");
  set_options.mask_value =
    get_ulong("Options BITMASK (1=NETWRK, 2=LOCAL, 4=LOCK): ");

  /* Add the input to the routing table. */

  eln$inet_set_route(&status,
                     &internet_address,
                     &gateway_address,
                     &set_options);
  .
  .
  .
  /* When the routing table entry is no longer needed, mark it for */
  /* deletion.                                                     */

  del_options.mask_value = get_ulong("Options BITMASK (1=NETWRK): ");

  eln$inet_delete_route(&status,
                        &internet_address,
                        &del_options);
}
```

10.3.2.2 Checking the Status of Routing Table Entries

An application can check the status of a routing table entry by calling the ELN$INET_CHECK_ROUTE routine. Using this routine an application can check whether an entry contains a network or local route, can be updated by ICMP redirect messages, or is marked for deletion but is still in use.

A call to ELN$INET_CHECK_ROUTE must specify an Internet address, a credit value, a routing table entry returned by a previous call to ELN$INET_CHECK_ROUTE, and variables that are to receive the gateway address and the routing table entry status value.

The Internet address identifies the entry for which the Internet software is to return the status information. If the Internet software does not find an entry for the specified Internet address, the routine returns an error.

The Internet software uses reference counts for the table entries to prevent a route from being deleted while it is being used. The credit argument specifies whether or not the table entry's reference count is to be updated. You must specify one of the following credit values:

Credit Value	Effect
0	Reference count remains unchanged.
1	Increments reference count.
−1	Decrements reference count.

If you specify a credit value of −1, ELN$INET_CHECK_ROUTE uses the routing table entry argument. This argument specifies an entry returned by a previous call to ELN$INET_CHECK_ROUTE and receives the table entry number for the specified Internet address's route upon successful completion.

The gateway address and route status arguments receive the destination Internet address and the route status, respectively.

The route status indicates whether the entry is for a network route, whether the route is to a host, whether ICMP redirect messages can update the entry, and whether the entry is marked for deletion. The status information is returned as an aggregate of bits. You can examine and manipulate the status bits individually, using aggregate field names, or collectively, using one or more mask values. The status fields and mask values are defined as follows:

Status	Field Name	Mask Name	Mask Value
Searched for network address	*network_field*	INET$ROUTE_NETWORK_MASK	1
Used host route	*local_field*	INET$ROUTE_LOCAL_MASK	2
Route is locked	*lock_field*	INET$ROUTE_LOCK_MASK	4
Route is marked for deletion	*deleted_field*	INET$ROUTE_DELETED_MASK	8

If multiple status values apply to an entry, the Internet software adds the mask values of the appropriate status values and returns the sum.

The following function checks the status of a routing table entry and displays the status information:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void check_route()
{
  long int status;
  INET$INTERNET_ADDRESS internet_address;
  INET$INTERNET_ADDRESS gateway_address;
  INET$ROUTE_STATUS route_status;
  short int credit;
  unsigned long int rte;
  BOOLEAN parenthesis_displayed = FALSE;

  credit = 0;
  rte = 0;

  /* Get an Internet address. */

  internet_address.S_un.S_addr = get_ia("Internet address: ");

  /* Search the routing table for an entry for the specified address. */

  eln$inet_check_route(&status,
                       &internet_address,
                       credit,
                       &rte,
                       &gateway_address,
                       &route_status);
```

```
/* If an entry is found, display the data. */

if (!(status & 1))
  disp_status(status);
else
  {
    printf ("Route for %s", format_ia(internet_address));
    printf (" is %s", format_ia(gateway_address));
    if (route_status.mask_value)
      {
        if (route_status.fields.network_field)
          {
            parenthesis_displayed = TRUE;
            printf(" (NETWRK")
          }

        if (route_status.fields.local_field)
          {
            if (parenthesis_displayed)
              printf(",LOCAL");
            else
              {
         parenthesis_displayed = TRUE;
                printf(" (LOCAL");
              }
          }

 if (route_status.fields.lock_field)
          {
            if ( parenthesis_displayed )
              printf(",LOCKED");
            else
              {
                parenthesis_displayed = TRUE;
                printf(" (LOCKED");
       }
          }

        if (route_status.fields.deleted_field)
          {
            if (parenthesis_displayed)
              printf(",DELETED");
            else
              {
                parenthesis_displayed = TRUE;
                printf(" (DELETED");
              }
          }

        printf (")");
      }

    printf ("\n");
  }
}
```

10.3.2.3 Retrieving Routing Table Entries

An application can retrieve all the entries in the Internet routing table by calling the ELN$INET_SHOW_ROUTES routine. A call to this routine must specify the name of a user-defined routine that returns routing table entry information. ELN$INET_SHOW_ROUTES invokes the user-defined routine once for each entry in the table. If the table is empty, ELN$INET_SHOW_ROUTES returns an error.

The user-defined routine returns the routing data to an aggregate called INET$ROUTE_ENTRY. A program can then extract the following information:

- Destination Internet address
- Gateway address
- Route status
- Reference count
- Usage count

The route status information is returned to a flag field of bits that indicate whether IP is to search for a match using only the network portion of the destination Internet address, the route is to a host, the route is locked, and the route is marked for deletion. An application can set the deleted bit in calls to ELN$INET_SET_ROUTE. IP sets the deleted bit when an application calls ELN$INET_DELETE_ROUTE.

Once the route entry data is returned, a program can examine and manipulate the data using the field names *destination_address*, *gateway_address*, *route_status*, *reference_count*, and *use_count*. You can examine the route status bits individually, using Boolean values, or collectively, using bit mask values. If you choose the Boolean method, examine the bits using aggregate field names. When using the bit mask method, specify one or more mask values. The status fields and mask values are defined as follows:

Status	**Field Name**	**Mask Name**	**Mask Value**
Searched for network address	*network_field*	INET$ROUTE_NETWORK_MASK	1
Used host route	*local_field*	INET$ROUTE_LOCAL_MASK	2

Status	Field Name	Mask Name	Mask Value
Route is locked	*lock_field*	INET$ROUTE_LOCK_MASK	4
Route is marked for deletion	*deleted_field*	INET$ROUTE_DELETED_MASK	8

You can also manipulate groups of status values by specifying the sum of the appropriate mask values for the *mask_value* field.

The following function shows the contents of the routing table:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void show_routes()
{
  char ch;
  long int status;
  FUNCTION_DESCRIPTOR fn_desc;
  void show_route_entry();
  version_displayed = FALSE;

  /* Show the routing table entries. */

  eln$inet_show_routes(&status,
                       ELN$PASS_FUNCTION_DESCRIPTOR(fn_desc,
                       show_route_entry));

  if (!(status & 1))
    disp_status (status);
  else
    ch = get_char("\nPress <RETURN> to continue.\n");
    .
    .
    .
}

INET$SHOW_ROUTE_ENTRY(show_route_entry)
{
  BOOLEAN parenthesis_displayed = FALSE;

  if (!version_displayed)
    {
      version_displayed = TRUE;
      printf("Route information version number is: %d\n\n", version);
    }

  printf ("%s", format_ia(entry->destination_address));
  printf (" => %s", format_ia (entry->gateway_address));
  printf (" REFCNT: %d USECNT: %d", entry->reference_count,
          entry->use_count);
```

```
 if (entry->route_status.mask_value)
  {
   if (entry->route_status.fields.network_field)
    {
     parenthesis_displayed = TRUE;
     printf(" (NETWRK");
    }

   if (entry->route_status.fields.local_field)
    {
     if (parenthesis_displayed)
      printf(",LOCAL");
     else
      {
       parenthesis_displayed = TRUE;
       printf(" (LOCAL");
      }
    }

   if (entry->route_status.fields.lock_field)
    {
     if (parenthesis_displayed)
      printf(",LOCKED");
     else
      {
       parenthesis_displayed = TRUE;
       printf(" (LOCKED");
      }
    }

   if (entry->route_status.fields.deleted_field)
    {
     if (parenthesis_displayed)
      printf(",DELETED");
     else
      {
    parenthesis_displayed = TRUE;
    printf(" (DELETED");
   }
   printf (")");
  }
 printf ("\n");
}
```

10.3.3 Managing Internet Network Interfaces

The VAXELN Internet Services provide the following Internet network control routines for managing Internet network interfaces:

Routine	Description
ELN$INET_SET_INTERFACE	Associates an Internet address with the name of an Internet network interface that resides on the VAXELN target system.
ELN$INET_SHOW_INTERFACE	Returns the Internet network characteristics for Internet interfaces.

For more information about setting up Internet network interfaces for VAXELN systems, see the *VAXELN Development Utilities Guide*. Sections 10.3.3.1 and 10.3.3.2 explain how to use the Internet control routines to do the following:

- Set an Internet network interface dynamically at runtime
- Retrieve Internet network interface characteristics

10.3.3.1 Setting Internet Network Interfaces

If you did not specify an Internet address for an Internet network interface when you built your system, you can do so at runtime by calling the ELN$INET_SET_INTERFACE routine. You can also use this routine to set an interface's broadcast and network masks once.

A call to ELN$INET_SET_INTERFACE must specify the name of the communication interface that is to be associated with an Internet address. You must also specify a new fields argument and values for the Internet address, broadcast mask, and network mask arguments.

The new fields argument is an aggregate that identifies characteristics that you intend to set. You set characteristics by setting the appropriate bits in the aggregate. You can specify the aggregate fields to be set individually, using Boolean values, or collectively, using a bit mask value. If you choose the Boolean method, you set the Boolean value for aggregate fields to TRUE, as appropriate. If you choose the bit mask method, you specify the sum of the appropriate mask values in the mask value field. The interface characteristics fields and mask values are defined as follows:

Characteristic	Field Name	Mask Name	Mask Value
Internet address	*internet_address_field*	INET$INTERNET_ADDR_MASK	1
Broadcast address	*broadcast_address_field*	INET$BROADCAST_MASK	2
Network mask	*network_mask_field*	INET$NETWORK_MASK	4

You can set fields without changing other fields in the aggregate. However, for each field that you set, you must specify a value for a corresponding argument. For example, to set the Internet address, you must set the bit for the Internet address field in the fields argument and specify the Internet address for the Internet address argument. For the fields that you choose not to set, you can specify a null string for the corresponding argument.

The following section of code shows how an application might use ELN$INET_SET_INTERFACE:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void show_interface()
{
  char ch;
  int status;
  VARYING_STRING(32) interface_name;
  INET$SET_INTERFACE_FIELDS new_fields;
  INET$INTERNET_ADDRESS internet_address;
  INET$INTERNET_ADDRESS network_mask;
  INET$INTERNET_ADDRESS broadcast_mask;
  INET$SET_INTERFACE_OPTIONS options;

  /* Get interface input. */

  new_fields.mask_value = 0;
  get_varying_string("Interface to set: ", 32, &interface_name);

  ch = get_char("Set Interface address? Y or N: [N] ");
  if (toupper(ch) == 'Y')
    {
      new_fields.mask_value += INET$INTERNET_ADDR_MASK;
      internet_address.S_un.S_addr = get_ia ("Internet address: ");
    }
  ch = get_char("Set Address (subnet) mask address? Y or N: [N] ");
  if (toupper(ch) == 'Y')
    {
      new_fields.mask_value += INET$NETWORK_MASK;
      network_mask.S_un.S_addr = get_ia ("Address mask: ");
    }

  ch = get_char ("Set Broadcast mask address? Y or N: [N] ");
  if (toupper(ch) == 'Y')
    {
      new_fields.mask_value += INET$BROADCAST_MASK;
      broadcast_mask.S_un.S_addr = get_ia ("Broadcast mask: ");
    }

  options.mask_value = get_ulong("Interface options (reserved) : ");

  /* Set the interface. */

  eln$inet_set_interface(&status,
                         &interface_name,
                         &new_fields,
                         &internet_address,
                         &network_mask,
                         &broadcast_mask,
                         &options);

  if (!(status &1))
    disp_status (status);
}
```

You must also specify an interface options argument. This argument is reserved for future use.

10.3.3.2 Retrieving Internet Network Interface Characteristics

An application can retrieve the characteristics for all Internet network interfaces on a VAXELN system by calling the ELN$INET_SHOW_INTERFACES routine. A call to this routine must specify the name of a communication interface and the name of a user-defined routine.

The interface name identifies the interface for which information is to be returned. To return information about all network interfaces, specify an asterisk (*). If you specify a name that has not been defined, the routine returns an error.

The user-defined routine returns the network interface information. ELN$INET_SHOW_INTERFACES invokes the user-defined routine for the specified interface. If you specify an asterisk, ELN$INET_SHOW_INTERFACES invokes the routine once for each interface defined for the system. If no interfaces are defined, ELN$INET_SHOW_ROUTES returns an error.

Once the program retrieves the interface data, it can extract the following information:

- Interface name
- Interface state
- Internet address
- Ethernet address
- Network mask
- Broadcast mask
- Number of IP datagrams received
- Number of IP datagrams transmitted
- Number of trailer datagrams received
- Number of trailer datagrams transmitted
- Number of ARP datagrams received
- Number of ARP datagrams transmitted
- Number of ICMP datagrams received
- Number of ICMP datagrams transmitted
- Number of receive errors
- Number of transmit errors

The following code shows the characteristics for a specified interface:

```
#include $vaxelnc
#include $internet_utility
       .
       .
       .
void show_interface()
{
  char ch;
  int status;
  FUNCTION_DESCRIPTOR fn_desc;
  void show_interface_entry();
  VARYING_STRING(32) interface_name;

  /* Get the name of an interface. */

  get_varying_string("Interface name: [* for all] ",
                     32,
                     &interface_name);

  eln$inet_show_interface(&status,
           &interface_name,
           ELN$PASS_FUNCTION_DESCRIPTOR(fn_desc, show_interface_entry));
  if (!(status & 1))
    disp_status (status);
}

INET$SHOW_INTERFACE_ENTRY(show_interface_entry)
{
  char ch;

  printf("\nInterface:        %.*s\n",
         entry->interface_name.string_length,
         &entry->interface_name.string_text);
  printf("Interface State:  %d\n",
          entry->interface_state.mask_value);
  printf("Internet Address: %s\n",
         format_ia(entry->internet_address));
  printf("Ethernet Address: %s\n",
         format_epa (&entry->ethernet_address));
  printf("Address Mask:     %s\n",
         format_ia(entry->network_mask));
  printf("Broadcast Mask:   %s\n",
         format_ia(entry->broadcast_mask));
  printf("\n\n Counters: (version %d):\n\n", version);
  printf("              RECEIVED     TRANSMITTED\n");
  printf("IP Packets    %10u  %10u\n", entry->ip_rcvd,entry->ip_xmit);
  printf("IP Trailer 1 %10u       --\n", entry->trailer1_rcvd);
  printf("IP Trailer 2 %10u       --\n", entry->trailer2_rcvd);
  printf("ARP Packets  %10u  %10u\n", entry->arp_rcvd, entry->arp_xmit);
  printf("ICMP Packets %10u  %10u\n", entry->icmp_rcvd, entry->icmp_xmit);
  printf("Errors       %10u  %10u\n", entry->errors_rcvd,
         entry->xmit_errors);

  ch = get_char ("\nPress <RETURN> to continue.\n");
}
```

10.3.4 Retrieving Internet Performance and Error Data

The VAXELN Internet Services provide the following Internet network control routines for retrieving data concerning performance and errors:

Routine	Description
ELN$INET_SHOW_IP_STATISTICS	Returns performance and error statistics for IP.
ELN$INET_SHOW_TCP_STATISTICS	Returns performance and error statistics for TCP.
ELN$INET_SHOW_UDP_STATISTICS	Returns performance and error statistics for UDP.

The ELN$INET_SHOW_IP_STATISTICS, ELN$INET_SHOW_UDP_STATISTICS, and ELN$INET_SHOW_TCP_STATISTICS routines return performance and error statistics for IP, UDP, and TCP, respectively. These routines allocate a statistics aggregate for the appropriate protocol. The application program can then extract the following information from the aggregate:

IP	UDP	TCP
Transmission time in seconds	Transmission time in seconds	Transmission time in seconds
Number of packets received	Datagrams transmitted	Connection requests forwarded
IP datagram received	Datagrams received	Connections accepted
Received IP datagram has bad size	Invalid transmit	Connection requests issued
Received IP datagram has bad checksum	Invalid receive	Open connections
Received IP datagram has bad destination address	Datagrams received but not delivered	Connections reset
Received IP datagram includes disabled IP protocol		Segments transmitted
Received IP datagram fragmented		Segments retransmitted
Received IP datagram fragments dropped		Segments received
Received fragmented IP datagram reassembled		Invalid segments received
ICMP datagram received		Out-of-sequence segments received
Received ICMP datagram has bad size		Concatenated record descriptor buffers
Received ICMP datagram has bad checksum		
ARP datagram received		
Received ARP datagram replies		
Received ARP datagram requests		

IP	UDP	TCP
Trailers received		
Trailers received invalid		
Packets transmitted		
Packet trasmissions that failed		
Transmitted IP datagram invalid		
Transmitted IP datagram has bad destination address		
IP datagram transmitted		
Transmitted IP datagram fragmented		
Transmitted IP datagram fragments		
ICMP datagram transmitted		
ARP datagram transmitted		
Transmitted ARP datagram replies		
Transmitted ARP datagram requests		

A call to ELN$INET_SHOW_IP_STATISTICS, ELN$INET_SHOW_UDP_STATISTICS, or ELN$INET_SHOW_TCP_STATISTICS must specify a Boolean flag that indicates whether counters are to be cleared after they are read and variables that receive a version number and a pointer to the appropriate protocol statistics aggregate. The version number identifies the version of the statistics aggregate that the routine returns to the statistics argument.

When you finish accessing a statistics record, you must use the **free** function to deallocate it.

The following code shows how an application might retrieve TCP statistics:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void show_tcp_statistics()
{
  int status;
  int version;
  char ch;
  BOOLEAN clear_counters = FALSE;
  INET$TCP_STATISTICS *statistics;

  /* Check whether counters should be cleared. */

  ch = get_char("Clear Counters? Y or N: [N] ");
  if (toupper(ch) == 'Y')
    clear_counters = TRUE;

  /* Show all TCP statistics. */

  eln$inet_show_tcp_statistics(&status,
                               clear_counters,
                               &version,
                               &statistics);

  if (status & 1)
    {
      printf ("TCP Statistics Version     %10u\n",
              version);
      printf ("Seconds                    %10u\n",
              statistics->seconds);
      printf ("Connections forwarded:     %10u\n",
              statistics->connects_forwarded);
      printf ("Connections accepted:      %10u\n",
              statistics->connects_accepted);
      printf ("Connections issued:        %10u\n",
              statistics->connects_issued);
      printf ("Connections opened:        %10u\n",
              statistics->connects_opened);
      printf ("Connections reset:         %10u\n",
              statistics->connects_reset);
      printf ("Segments transmitted:      %10u\n",
              statistics->segments_xmit);
      printf ("Segments retransmitted:    %10u\n",
              statistics->segments_rexmit);
      printf ("Segments received:         %10u\n",
              statistics->segments_rcvd);
      printf ("Invalid segments received: %10u\n",
              statistics->invalid_rcvd);
      printf ("Out of sequence received:  %10u\n",
              statistics->out_of_sequence_rcvd);
      printf ("Concatenated messages:     %10u\n",
              statistics->concatenated_rdbs);
```

```
        free(statistics);
        ch = get_char ("\nPress <RETURN> to continue\n");
      }
     else
       disp_status(status);
}
```

10.3.5 Retrieving TCP Connection Data

An application can retrieve data concerning active TCP connections by calling the ELN$INET_SHOW_CONNECTIONS routine. This routine does not report information about listening servers (applications waiting on a connection).

A call to ELN$INET_SHOW_TCP_CONNECTIONS must specify the name of a user-defined routine to be invoked by ELN$INET_SHOW_TCP_CONNECTIONS once for each active TCP connection. The user-defined routine returns the connection information. If no connections exist, ELN$INET_SHOW_TCP_CONNECTIONS returns an error.

Once the program retrieves the connection data, it can extract the following information:

- Local Internet address
- Local port number
- Remote Internet address
- Remote port number
- Connection state
- Connection options
- Number of messages in the receive queue
- Number of messages in the send queue
- Number of urgent messages received
- Number of urgent messages sent
- Number of messages in the receive window
- Number of messages in the send window
- Send sequence number
- Acknowledgment sequence number
- Retransmit timer value
- Persist timer value

- Keep-alive timer value
- Maximum linger timer value
- Number of retransmissions

The connection state and connection options information is returned to flag bit fields. For the connection state, the bits indicate whether:

- The connection is open
- The connection is listening
- The connection is waiting for a matching connection request
- The connection is waiting for a connection request ACK message after having received and transmitted a connection request
- The connection is established
- The connection is waiting for a connection termination request from a remote peer or an ACK message for the connection termination request previously sent
- The connection is waiting for a connection termination request from a remote peer
- The wait is closed
- The connection is being closed
- The last ACK has been sent
- The wait time expired

For the connection options, the bits indicate whether the connection is to have a linger time and keep-alive time.

A program can examine and manipulate the connection data using field names. You can examine or manipulate the state and option bits individually using Boolean values, or collectively, using bit mask values. If you choose the Boolean method, examine the bits using aggregate field names. When using the bit mask method, specify one or more mask values. The state fields and mask values are defined as follows:

State	Field Name	Mask Name	Mask Value
Connection is closed	*closed_field*	INET$TCP_STATE_CLOSED_ MASK	1
Connection is listening for requests	*listen_field*	INET$TCP_STATE_LISTEN_ MASK	2
Waiting for matching connection request	*syn_sen_field*	INET$TCP_STATE_SYN_SENT_ MASK	4
Waiting for ACK message after receive and transmit	*syn_rcvd_field*	INET$TCP_STATE_SYN_ RCVD_MASK	8
Connection is established	*established_field*	INET$TCP_STATE_ ESTABLISHED_MASK	16
Waiting for connection termination request from remote peer	*fin_wait_1_field*	INET$TCP_STATE_FIN_WAIT_ 1_MASK	32
Waiting for connection termination request from remote peer	*fin_wait_2_field*	INET$TCP_STATE_FIN_WAIT_ 2_MASK	64
Wait is closed	*close_wait_field*	INET$TCP_STATE_CLOSE_ WAIT_MASK	128
Connection is being closed	*closing_field*	INET$TCP_STATE_CLOSING_ MASK	256
Last ACK has been received	*last_ack_field*	INET$TCP_STATE_LAST_ACK_ MASK	512
Time to wait for connection expired	*last_ack_field*	INET$TCP_STATE_TIME_ WAIT_MASK	1024

The option fields and mask values are defined as follows:

Option	Field Name	Mask Name	Mask Value
No linger time	*nolinger_field*	INET$TCP_OPT_NOLINGER_ MASK	1
Linger time	*linger_field*	INET$TCP_OPT_LINGER_ MASK	2

Option	Field Name	Mask Name	Mask Value
No keep alive time	*nokeepalive_field*	INET$TCP_OPT_ NOKEEPALIVE_MASK	4
Keep alive time	*keepalive_field*	INET$TCP_STATE_ KEEPALIVE_MASK	8

You can also examine groups of status values by specifying the sum of the appropriate mask values for the *mask_value* field.

The following function shows how an application might use ELN$INET_SHOW_TCP_CONNECTIONS to retrieve information about active TCP connections:

```
#include $vaxelnc
#include $internet_utility
    .
    .
    .
void show_tcp_connections()
{
  char ch;
  int status;
  FUNCTION_DESCRIPTOR fn_desc;
  void show_tcp_connection_entry();

  /* Clear error in version displayed flag. */

  con_ver_err_dspld = FALSE;

  eln$inet_show_tcp_connections(&status,
             ELN$PASS_FUNCTION_DESCRIPTOR(fn_desc,
                                         show_tcp_connection_entry));

  if (!(status & 1))
    disp_status(status);
  else
    ch = get_char ("\nPress <RETURN> to continue.\n");

}
```

```
INET$SHOW_TCP_CONNECTION_ENTRY(show_tcp_connection_entry)
{
  if (version != INET$TCP_CONNECTION_VERSION )
    {
      if (!(con_ver_err_dspld))
        {
          printf("TCP connection entry version number is
                  unrecognized.\n");
          con_ver_err_dspld = TRUE;
        }
          return;
    }
      else
        {
          printf ("TCP Connection version: %10u\n", version);
          printf("Local IA:  %s  Local PN:  %u\n",
                  format_ia (entry->local_internet_address),
                  entry->local_port_number);
          printf("Remote IA: %s  Remote PN: %u  CCB ID: %X (hex)\n",
                  format_ia (entry->remote_internet_address),
                  entry->remote_port_number,
                  entry->ccb_id);
          printf("TCP state: %s, Options: %s\n",
                  format_tcp_state (entry->state),
                  format_options (entry->options));
          printf("Receive:  Q: %8u, Urg: %8u, Window: %6u\n",
                  entry->recv_queue,
                  entry->recv_urgent,
                  entry->recv_window);
          printf("Send:     Q: %8u, Urg: %8u, Window: %6u\n",
                  entry->send_queue,
                  entry->send_urgent,
                  entry->send_window);
          printf("Timers: Rexmit: %u, Prst: %u, Keep: %u, MSL: %u\n",
                  entry->rexmit_tmr,
                  entry->persist_tmr,
                  entry->keep_tmr,
                  entry->msl_tmr);
          printf("Rexmit value: %u, Snd seq: %12u, Ack seq: %12u\n",
                  entry->rexmit_value,
                  entry->snd_seq,
                  entry->ack_seq);
        }
}
```

10.4 Converting the Byte Order of Network and Host Binary Data

Not all hosts store bits the same way. To enable different types of hosts to communicate, regardless of how bits are represented, the Internet Services define a standard byte order for Internet packet binary fields. The standard *network byte order* places the byte with the most significant bits at the lower addresses. All hosts must use this format when sending data.

The VAXELN Internet Services provide the following routines for converting the byte order of network and host binary data:

Routine	Description
htonl	Converts a 32-bit unsigned integer from host byte order to network byte order.
htons	Converts a short integer from host byte order to network byte order.
ntohl	Converts a 32-bit unsigned integer from network byte order to host byte order.
ntons	Converts s short integer from network byte order to host byte order.

Before sending a message, a host must convert the byte order of binary data from its local representation to the standard network representation. An application can convert data to network byte order by calling **htonl** or **htons**. You specify **htonl** and **htons** with a longword or short integer, as appropriate, in host byte order. The functions return a longword or short integer in network byte order. You cannot use integers in network byte order for arithmetic computations on VAX systems.

When a host receives a message, it must convert the byte order of the message data to its byte-order representation. To convert data to the host's representation, call **ntohl** and **ntons**. Specify these functions with a longword or short integer, as appropriate, in network byte order. The functions return a longword or short integer in host byte order.

For descriptions of the conversion routines, see the *VAXELN C Reference Manual.*

10.5 Manipulating Internet Addresses

The VAXELN Internet Services provide a set of routines for manipulating Internet addresses. An application might use these routines while managing an ARP cache or for programming socket communication. The routines are as follows:

Routine	Description
inet_addr	Converts an Internet address in the standard text Internet "." notation to a numeric (binary) Internet address in network byte order.
inet_lnaof	Returns the local network (subnet) portion of an Internet address.
inet_makeaddr	Returns an Internet address given a network address and local (subnet and host) address on that network.
inet_netof	Returns the network portion of an Internet address.
inet_network	Converts an Internet address in the standard text Internet "." notation to a numeric (binary) Internet address in host byte order.
inet_ntoa	Converts an Internet address to a text string representing the addess in the standard Internet "." notation.

The **inet_makeaddr** and **inet_addr** functions provide a means for deriving an Internet address in network byte format given appropriate Internet address information. In the case of **inet_makeaddr**, you specify the network and local portions of an Internet address in host byte order. You specify **inet_addr** with a pointer to an ASCIZ NULL-terminated text string that identifies an Internet address in standard Internet "." notation. If the argument does not point to a valid Internet address, the function returns −1.

If you have an Internet address in network byte order, you can use the **inet_netof**, **inet_lnaof**, or **inet_ntoa** function to get the network portion, local portion, or string representation of an Internet address. The **inet_netof** and **inet_lnaof** functions return the network and local portions of the specified Internet address in byte host order. To get a pointer to a text string that identifies an Internet address in standard Internet "." notation, specify the network byte order Internet address in a call to **inet_ntoa**.

To convert the string representation of an Internet address to a numeric (binary) Internet address in host byte order, use the **inet_network** function. Specify a pointer to an ASCIZ NUL-terminated text string that identifies an Internet address in standard Internet "." notation. If the argument does not point to a valid Internet address, the function returns −1.

For descriptions of the Internet address manipulation routines, see the *VAXELN C Reference Manual.*

10.6 Programming Internet Communication

You program Internet communication using socket interface routines. You can use the routines to program connectionless communication, sending datagrams to specified destinations, or you can use them to program connection-oriented communication. The VAXELN Toolkit provides the following socket interface routines:

Routine	Description
accept	Accepts a connection on a socket.
bind	Binds a name to a socket.
close	Closes a socket.
connect	Initiates a connection on a socket.
listen	Sets the maximum limit of outstanding connection requests for a connection-oriented socket.
read	Reads bytes from a file or connected socket and places them in a buffer.
recv	Receives bytes from a connected socket and places them in a buffer.
recvfrom	Receives bytes for a socket from any source.
recvmsg	Receives bytes from a socket and places them in scattered buffers.
select	Polls and checks a group of sockets for I/O activity.
send	Sends bytes through a socket to its connected peer.
sendmsg	Sends gathered bytes through a socket to any other socket.
sendto	Sends bytes through a socket to any other socket.

Routine	Description
shutdown	Shuts down a socket.
socket	Creates a socket and returns the socket's descriptor.
vaxc$get_ sdc	Returns a socket device descriptor.
vaxc$socket_ control	Sets socket characteristics.
write	Writes bytes from a buffer to a file or connected socket.

Table 10–6 lists the calling sequence for programming connection-oriented and connectionless socket communication.

Table 10–6: Calling Sequence for Socket Communication

Task	Connectionless (IP and UPD/IP)	Connection-Oriented (TCP/IP)
Create a socket.	**socket**	**socket**
Bind a name to the socket.	**bind**	**bind**
Define the socket as a listener.		**listen**
Client: Send a connection request.		**connect**
Server: Accept a connection request.		**accept**
Send data.	**sendto** **sendmsg**	**write** **send** **sendto** **sendmsg**
Receive data.	**recvfrom** **recvmsg**	**read** **recv** **recvfrom** **recvmsg**
Shut down the socket.	**shutdown**	**shutdown**
Close (delete) the socket.	**close**	**close**

Sections 10.6.1 to 10.6.7 explain how to use the socket communication routines. Sections 10.6.1 and 10.6.2 explain how to create and bind names to sockets. Section 10.6.4 explains how to establish connections for TCP/IP communication. Section 10.6.5 explains how applications

can use sockets to transfer data. Sections 10.6.6 and 10.6.7 explain how to shut down and close sockets, respectively.

For descriptions of the socket communication routines, see the *VAXELN C Reference Manual*.

10.6.1 Creating Sockets

An application creates sockets by calling the **socket** function. A call to **socket** must specify an address format, type, and protocol. The address format argument defines the address format to be used in subsequent operations that use the socket. The VAXELN socket routines support Internet (AF_INET) addresses.

A socket's type and protocol affect the way the socket operates and how an application uses it. The type argument specifies whether the socket operates as a stream, datagram, or raw data transmission mechanism. Sockets of type SOCK_STREAM are for reliable, sequenced, two-way connection-based communication that can handle out-of-band data. Sockets of type SOCK_DGRAM are for connectionless communication. Sockets of type SOCK_RAW provide access to internal network interfaces, and are available only to programs authorized with a system group UIC (that is, a UIC less than or equal to %X0008FFFF or [10,177777]).

The protocol argument specifies the protocol to be used with the socket. Normally, only one protocol supports a particular socket type using a given address format. Generally, the stream, datagram, and raw socket types map to the protocols TCP, UDP, and IP, respectively. However, multiple protocols can exist for a socket type. If so, you must specify a protocol. The protocol number you need to specify depends on the communication domain in which the socket is to be used.

The following call to **socket** creates a stream socket and returns a socket descriptor to *socket_2*:

```
#include types
#include socket
#include in
   .
   .
   .
main(argc, argv)
int        argc;
char       **argv;
{
       int    socket_2;
   .
   .
   .
socket_2 = socket(AF_INET, SOCK_STREAM, 0)
   .
   .
   .
}
```

You can gain more control over how a socket operates by using the **setsockopt** function to set the following socket options:

- Let local addresses be reused
- Keep connections alive
- Do not apply routing on outgoing messages
- Linger on close operations if data is present
- Let broadcast messages be sent

For more information about setting socket options, see Section 10.7.2.

10.6.2 Binding Names to Sockets

When an application creates a socket, the socket exists in an address family's name space but has no name (direct address) assigned. To use the socket, the application must bind a name to it, using a call to the **bind** function. A call to **bind** must specify a socket descriptor returned by **socket**, a name, and the length of the name.

The name argument specifies the address of a structure that defines a name for the socket. The structure must define the name using the socket's address format. The VAXELN socket interface defines two such structures: **sockaddr** and **sockaddr_in**. The **sockaddr** structure defines names for sockets that use a general address format. Members of this structure identify the socket's address family and a data string of up to 14 bytes of direct address. The **sockaddr_in** structure defines names for sockets that use the Internet address format. This structure defines a name that includes the socket's address family (AF_INET), a port number in network byte order, an Internet address in network byte order, and an 8-byte field that contains all zeros.

The name length argument must specify the size of the name structure in bytes.

The following code binds an Internet address name *socket_2_name* to the socket *socket_2*:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int        argc;
char       **argv;
{
       int    socket_2;
static struct sockaddr_in socket_2_name;
    .
    .
    .
/*
 *  Fill in the name structure.
 */

socket_2_name.sin_family = AF_INET;
socket_2_name.sin_port = htons(atoi(argv[1]));
socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.1");

/*
 *  Bind the name to the socket.
 */

return_val = bind(socket_2, &socket_2_name, sizeof(socket_2_name));
    .
    .
    .
}
```

Once a socket has a name, an application can use the socket for either connection-oriented or connectionless communication.

10.6.3 Controlling Socket Characteristics

The VAXELN Internet software provides the routines **vaxc$get_sdc** and **vaxc$socket_control** for controlling certain socket characteristics. The **vaxc$get_sdc** routine returns the socket device descriptor associated with a specified socket descriptor. Once an application has the socket device descriptor, it can specify that descriptor in calls to **vaxc$socket_control** to do the following:

- Set the socket to a blocking or nonblocking state
- Determine whether the socket's read pointer is pointing at the out-of-band data marker

A blocking socket waits for the current operation to complete, while a nonblocking socket does not block if the requested operation takes a considerable amount of time.

Calls to the **vaxc$socket_control** must specify a socket device descriptor returned by **vaxc$get_sdc**, a request, and an argument pointer. The request argument specifies the characteristic to be set or returned. The argument pointer specifies the address of a buffer that supplies information to or receives characteristics from the routine.

To set a socket to the blocking or nonblocking state, you must specify FIONBIO as the request. If you specify FIONBIO and the specified buffer contains 0, the socket is set to the blocking state. Otherwise, it is set to the nonblocking state.

To determine whether a socket's read pointer is pointing at the out-of-band data marker in the data stream, specify SIOCATMARK as the request. When you specify this request, the routine returns the value 1 to the specified buffer if the next read is to return data after the mark.

The following example shows how you might use **vaxc$get_sdc** and **vaxc$socket_control** to create an I/O control function that is similar to the UNIX **ioctl** function.

```
/*
**  Include files
*/

#include inetdef
```

```
/*
 *  I/O control functions have the command encoded in the lower word
 *  and the size of the input and output parameters in the upper word.
 *  The high two bits of the upper word are used to encode the
 *  parameter's I/O status.  For now, we restrict parameters to at most
 *  128 bytes.
 *
 *  The IOC_VOID field of 0x20000000 is defined so that new I/O control
 *  functions can be distinguished from old I/O control functions.
 */

#ifndef _IO
#define IOCPARM_MASK   0x7f            /* Parameters are < 128 bytes  */
#define IOC_VOID       (int)0x20000000 /* No parameters               */
#define IOC_OUT        (int)0x40000000 /* Copy output parameters      */
#define IOC_IN         (int)0x80000000 /* Copy input parameters       */
#define IOC_INOUT      (int)(IOC_IN|IOC_OUT)
#define _IO(x,y)       (int)(IOC_VOID|('x'<<8)|y)
#define _IOR(x,y,t)    (int)(IOC_OUT|((sizeof(t)&IOCPARM_MASK)<<16)|('x'<<8)|y)
#define _IOW(x,y,t)    (int)(IOC_IN|((sizeof(t)&IOCPARM_MASK)<<16)|('x'<<8)|y)
#define _IOWR(x,y,t)   (int)(IOC_INOUT|((sizeof(t)&IOCPARM_MASK)<<16)|('x'<<8)|y)
#endif _IO

#define ODD(s) (s & 01)
/*-------------------------------------------------------------------------*/
int ioctl(d, request, argp)

int d, request;
char *argp;

/* Arguments:
**          d       - Specifies the socket descriptor
**          request - Specifies the characteristics; either FIONBIO or
**                    SIOCATMARK
**          argp    - Points to the buffer that specifies input to or
**                    receives output from the routine
*/
{
        int     sdd;                    /* Socket device descriptor */
        int     retval;                 /* Return value             */

        /*
        ** Get the socket device descriptor.
        ** If failure, then errno will contain error number.
        */

        sdd = vaxc$get_sdc(d);

        /*
        ** Do socket control.
        ** If failure, then errno will contain error number.
        */
        if (sdd) {
          retval = vaxc$socket_control(sdd,
                                       request,
                                       argp);
          return(retval);
        }
        else
          return (-1);                          /* Return failure */
}
```

For descriptions of the **vaxc$get_sdc** and **vaxc$socket_control**, see the *VAXELN C Reference Manual*.

10.6.4 Establishing Connections for Socket Communication

To use sockets for TCP communication, an application must establish a connection between client and server sockets. A client initiates a connection by sending a connection request to a server's socket. A server waits for connection requests, and depending on the state and characteristics of its socket, receives the requests, places the requests in a queue, or rejects the requests.

The following sections explain how to establish socket connections. Sections 10.6.4.1 and 10.6.4.3 explain how to send and accept socket connection requests. Section 10.6.4.2 explains how to associate a socket with a queue for pending connection requests.

10.6.4.1 Initiating Socket Connections

A client initiates a connection on a socket by calling the **connect** function. The function call must specify a socket descriptor returned by **socket**, the name of a remote socket, and the length of the name.

The socket descriptor can be of type SOCK_DGRAM or SOCK_STREAM. If the socket is of type SOCK_DGRAM, the call permanently specifies the peer to which data is to be sent. If the socket is of type SOCK_STREAM, the function sends a connection request to another socket.

The name argument specifies the address of a structure that names the remote socket to which the specified socket is to connect. The structure must define the name using the remote socket's address format. Section 10.6.2 provides information about the VAXELN socket address structures and binding names to sockets.

NOTE

> If an application does not bind an Internet address (name) to a socket before calling the **connect** function, the function uses the local Internet address. If the Internet address is not defined when the application calls **connect**, the function blocks until the Internet address is set.

The name length argument must specify the size of the name structure in bytes.

The following code fragment connects the socket *socket_1* to the remote socket named *socket_2_name*:

```
#include types
#include socket
#include in
   .
   .
   .
main(argc, argv)
int        argc;
char       **argv;
{
       int    socket_1;
static struct sockaddr_in socket_2_name;
   .
   .
   .
/*
 *  Fill in the name structure for the remote socket.
 */

socket_2_name.sin_family = AF_INET;
socket_2_name.sin_port = htons(atoi(argv[2]));
socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

/*
 *  Connect socket_1 to socket_2_name.
 */

return_val = connect(socket_1, &socket_2_name, sizeof(socket_2_name));
   .
   .
   .
}
```

10.6.4.2 Creating a Queue for Pending Connection Requests

Before a server can accept a connection on a socket, it must create and associate the socket with a queue that stores pending connection requests. The socket uses the queue to listen for requests. If the server is busy when a request arrives, the request is queued. If the queue is empty when the server is ready to service a request, the server waits on the queue for a new request.

To create a queue for a socket, call the **listen** function. The function call must specify a socket descriptor of type SOCK_STREAM returned by **socket** and an integer in the range 1 to 5 that specifies the maximum number of pending connections that may be queued for the socket at any given time. If a connection request arrives when the queue is full, the client receives an error.

The call to **listen** in the following example creates a connection request queue for *socket_2*. The queue entry limit is set to 5.

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       int    socket_2;
    .
    .
    .
/*
 *  Listen on socket_2 for connection requests.
 */
return_val = listen(socket_2, 5);
    .
    .
    .
}
```

10.6.4.3 Accepting Socket Connections

A server accepts a connection on a socket by calling the **accept** function. A call to **accept** must specify a socket descriptor of type SOCK_STREAM returned by **socket**, a variable that receives the address of the connecting entity, and an argument that specifies and receives the length of the connecting entity's address in bytes. The socket descriptor that you specify must be bound to a name and listening for connection requests.

The address of the connecting entity is filled in as it is known to the communication layer. The format of the structure to which the address points is determined by the communication domain. The VAXELN Internet Services support the Internet (AF_INET) domain.

The address length argument should specify the size of the structure to which the address argument points. When the function returns, the argument contains the actual length of the structure that the communication layer places in the address argument.

The **accept** function completes the first connection on the socket's connection pending queue, creates a new socket with the same properties as the specified socket, and allocates and returns a new descriptor for the socket. If no connections are pending and the socket is not marked as nonblocking, the function blocks the calling process until a connection request is present. If the socket is marked as nonblocking and no connections are pending, the function returns an error. The original socket continues to listen for other connection requests.

The following code accepts a connection from *socket_2* and places the accepted connection on *socket_3*:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       int    socket_2, socket_3;
static struct sockaddr_in socket_2_name;
       int    socket_2_namelen;
    .
    .
    .
/*
 *  Accept connection request from socket_2.
 *  Accepted connection will be on socket_3.
 */

socket_2_namelen = sizeof(socket_2_name);
socket_3 = accept(socket_2, &socket_2_name, &socket_2_namelen);
    .
    .
    .
}
```

10.6.5 Transferring Data

A variety of socket communication routines are available for transferring data between sockets. Sections 10.6.5.1 and 10.6.5.2 explain how to use the routines to send and receive data, respectively. Section 10.6.5.3 explains how to poll sockets for I/O activity while programming data transfers between sockets.

10.6.5.1 Sending Data to Sockets

Internet applications can send data from sockets by calling the **write**, **send**, **sendto**, or **sendmsg** function. You can use any of these functions to send data in connection-oriented communication. You must use **sendto** or **sendmsg** to send data in a connectionless environment.

The **write** function writes a buffer of data to a connected socket or file. You specify **write** with a destination socket or file descriptor, the address of contiguous storage from which the output data is to be taken, and the maximum number of bytes to be written.

The **send** function provides an alternative method of sending data between connected sockets. A call to **send** must specify a socket descriptor, the address of the buffer containing the data to be sent, the length (in bytes) of the data being sent, and an out-of-band character flag. In the case of **send**, the socket descriptor specifies a source socket — the socket from which data is sent — that is connected to another socket. The function sends bytes of data through the specified socket to its connected peer and returns an integer indicating the number of bytes of data that were sent.

The out-of-band character flag that you specify with **send** can be 0 or MSG_OOB. If you specify MSG_OOB, data can be received before other pending data on the receiving socket if the receiver also specifies MSG_OOB.

The following code uses a call to **send** to send a message to *socket_2*:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       int    socket_1;
static struct sockaddr_in socket_2_name;
       int    socket_2_namelen;
    .
    .
    .
/*
 *  Fill in the name structure for the remote socket.
 */

socket_2_name.sin_family = AF_INET;
socket_2_name.sin_port = htons(atoi(argv[2]));
socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

/*
 *  Connect socket_1 to socket_2_name.
 */

socket_2_namelen = sizeof(socket_2_name);
return_val = connect(socket_1, &socket_2_name, &socket_2_namelen);

/*
 *  Send message to socket_2.
 */
flag = 0;
return_val = send(socket_1, message, sizeof(message), flag);
    .
    .
    .
}
```

If your application uses connectionless socket communication, you can send data by calling the **sendto** or **sendmsg** function. These functions send data to any other socket. The **sendto** function sends bytes of data through a socket to any other socket. The **sendmsg** function sends gathered bytes of data through a socket to any other socket.

Like calls to **send**, calls to **sendto** must specify a socket descriptor, the address of the buffer containing the data to be sent, the length (in bytes) of the data being sent, and an out-of-band character flag. In addition, you must supply the destination socket by specifying the address of the destination socket's address structure and the length of that structure.

The call to **sendto** in the following code fragment sends data between unconnected sockets:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       int    sendlength, tolength;
static char   sendbuf[] = "Hi.";
       int    flag;
       int    return_val;
static struct sockaddr_in socket_2_name;
    .
    .
    .
/*
 *  Fill in the address structure for socket_2.
 */

socket_2_name.sin_family = AF_INET;
socket_2_name.sin_port = htons(atoi(argv[2]));
socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

/*
 *  Initialize send block.
 */
sendlength = sizeof(sendbuf);
tolength = sizeof(socket_2_name);
flag =0;

/*
 *  Send message from socket_1 to socket_2.
*/

return_val = sendto(socket_1, sendbuf, sendlength, flag, &socket_2_name,
                    tolength);
    .
    .
    .
}
```

When you use the **sendmsg** function, you must specify a socket descriptor created by **socket**, a message argument, and an out-of-band character flag.

The message argument specifies the address of a message header structure of type **msghdr** that contains the message to be sent. The message header structure lets the **sendmsg** function gather data from several user transmit buffers before sending a message. Members of the **msghdr** structure identify the following:

- The address of the destination socket if the source socket is not connected.
- The length of the message name field.
- An array of I/O buffer pointers of the **iovec** structure form. Each of the buffer pointers specifies the address of a buffer and that buffer's length.
- The number of buffers in the message array.
- The address of a buffer containing access rights sent with the message.
- The length of the access rights buffer.

The **sendmsg** function sends the data in the **msg_iovec** field of the **msghdr** structure to the socket whose address is specified in the **msg_name** field of **msghdr**. The receiving socket can then receive the data. If the array specifies multiple buffers, **sendmsg** gathers the data from all specified buffers before sending the message.

If blocking is enabled for a socket, **send**, **sendto**, and **sendmsg** will block if the receiving end of a socket connection does not have enough space to buffer the data being sent. However, if the sending socket is defined as nonblocking, an error results and the send operation fails.

The send operation will fail also if the sending socket is of type SOCK_ DGRAM and the message is too large to be sent in one piece.

10.6.5.2 Receiving Data from Sockets

Internet applications can receive data from sockets by calling the **read**, **recv**, **recvfrom**, or **recvmsg** function. You can use any of these functions for receiving data in connection-oriented communication. You must use **recvfrom** or **recvmsg** to receive data in a connectionless environment.

The **read** function reads data from a connected socket or file and places the data in a buffer. You specify **read** with the descriptor of a socket or file opened for reading, address of contiguous storage in which the input data is to be placed, and the maximum number of bytes to be read. The function returns the number of bytes actually read and placed in the buffer.

The **recv** function provides an alternative means of receiving data between connected sockets. A call to **recv** must specify a socket descriptor, the address of the buffer into which received data is to be placed, the length (in bytes) of the specified buffer, and a flags argument. In the case of **read**, the socket descriptor specifies a destination socket — the socket from which data is received — that is connected to another socket. The function receives bytes of data through the specified socket from its connected peer and returns an integer indicating the number of bytes of data received and placed in the buffer.

The flags argument is a bit mask that specifies whether the function should receive out-of-band characters and be allowed to peek at data before it is read. The out-of-band character flag can be 0 or MSG_OOB. If you specify MSG_OOB, available out-of-band data will be read before any other available data. The peek flag can be 0 or MSG_PEEK.

The following code uses a call to **recv** to receive a message from *socket_ 2*:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int        argc;
char       **argv;
{
       int    socket_2, socket_3;
static char   message[BUFSIZ];
static struct sockaddr_in socket_2_name;
       int    socket_2_namelen;
       int    flag;

    .
    .
    .
/*
 *  Accept connection request from socket_2.
 *  Accepted connection will be on socket_3.
 */

socket_2_namelen = sizeof(socket_2_name);
socket_3 = accept(socket_2, &socket_2_name, &socket_2_namelen);

/*
 *  Receive message from socket_1.
 */

flag = 0;

retval = recv(socket_3, message, sizeof(message), flag);
    .
    .
    .
}
```

If your application uses connectionless socket communication, you can receive data by calling the **recvfrom** or **recvmsg** function. These functions receive data from another source. The **recvfrom** function receives bytes of data on a socket from any source. The **recvmsg** function receives bytes of data on a socket and places them in scattered buffers.

Like calls to **recv**, calls to **recvfrom** must specify a socket descriptor, the address of the buffer into which received data is to be placed, the size of the buffer, and a flags argument. In addition, you must supply a source and source length. The source argument can be zero or nonzero. If nonzero, the argument points to the buffer into which **recvfrom** is to place the address structure of the socket from which data is received. If you specify zero, the address is not returned. The source length argument points to an integer that indicates the buffer's size. When the function returns, the size is modified such that it contains the actual length of the socket address returned.

The call to **recvfrom** in the following code fragment receives data from an unconnected socket:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int        argc;
char       **argv;
{
       int    buflength, fromlength;
static char   recvbuf[] = "Hi.";
       int    flag;
       int    return_val;
static struct sockaddr_in socket_2_name;
    .
    .
    .
/*
 *  Receive data from socket_1 on socket_2.
 */

buflength = sizeof(recvbuf);
fromlength = sizeof(socket_1_name);
flag = 0;

return_val = recvfrom(socket_2, recvbuf, buflen, flag, &socket_1_name,
                         &fromlength);
    .
    .
    .
}
```

When you use the **recvmsg** function, you must specify a socket descriptor created by **socket**, a message argument, and a flags argument.

The message argument specifies the address of a message header structure of type **msghdr** into which the received message is to be placed. The message header structure lets the **recvmsg** function scatter data to several user transmit buffers after a message is received. Members of the **msghdr** structure identify the following:

- The address of the destination socket if the source socket is not connected.
- The length of the message name field.

- An array of I/O buffer pointers of the **iovec** structure form. Each of the buffer pointers specifies the address of a buffer and that buffer's length.
- The number of buffers in the message array.
- The address of a buffer containing access rights sent with the message.
- The length of the access rights buffer.

The **recvmsg** function scatters a message into several user buffers if such buffers are available. The data is scattered into the message array buffers as specified by the **iovec** structure.

When **recvmsg** receives a message, the message is split among buffers by filling the first buffer in the list, then the second, and so on, until all the buffers are full or no more data is available.

If blocking is enabled for a socket, **recv** and **recvto** block and wait for data to arrive if no data is available at the time of the function call. However, if the sending socket is defined as nonblocking, an error results and the receive operation fails.

10.6.5.3 Polling Sockets for I/O Activity

While programming data transfers to and from sockets, you may want to poll the sockets for I/O activity. By polling the sockets, you can check which sockets are ready to receive or send data, or which sockets have a pending exception.

To poll sockets for I/O activity, use the **select** function. This function determines the I/O status of the sockets specified in various mask arguments. The function returns when a socket is ready to receive or send data, or when a timeout value expires.

A call to **select** must specify the highest numbered socket descriptor for which the function must search; pointers to arrays of bits that indicate which sockets should be checked for read or write readiness or for exceptions; and a timeout value. The first argument improves efficiency by specifying the highest numbered bit +1 to be checked. A descriptor is represented by 1<<*s* (1 shifted to the left *s* times). If you are not sure what the highest numbered descriptor is, you can safely specify a number less than 64.

The read, write, and exception fields arguments are pointers to arrays of bits, organized as integers (each integer describes 32 descriptors) that you can examine. If bit *n* of a bit array is set, the function checks to see if socket descriptor *n* is ready to be read from, is ready to be written to, or has any pending exceptions. All bits in the bit masks must correspond to socket descriptors.

On return, the bit array to which each of the fields arguments points contains a bit mask of sockets that are ready for reading, are ready for writing, or have exceptions pending. Only bits that are set on entry to **select** can be set on exit.

The timeout argument is a **timeval** structure that specifies the maximum interval to wait for a selection to be completed. The **timeval** structure consists of members that specify the number of seconds and number of microseconds to wait. If one of the sockets specified in the bit masks is ready for I/O, the function returns before the timeout expires.

The following code fragment selects a socket to receive a message:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       unsigned long rmask, wmask, emask;
       int    socket_2;
       int    return_val;
static struct sockaddr_in socket_2_name;
struct timeval timeout;
    .
    .
    .
/*
 *  Select socket to receive message.
 */

emask = wmask = 0;
rmask = (1<<socket_2);       /* Set read mask */
timeout.tv_sec = 30;
timeout.tv_usec = 0;

return_val = select(32, &rmask, &wmask, &emask, &timeout);
switch(return_val)
{
  case -1:  perror("select error");
            break;

  case 0:   printf("Select timed out with status 0.\n");
            break;

  default:  if ((rmask& (1<<socket_2)) == 0)
              printf("Select not reading on socket_2.\n");
            break;
} /* switch */
   .
   .
   .
}
```

If a call to **select** blocks a process while waiting for input from a socket and the sending process closes the socket, **select** notes this as an event and unblocks the process. The descriptors are modified on return if **select** returns because of a timeout.

10.6.6 Shutting Down Sockets

When an application no longer needs a socket, it should use the **shutdown** function to shut down the socket. An application can shut down a socket completely or shut down the socket's ability to receive or send data. You might use this function to program a more controlled shutdown. The **shutdown** function is also useful for setting up one-way (half-duplex) communication rather than normal two-way (full-duplex) communication.

A call to **shutdown** must specify a socket descriptor and an integer in the range 0 to 2 that indicates how the socket is to be shut down. If you specify 0, the socket can no longer receive data. If you specify 1, the socket can no longer send data. The value 2 prevents the socket from receiving or sending data.

In the following example, **shutdown** shuts down *socket_1* completely:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       int    socket_1;
       int    return_val;
    .
    .
    .
/*
 *  Shut down socket_1.
 */

return_val = select(socket_1, 2);
}
```

10.6.7 Closing Sockets

When a socket is no longer being used, the application should close it. To close a socket, call the **close** function. If the socket is a connected socket, the function breaks the connection and then deletes the socket's descriptor from the appropriate reference table. Otherwise, the function just deletes the descriptor. If the close operation is the last reference to the socket, the socket is deactivated.

The following code fragment shuts down and closes the socket *socket_1*:

```
#include types
#include socket
#include in
    .
    .
    .
main(argc, argv)
int         argc;
char        **argv;
{
       int    socket_1;
       int    return_val;
    .
    .
    .
/*
 *  Shut down and close socket_1.
 */
return_val = shutdown(socket_1, 2);
return_val = close(socket_1);
}
```

10.6.8 Programming Socket Communication for a UDP Application

This section shows an example of how you might use the socket communication routines to program a UDP application. The example consists of a UDP server (see Example 10–1) and a UDP client (see Example 10–2). The server creates a socket of type SOCK_DGRAM (UDP), binds it, and selects to receive a message on the socket. The server program expects the number of the port where it is waiting for requests. The client creates a socket of type SOCK_DGRAM (UDP), binds it, and sends a message to a specified destination address. The client program expects the name of a remote host and the port number where the remote host is waiting.

To run the sample application you must do the following:

- Pass 7 as the fourth program argument for both the server and client programs. The first argument (program name) is not supported by the VAXELN Toolkit, and the **stdin**, **stdout**, and **stderr** arguments are not used.
- Set the priority of the server to a higher priority than that of the client if the server and client are to run on the same node.
- The Internet address in the example code must match the Internet address that you specify for the **Internet address** entry on the System Builder's Internet Network Description menu.

Example 10–1: Sample UDP Server

```
/*
 *  Include Files
*/

#include  $vaxelnc
#include  errno
#include  types
#include  stdio
#include  socket
#include  in
#include  inet

main(argc, argv)
int         argc;
char        **argv;
{
        unsigned long read_mask, write_mask, exception_mask;
        int     socket_2;                   /* Socket 2 descriptor.        */
        int     buflen, fromlen;
        char    recvbuf[BUFSIZ];
static  struct  sockaddr_in socket_1_name;  /* Address structure for Socket 1. */
static  struct  sockaddr_in socket_2_name;  /* Address structure for Socket 2. */
        int     namelength;
        int     retval;
        int     flag;
        struct  timeval timeout;

        /*
         *  Check input parameters.
         */

        if (argc != 2)
                {
                printf("Usage: server port number.\n");
                exit();
                }
        /*
         *  Create a datagram socket (SOCK_DGRAM) that is to use Internet
         *  addresses.  Return the socket descriptor to socket_2.
         */

        if ((socket_2 = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
          {
            perror( "socket error");
            exit();
          }
```

Example 10–1 Cont'd on next page

Example 10–1 (Cont.): Sample UDP Server

```
        /*
         *  Build the address structure for socket_2.
         */

        socket_2_name.sin_family = AF_INET;
        socket_2_name.sin_port = htons(atoi(argv[1]));
        socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

        /*
         *  Bind socket_2 to the name structure socket_2_name.
         */

        retval = bind(socket_2, &socket_2_name, sizeof(socket_2_name));
        if (retval)
          {
            perror("bind error");
            cleanup(socket_2);
          }

        /*
         *  Set the read mask and poll socket_2 for read requests.  Use
         *  a timeout value of 30 seconds.
         */

        exception_mask = write_mask = 0;
        read_mask = (1<<socket_2);  /* Set read mask */
        timeout.tv_sec = 30;
        timeout.tv_usec = 0;
```

Example 10–1 Cont'd on next page

Example 10–1 (Cont.): Sample UDP Server

```
        retval = select(32, &read_mask, &write_mask,
                        &exception_mask, &timeout);
        switch(retval)
        {
          case -1:
                {
                  perror("select error");
                  cleanup(socket_2);
                }
                break;
          case 0:
                {
                  printf("Select timed out with status 0.\n");
                  cleanup(socket_2);
                }
                break;
          default:
                if ((read_mask & (1<<socket_2)) == 0)
                  {
                    printf("Select not reading on socket_2.\n");
                    cleanup(socket_2);
                  }
        } /*switch*/

        /*
         *  Initialize the receive buffer.
         */

        buflen = sizeof(recvbuf);
        fromlen = sizeof(socket_1_name);
        flag = 0;   /* Flag can be MSG_OOB or MSG_PEEK */

        /*
         *  Receive data from a socket named socket_1_name, using
         *  socket_2, and place the data in the buffer recvbuf.
         */

        retval = recvfrom(socket_2, recvbuf, buflen, flag,
                          &socket_1_name, &fromlen);
        if (retval == -1)
          perror("recvfrom error");
        else
          printf(" %s\n", recvbuf);
```

Example 10–1 Cont'd on next page

Example 10–1 (Cont.): Sample UDP Server

```
        /*
         *  Call cleanup to shut down and close socket_2.
         */

        cleanup(socket_2);

 } /* end main */

/*--------------------------------------------------------------*/

cleanup(socket)
int  socket;

{
        int   retval;

        /*
         *  Shut down socket completely.
         */

        retval = shutdown(socket,2);
        if (retval == -1)
          perror ("udp_server shutdown error");

        /*
         *  Close the socket.
         */
        retval = close(socket);
        if (retval)
          perror("close error");

        exit();

} /* end cleanup */
```

Example 10–2:   Sample UDP Client

```
/*
 *  Include Files
 */

#include  $vaxelnc
#include  errno
#include  types
#include  stdio
#include  socket
#include  in
#include  inet

main(argc, argv)
int          argc;
char         **argv;
{
        int      socket_1;                  /* Socket descriptor for Socket 1 */
        int      sendlen, tolen;
static  char     sendbuf[] = "Have a nice day.";
static struct    sockaddr_in socket_2_name; /* Address structure for Socket 2 */
        int      namelength;
        int      flag;
        int      retval;

        /*
         *  Check input parameters.
         */
        if (argc != 2)
                {
                printf("Usage: port number.\n");
                exit();
                }

        /*
         *  Create a datagram socket (SOCK_DGRAM) that is to use Internet
         *  addresses.  Return the socket descriptor to socket_1.
         */

        if ((socket_1 = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
          {
            perror("socket error");
            exit();
          }

        /*
         *  Build an address structure for socket_2 for receiving the
         *  message.
         */
```

Example 10–2 Cont'd on next page

Example 10–2 (Cont.): Sample UDP Client

```
        socket_2_name.sin_family = AF_INET;
        socket_2_name.sin_port = htons(atoi(argv[1]));
        socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

        /*
         *  Initialize the send buffer.
         */

        sendlen = sizeof(sendbuf);
        tolen = sizeof(socket_2_name);
        flag = 0; /* Flag may be MSG_OOB */

        /*
         *  Send data from the buffer sendbuf using socket_1 to
         *  a socket named socket_2_name.
         */

        retval = sendto(socket_1, sendbuf, sendlen, flag,
                        &socket_2_name, tolen);
        if (retval == -1)
          {
            perror("sendto error");
            cleanup(socket_1);
          }

        /*
         *  Call cleanup to shut down and close socket_1.
         */

        cleanup(socket_1);

 } /* end main */

/*-----------------------------------------------------------*/

cleanup(socket)
int         socket;

{
        int         retval;

        /*
         *  Shut down socket completely.
         */

        retval = shutdown(socket, 2);
        if (retval == -1)
          perror("udp_client shutdown error");
```

Example 10–2 Cont'd on next page

Example 10–2 (Cont.): Sample UDP Client

```
        /*
         *  Close the socket.
         */

        retval = close(socket);
        if (retval)
          perror("close error");

        exit();

} /* end cleanup */
```

10.6.9 Programming Socket Communication for a TCP/IP Application

This section shows an example of how you might use the socket communication routines to program a TCP/IP application. The example consists of a TCP/IP server (see Example 10–3) and a TCP/IP client (see Example 10–4). The server creates a socket of type SOCK_STREAM (TCP), binds it, listens on it, receives a message, and closes it. The server program expects the number of the port where it is listening. The client creates a socket of type SOCK_STREAM (TCP), initiates a connection to the remote host, sends a message to the remote host, and closes the connection. The client program expects the name of the remote host port where the remote host (server) is listening.

To run the sample application you must do the following:

- Pass 7 as the fourth program argument for both the server and client programs. The first argument (program name) is not supported by the VAXELN Toolkit, and the **stdin**, **stdout**, and **stderr** arguments are not used.
- Set the priority of the server to a higher priority than that of the client if the server and client are to run on the same node.
- The Internet address in the example code must match the Internet address that you specify for the **Internet address** entry on the System Builder's Internet Network Description menu.

Example 10–3: Sample TCP/IP Server

```
/*
*  Include Files
*/
#include  $vaxelnc
#include  errno
#include  types
#include  stdio
#include  socket
#include  in
#include  inet

main(argc,argv)
int         argc;
char        **argv;
{
        int     socket_2, socket_3;          /* Socket descriptors for      */
                                             /* Socket 2 and Socket 3.      */
static  char    message[BUFSIZ];
static struct   sockaddr_in socket_2_name;    /* Address structure for socket_2 */
static struct   sockaddr_in retsocket_2_name; /* Address structure for socket_2 */
        int     flag;
        int     retval;
        int     namelength;

        /*
         *  Check input parameters.
         */

        if (argc != 2)
          {
            printf("Usage: server port number.\n");
            exit();
          }

        /*
         *  Create a stream socket (SOCK_STREAM) that is to use Internet
         *  addresses.  Return the socket descriptor to socket_2.
         */

        if ((socket_2 = socket(AF_INET, SOCK_STREAM, 0)) == -1)
          {
            perror("socket error");
            exit();
          }
```

Example 10–3 Cont'd on next page

Example 10–3 (Cont.): Sample TCP/IP Server

```
/*
 *  Build an address structure for socket_2.
 */

socket_2_name.sin_family = AF_INET;
socket_2_name.sin_port = htons(atoi(argv[1]));
socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

/*
 *  Bind socket_2 to the name structure socket_2_name.
 */

retval = bind(socket_2, &socket_2_name, sizeof(socket_2_name));
if (retval)
  {
    perror("bind error");
    cleanup(1, socket_2, 0);
  }

/*
 *  Create and associate socket_2 with a queue for pending connection
 *  requests.  The socket uses the queue to listen for requests.
 */

retval = listen(socket_2, 5);
if (retval)
  {
    perror("listen error");
    cleanup(1, socket_2, 0);
  }

/*
 *  Accept a connection request from socket_2.  Place accepted
 *  requests on socket_3.
 */

namelength = sizeof(socket_2_name);
socket_3 = accept(socket_2, &socket_2_name, &namelength);
if (socket_3 == -1)
  {
    perror ("accept error");
    cleanup(2, socket_2, socket_3);
  }
```

Example 10–3 Cont'd on next page

Example 10–3 (Cont.): Sample TCP/IP Server

```
        /*
         *  Receive data from socket_1, using socket_3, and place the
         *  data in the buffer message.
         */

        flag = 0; /* Can be 0, MSG_OOB, or MSG_PEEK. */

        retval = recv(socket_3, message, sizeof(message), flag);
        if (retval == -1)
          {
            perror("receive error");
            cleanup(2, socket_2, socket_3);
          }
        else
          printf (" %s\n", message);

        /*
         *  Call cleanup to shut down and close the sockets.
         */

        cleanup(2, socket_2, socket_3);

 } /* end main */

/*-----------------------------------------------------------*/

cleanup(how_many, socket_1, socket_2)
int         how_many;
int         socket_1, socket_2;

{
        int         retval;

        /*
         *  Shut down and close socket_1 completely.
         */

        retval = shutdown(socket_1, 2);
        if (retval == -1)
          perror("tcp_server shutdown error, socket_1");

        retval = close(socket_1);
        if (retval)
          perror("close error");
```

Example 10–3 Cont'd on next page

Example 10–3 (Cont.): Sample TCP/IP Server

```
        /*
         *  If given, shut down and close socket_2.
         */
        if (how_many == 2)
          {
            retval = shutdown(socket_2, 2);
            if (retval == -1)
              perror("tcp_server shutdown error, socket_2");

            retval = close(socket_2);
            if (retval)
              perror("close error");
          }

        exit();

} /* end cleanup*/
```

Example 10–4: Sample TCP/IP Client

```
/*
*  Include Files
*/

#include  $vaxelnc
#include  errno
#include  types
#include  stdio
#include  socket
#include  in
#include  inet


main(argc,argv)
int         argc;
char        **argv;
{
        int     socket_1;                  /* Socket descriptor for Socket 1 */
static  char    message[] = "Have a nice day.";
static  struct  sockaddr_in socket_2_name; /* Address structure for Socket 2 */
        int     flag;
        int     retval;
        int     shut = FALSE;              /* Flag to cleanup                */
```

Example 10–4 Cont'd on next page

Example 10–4 (Cont.): Sample TCP/IP Client

```
/*
 *  Check input parameters.
 */
if (argc != 2 )
  {
    printf("Usage: port number.\n");
    exit();
  }

/*
 *  Create a stream socket (SOCK_STREAM) that is to use Internet
 *  addresses.  Return the socket descriptor to socket_1.
 */

if ((socket_1 = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  {
    perror("socket error");
    exit();
  }

/*
 *  Build an address structure for socket_2.
 */

socket_2_name.sin_family = AF_INET;
socket_2_name.sin_port = htons(atoi(argv[1]));
socket_2_name.sin_addr.S_un.S_addr = inet_addr("5.0.0.5");

/*
 *  Connect socket_1 to the remote socket named socket_2_name.
 */

retval = connect(socket_1, &socket_2_name, sizeof(socket_2_name));
if (retval)
  {
    perror("connect error");
    cleanup(shut, socket_1);
  }
```

Example 10–4 Cont'd on next page

Example 10–4 (Cont.): Sample TCP/IP Client

```
        /*
         *  Send data from the message buffer, using socket_1, to
         *  the connected socket.
         */

        flag = 0;  /* Can be 0 or MSG_OOB. */
        retval = send(socket_1, message, sizeof(message), flag);
        if (retval < 0)
          {
            perror ("send error");
            shut = TRUE;
          }

        /*
         *  Call cleanup to shut down and close the socket.
         */

        cleanup(shut, socket_1);

 } /* end main */

/*-------------------------------------------------------------*/

cleanup(shut, socket)
int         shut;
int         socket;

{
        int         retval;

        /*
         *  Shut down socket completely if it was connected.
         */

        if (shut)
          {
            retval = shutdown(socket, 2);
            if (retval == -1)
              perror ("tcp_client shutdown error");
          }
```

Example 10–4 Cont'd on next page

Example 10–4 (Cont.): Sample TCP/IP Client

```
        /*
         *  Close the socket.
         */

        retval = close(socket);
        if (retval)
          perror("close error");

        exit();

} /* end main */
```

10.7 Retrieving and Setting Socket Characteristics

The VAXELN Toolkit also provides routines for retrieving and setting socket characteristics. These routines include the following:

Routine	Description
getpeername	Returns the name of a socket's connected peer.
getsockname	Returns the name associated with a socket.
getsockopt	Returns the options set for a socket.
setsockopt	Sets options for a socket.

Sections 10.7.1 and 10.7.3 explain how to retrieve socket names and options. Section 10.7.2 explains how to set socket options.

10.7.1 Retrieving Socket Names

An application can retrieve a socket name by calling the **getsockname** or **getpeername** routine. The **getsockname** routine returns the name associated with a socket. The **getpeername** routine returns the name of a socket's connected peer.

You must specify these routines with a socket descriptor that was previously created with **socket**, a pointer to a buffer in which the name is to be returned, and the size of the name buffer. The socket descriptor that you specify in a call to **getsockname** must be bound to a name.

The routines return the socket name and update the name length argument with the name's actual size.

10.7.2 Setting Socket Characteristics

To set options on a socket, an application must call the **setsockopt** routine. A call to **setsockopt** must specify a socket descriptor, the protocol level for which the options are to be modified, the options to be set, the address of a buffer that contains option parameters, and the size of the option parameter buffer.

Options can exist at multiple protocol levels. However, options are always present at the uppermost socket level. To set options at the socket level, you specify the level SOL_SOCKET. To set options at any other level, specify the number of the protocol that controls the option. For example, to specify that an option be interpreted by the TCP protocol, specify the TCP protocol number (IPPROTO_TCP). See the module **in.h** for a list of the protocol values.

The interpretation of the options you specify is based on the protocol level. Table 10–7 lists the options that are available at the socket level:

Table 10–7: Socket-Level Socket Options

Option	Description
SO_BROADCAST	Lets the socket broadcast messages.
SO_DONTROUTE	Specifies that messages sent through the socket are to bypass the routing facilities. Messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_KEEPALIVE	Lets a connected socket transmit messages periodically. If a connected peer fails to respond to the messages, the connection is considered broken and the processes using the socket receive an error.
SO_REUSEADDR	Specifies that reused local addresses can be supplied in calls to **bind**.
SO_LINGER	Delays the deletion of transmitted data when a socket is closed until the data is transmitted or the device times out (approximately eight minutes).

You must specify all socket-level options except SO_LINGER with an integer parameter. Specify a nonzero value if the option is to be enabled. Specify zero to disable the option.

When you use the SO_LINGER option, you must specify the address of a **linger** structure that indicates the state of the option (on or off) and the linger interval. The linger interval indicates the number of seconds to linger. If the linger interval is zero, the value specified for the linger time when the system was built is used. The **linger** structure is defined as follows:

```
struct  linger {
        int     l_onoff;        /* option on/off */
        int     l_linger;       /* linger time   */
};
```

If the value of *l_onoff* is nonzero, the system does not delete the socket until the socket is able to transmit the data or until the socket times out. If the value of *l_onoff* is zero, the system processes a close operation as quickly as possible.

10.7.3 Retrieving Socket Options

An application can check which options are set for a socket by calling the **getsockopt** routine. A call to **getsockopt** must specify a socket descriptor, the protocol level for which the options are to be returned, the option to be returned, the address of a buffer into which the option value is to be placed, and the size of the option value buffer.

To retrieve options at the socket level, specify the level SOL_SOCKET. To retrieve options at any other level, specify the number of the protocol that controls the option. For example, to specify that an option or the TCP protocol be returned, specify the TCP protocol number (IPPROTO_ TCP). See the module **in.h** for a list of the protocol values.

The interpretation of the options you specify is based on the protocol level. See Table 10–7 for a list of the socket level options.

Chapter 11

LAT Host Services

The VAXELN Toolkit includes local area transport (LAT) host services that VAXELN systems can use to communicate with devices attached to terminal servers, such as the DECserver 500. This chapter provides an overview of the LAT host services (see Section 11.1) and explains how to use the services to do the following:

- Establish circuits for LAT communication, Section 11.2
- Manage VAXELN service nodes, Section 11.3
- Set up a dedicated service environment, Section 11.4
- Set up an application device environment, Section 11.5
- Retrieve and set terminal characteristics, Section 11.6

11.1 LAT Host Services Overview

LAT is a communications protocol that lets system nodes running LAT host services communicate with dedicated terminal server nodes running LAT server services. The collection of system nodes and terminal server nodes in a local area network (LAN) constitutes a LAT network.

The VAXELN LAT host services support the following:

- Terminal server communication
- Terminal I/O
- A control interface that LAT application programs can use to manage and monitor the LAT environment on a VAXELN system

- An interactive utility you can use to manage and monitor the LAT environment on a VAXELN system

A VAXELN system that includes the LAT host services is a *VAXELN service node.* A service node can offer services to or request access to services offered by a terminal server. By default, a service node offers VAXELN Command Language Utility (ECL) as a service. You can access that service from an interactive terminal attached to a terminal server.

The LAT host services let application programs:

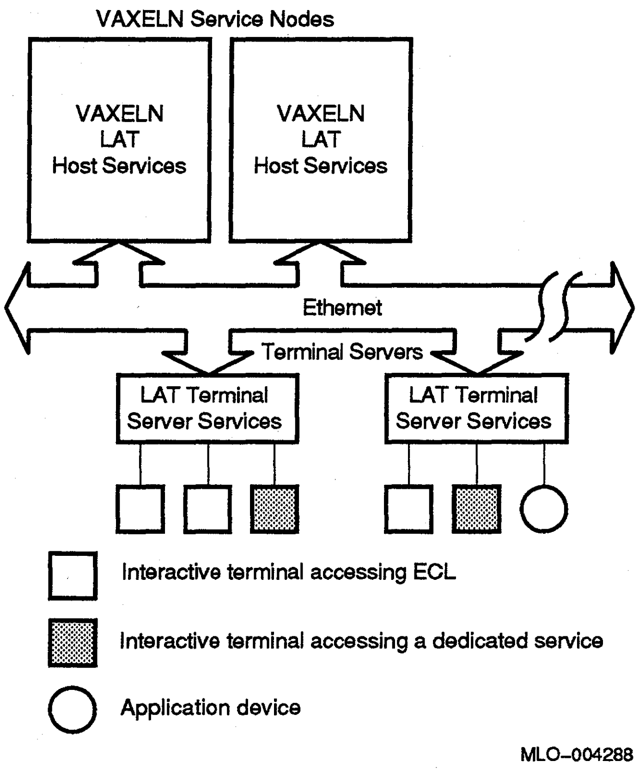
- Manage and monitor a VAXELN service node's characteristics and activities by calling VAXELN LAT utility procedures
- Set up dedicated service environments
- Set up application device environments

You can initiate communication between a service node and terminal server from an interactive terminal attached to the terminal server or from an application program running on the service node. From an interactive terminal, you establish a session with a service offered by the service node. The service can be ECL or a user-created dedicated service that is built into your VAXELN system and that executes as a job.

An application program running on a service node can establish a session with a remote application device or service attached to a terminal server. An application device offers a service to VAXELN service nodes in a LAT network. For example, a printer would offer printing services; a terminal device might offer display services.

Figure 11–1 shows a sample VAXELN LAT configuration.

Figure 11–1: Sample VAXELN LAT Configuration


VAXELN Service Nodes
VAXELN LAT Host Services
VAXELN LAT Host Services
Ethernet
Terminal Servers
LAT Terminal Server Services
LAT Terminal Server Services
Interactive terminal accessing ECL
Interactive terminal accessing a dedicated service
Application device
MLO–004288

To include the LAT host services, you build the VAXELN LAT driver (LTDRIVER) into your VAXELN system by selecting *ACTIVE* or *INACTIVE* for the **LAT host services** option on the System Builder's Network Node Characteristics Menu. If you specify *ACTIVE*, the driver's LAT protocol becomes active when your system starts executing. When the LAT protocol is active, the driver periodically multicasts a message to the terminal servers in the LAN, advertising the services that it offers. If a terminal server user tries to connect to one of the services, the service node accepts the connection request.

When the LAT protocol is inactive, the LAT driver does not multicast advertising messages to or accept connection requests from terminal servers. However, you can activate the protocol at run time with a utility command or a runtime procedure call. By using an initial inactive state, an application program can set up a LAT service node environment before the driver establishes connections with terminal servers. For example, you can set the service node's characteristics, or you can create ports for establishing connections with dedicated services or application devices.

A VAXELN application program can manage and monitor a LAT service node environment by calling LAT utility procedures. For descriptions of the utility procedures, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

The LAT driver also supports a LAT Control Program (LATCP) Utility that lets you manage and monitor LAT service node characteristics and activities interactively by entering LATCP commands. For more information about the LATCP Utility, see the *VAXELN Development Utilities Guide*.

The rest of this chapter explains how to do the following:

- Establish circuits for LAT communication, Section 11.2
- Manage VAXELN service nodes, Section 11.3
- Set up a dedicated service environment, Section 11.4
- Set up an application device environment, Section 11.5
- Retrieve and set terminal characteristics, Section 11.6

11.2 Establishing Circuits for LAT Communication

The LAT driver relies on VAXELN virtual circuits for communicating with application programs. Therefore, an application program must establish the appropriate circuit connections before it can call the VAXELN LAT utility procedures. Sections 11.2.1, 11.2.2, and 11.2.3 explain how to establish these connections.

11.2.1 Connecting to a LAT Control Port

LAT host service utility procedures manage the LAT environment on a VAXELN service node. Before an application program can call these procedures, it must create a VAXELN message port and connect that port in a circuit to a LAT control port.

When the LAT driver starts executing, it creates two control ports and two corresponding port names: the local port name $LAT_CONTROL and a universal PORT name of the form *node_name*$LAT_CONTROL. For example, if the LAT driver starts executing on the service node RTNODE, the driver names the control ports $LAT_CONTROL and RTNODE$LAT_CONTROL. The $LAT_CONTROL port makes the host service utility procedures available to application programs running on the local node; the *node_name*$LAT_CONTROL port makes the procedures available to programs running on remote nodes.

The following example shows how you might connect to the local LAT control port:

```
MODULE create_a_lat_port;

INCLUDE $LAT_UTILITY;

PROGRAM create_lat_port;

VAR
  lat_ctrl_port : PORT;
    .
    .
    .
BEGIN

  { Create a VAXELN message port. }

  CREATE_PORT(lat_ctrl_port);

  { Connect that message port in a circuit to the local LAT control
    port. }

  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');

  { Now call a LAT host service utility procedure. }

  ELN$LAT_CREATE_PORT(CIRCUIT := lat_ctrl_port,
                      PORT_NAME := 'LTA0',
                      PORT_TYPE := LAT$APPLICATION);
    .
    .
    .
END.
END;
```

Once a program connects to a control port, the program specifies the port on its end of the connection in calls to the LAT host service utility procedures. In the preceding example, the port *lat_ctrl_port* connects to the local control port $LAT_CONTROL. Thus, *lat_ctrl_port* can be used in the subsequent call to ELN$LAT_CREATE_PORT.

You must specify a LAT control port in calls to the following LAT host service utility procedures.

Routine	**Description**
ELN$LAT_CLEAR_COUNTERS	Clears a VAXELN service node's counters.
ELN$LAT_CREATE_PORT	Creates a VAXELN LAT port on a VAXELN service node.
ELN$LAT_CREATE_SERVICE	Creates a service to be offered by a VAXELN service node.
ELN$LAT_DELETE_PORT	Deletes a VAXELN LAT port.
ELN$LAT_DELETE_SERVICE	Deletes a service that is offered by a VAXELN service node.
ELN$LAT_SET_NODE	Sets a VAXELN service node's characteristics.
ELN$LAT_SET_PORT	Associates a dedicated LAT port with application service; or associates an application LAT port on a VAXELN service node with a remote port on a terminal server.
ELN$LAT_SET_SERVICE	Sets the characteristics of a service being offered by a VAXELN service node.
ELN$LAT_SHOW_CHAR	Returns a VAXELN service node's characteristics.
ELN$LAT_SHOW_COUNTERS	Returns performance and error statistics for a VAXELN service node or for all terminal servers connected to a VAXELN service node.
ELN$LAT_SHOW_PORT	Returns the characteristics of a VAXELN service node's LAT ports.

Routine	Description
ELN$LAT_SHOW_SERVERS	Returns the characteristics of terminal servers known to a VAXELN service node.
ELN$LAT_START_NODE	Activates the LAT protocol on a VAXELN service node.
ELN$LAT_STOP_NODE	Stops the LAT protocol on a VAXELN service node.

To use these procedures, you must also include the appropriate modules from the VAXELN runtime libraries.

Language	Module
VAXELN Pascal	$LAT_UTILITY
C	$vaxelnc and $lat_utility
FORTRAN	'ELN$FORTRAN_DEFS.FOR'

NOTE

The LAT utility procedures are in the shareable image LATSHR.EXE. If you dynamically load programs that use LAT utility procedures into a VAXELN system, you should specify ELN$:LATSHR.EXE in the **Guaranteed image list** entry on the System Builder's System Characteristics Menu when you build that system.

For descriptions of the LAT utility routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual.*

11.2.2 Creating a VAXELN LAT Port

A *VAXELN LAT port* is a service node structure for terminal I/O operations. The VAXELN LAT driver supports three types of LAT ports: interactive, dedicated, and application. The driver dynamically creates interactive LAT ports that offer the ECL service to terminal server users. A dedicated LAT port offers an application service (a service other than ECL) to terminal server users. An application LAT port lets application programs running on the service node gain access to a remote terminal's dedicated or application LAT port by using the LATCP command CREATE PORT or a call to the ELN$LAT_CREATE_PORT procedure.

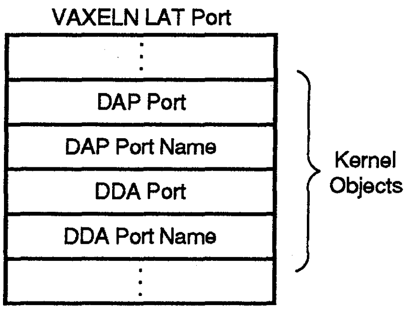
A call to ELN$LAT_CREATE_PORT must specify the port connected in a circuit to a LAT control port, a LAT port name, and a LAT port type. The following call to ELN$LAT_CREATE_PORT creates an application LAT port named LTA0:

```
MODULE create_a_lat_port;

INCLUDE $LAT_UTILITY;

PROGRAM create_lat_port;

VAR
  lat_ctrl_port : PORT;
    .
    .
    .
BEGIN

  { Create a VAXELN message port. }

  CREATE_PORT(lat_ctrl_port);

  { Connect that message port in a circuit to the local LAT control port. }

  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME :=  '$LAT_CONTROL');

  { Create an application LAT port named LTA0. }

  ELN$LAT_CREATE_PORT(CIRCUIT := lat_ctrl_port,
                      PORT_NAME := 'LTA0',
                      PORT_TYPE := LAT$APPLICATION);
    .
    .
    .
END.
END;
```

The LAT driver associates two VAXELN message ports with each LAT port that it creates: a *DAP port* for file- and record-oriented I/O and a *DDA port* for accessing serial line devices and managing connections between VAXELN LAT ports and remote terminal server ports. In the case of interactive LAT ports, the driver associates the DAP port with the name LTA*n* and associates the DDA port with the name LTA*n*$ACCESS, where *n* identifies a unique port name. When you create a dedicated or application LAT port, the driver associates the DAP port with the port name you specify with the CREATE PORT command or ELN$LAT_CREATE_PORT procedure. The driver also associates the DDA port with a port name of the form *port-name*$ACCESS, where *port-name* is the name of the DAP port.

Figure 11–2 distinguishes a VAXELN LAT port from its DAP and DDA VAXELN message ports.

Figure 11–2: VAXELN LAT Port



MLO-004289

The LAT host services include a set of port utility procedures for managing a connection between a VAXELN LAT port and a remote port or service on a terminal server. To use these procedures, an application program must connect to a LAT port's DDA port (see Section 11.2.3).

11.2.3 Connecting to a DDA Port

To use the VAXELN LAT port utility procedures, an application must create a port and connect that port in a circuit to a VAXELN LAT port's DDA port. The DDA port provides an interface for accessing serial line devices. The following example builds upon the example in Section 11.2.1 by showing how you might connect to a DDA port:

```
MODULE map_a_dda_port;

INCLUDE $LAT_UTILITY;

PROGRAM map_dda_port;

VAR
  lat_ctrl_port, dda_interface_port : PORT;
    .
    .
    .
BEGIN
  { Create a VAXELN message port. }

  CREATE_PORT(lat_ctrl_port);

  { Connect that port in a circuit to the local LAT control port. }

  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');

  { Create VAXELN LAT port named LTA0.  Driver creates DAP and DDA }
  { ports named LTA0 and LTA0$ACCESS, respectively.              }

  ELN$LAT_CREATE_PORT(CIRCUIT := lat_ctrl_port,
                      PORT_NAME := 'LTA0',
                      PORT_TYPE := LAT$APPLICATION);

  { Create a VAXELN message port for connecting to the DDA port. }

  CREATE_PORT(dda_interface_port);

  { Connect that port in a circuit to the LAT port's DDA port. }

  CONNECT_CIRCUIT(dda_interface_port, DESTINATION_NAME := 'LTA0$ACCESS')

  { Now call a port utility procedure. }

  ELN$LAT_MAP_PORT(CIRCUIT := dda_interface_port,
                   NEW_FIELDS := [LAT$SET_NODE, LAT$SET_QUEUED_STATUS,
                                  LAT$SET_PORT],
                   QUEUED_STATUS := TRUE,
                   REMOTE_SERVER_NAME := 'LAT100',
                   REMOTE_PORT_NAME := 'PORT_2',
                   SERVICE_NAME := '');
    .
    .
    .
END.
END;
```

Once a program connects a circuit to the LAT port's DDA port, the program specifies the port on its end of the connection in calls to the port utility procedures. The preceding example connects the port *dda_interface_port* to the DDA port named LTA0$ACCESS. Thus, the subsequent call to the ELN$LAT_MAP_PORT procedure can associate the LAT port LTA0 with the remote port named PORT_2 on the terminal server named LAT100.

You must specify a LAT port's DDA port in calls to the following port utility procedures:

Routine	Description
ELN$LAT_CONNECT_PORT	Connects an application LAT port on a VAXELN service node to a remote port or service offered by a terminal server.
ELN$LAT_DISCONNECT	Disconnects a VAXELN LAT port from a remote port offered by a terminal server.
ELN$LAT_MAP_PORT	Associates (maps) a dedicated LAT port with a service offered by a VAXELN service node; or associates an application LAT port on a VAXELN service node with a remote port or service offered by a terminal server.
ELN$LAT_SHOW_PORT_MAPPING	Returns mapping information for a LAT port on a VAXELN service node.
ELN$LAT_WAIT_FOR_CONNECTION	Waits for a connection to be established between a dedicated LAT port on a VAXELN service node and a remote device port or service offered by a terminal server.

To use these procedures, you must also include the appropriate modules from the VAXELN runtime libraries.

Language	Module
VAXELN Pascal	$LAT_UTILITY
C	$vaxelnc and $lat_utility
FORTRAN	'ELN$FORTRAN_DEFS.FOR'

NOTE

The LAT utility procedures are in the shareable image LATSHR.EXE. If you dynamically load programs that use LAT utility procedures into a VAXELN system, you should specify ELN$:LATSHR.EXE in the **Guaranteed image list** entry on the System Builder's System Characteristics Menu when you build that system.

For descriptions of these routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*. For more information about DDA or establishing a circuit with a DDA port, see Section 14.4.5.

11.3 Managing VAXELN Service Nodes

You can manage a VAXELN service node's characteristics and activities by calling LAT host service utility procedures from an application program. The procedures let you:

- Retrieve and set service node characteristics, Section 11.3.1
- Manage service node services, Section 11.3.2
- Retrieve port characteristics, Section 11.3.3
- Retrieve terminal server characteristics, Section 11.3.4
- Monitor a LAT environment's performance and error statistics, Section 11.3.5

11.3.1 Retrieving and Setting Service Node Characteristics

The LAT driver stores a service node's characteristics in a characteristics record. The values in this record identify the following node characteristics:

- Name
- Identification string
- Enabled LAT network groups
- Service announcement message time interval
- LAT driver state (active or inactive)
- LAT protocol version

An application program can retrieve a service node's characteristics by calling the ELN$LAT_SHOW_CHAR procedure. and ELN$TTY_GET_CHARACTERISTICS procedures This procedure allocates a characteristics record that the application program can access to retrieve service node characteristics. A call to ELN$LAT_SHOW_CHAR must specify the port connected in a circuit to a LAT control port, an argument that receives the version number of the characteristics record, and a pointer that points to the service node's characteristics record. For example:

```
VAR
  lat_ctrl_port : PORT;
  char_record_version, status : INTEGER;
  char_record : ^LAT$NODE_CHAR;
  one_shown : BOOLEAN := FALSE;
    .
    .
    .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
    .
    .
    .
  ELN$LAT_SHOW_CHAR(CIRCUIT := lat_ctrl_port,
                    VERSION := char_record_version,
                    CHARACTERISTICS := char_record);

  WITH char_record^ DO
  BEGIN
    WRITELN('Node name = ', NAME);
    WRITELN('Groups enabled = )';
    FOR i := LAT$GROUP0 TO LAT$GROUP255 DO
      IF i IN GROUPS THEN
        BEGIN
          IF one_shown THEN WRITE(',');
          WRITE(i:1);
          one_shown := TRUE;
        END;
    WRITELN(')');
  END;
  DISPOSE(char_record);
    .
    .
    .
END.
```

This section of code allocates a service node's characteristics record and then accesses the fields containing the service node's name and enabled LAT network groups.

Deallocate the characteristics record when the record is no longer needed.

An application program can also change a service node's characteristics by calling the ELN$LAT_SET_NODE procedure. You can use this procedure to change all characteristics but the LAT driver's state and the LAT protocol version. The call to ELN$LAT_SET_NODE in the following section of code changes a service node's name, identification string, and enabled groups:

```
VAR
  lat_ctrl_port : PORT;
  msg_interval : LAT$MULTICAST;
  disable_grps : LAT$GROUPS;
  status : INTEGER;
   .
   .
   .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
   .
   .
   .
  ELN$LAT_SET_NODE(CIRCUIT := lat_ctrl_port,
                   NEW_FIELDS := [LAT$SET_NODE, LAT$SET_IDENT,
                                  LAT$ENABLE_GROUPS],
                   NODE_NAME := 'RTNODE',
                   NODE_IDENT := 'VAXELN Service Node -- RTNODE',
                   SECONDS := msg_interval,
                   ENABLE_GROUPS := [0,4,7],
                   DISABLE_GROUPS := disable_grps);
   .
   .
   .
END.
```

Sections 11.3.1.1 to 11.3.1.5 provide more information about VAXELN service node names, identification strings, network groups, multicast timers, and LAT driver states.

11.3.1.1 Node Names

A LAT service node must have a *node name* that consists of 1 to 16 ASCII characters and is unique within the LAT network.

If a service node is part of a DECnet network, the LAT service node name should be the same as the DECnet node name. The DECnet node name must be unique within the same logical Ethernet and must be unique within the entire DECnet network. If a node name is not defined for a service node, the LAT driver uses a node name of the form LAT-*nnnnnnnnnnnn*, where *nnnnnnnnnnnn* represents the hexadecimal string for the Ethernet controller's address.

11.3.1.2 Node Identification Strings

A *node identification string* is a string of up to 64 ASCII characters that describes a LAT service node. When the LAT driver is active, it advertises the string by including it in periodic service announcement messages it sends to terminal servers.

If you do not specify a node identification string, the default is the VAXELN system identification string.

11.3.1.3 LAT Network Groups

You can distribute a LAT network among LAT *network groups*. Groups help manage the size of terminal server data bases by limiting the number of service nodes for which the server maintains information.

By controlling groups, you can restrict message traffic between the terminal servers and service nodes in a LAT network. For a terminal server to establish a connection with a service node, the server must share at least one group with that service node. A terminal server ignores the messages it receives from service nodes that are not in one of the server's groups.

For example, suppose a LAT configuration consists of two terminal servers, TS1 and TS2, and the service node RTNODE. Assume that the following groups are enabled for each:

Device	Groups Enabled
TS1	1,7
TS2	0,6
RTNODE	0

Initially, terminal server TS2 can communicate with service node RTNODE because the group 0 is enabled for both devices. If you use the LATCP command SET NODE or the ELN$LAT_SET_NODE procedure to enable group 7 for RTNODE, both terminal servers can communicate with that service node.

Group 0 is enabled by default for all service nodes and terminal servers. If you do not want group 0 enabled, you must disable it.

11.3.1.4 Multicast Timer

The *multicast timer* determines the time between a service node's service announcement messages. Service nodes send announcement messages to terminal servers to advertise the services that they offer. The messages include the following information:

- Node name
- Identification string
- Group designations
- Service names
- Service identification strings
- Service ratings

By default, the LAT driver sends the announcement messages every 60 seconds. However, the time interval can range from 10 to 255 seconds.

If you specify a larger value for the multicast timer, the LAT driver sends service announcement messages less frequently. Thus, a larger value minimizes network overhead but causes terminal server users to wait longer for services to become available after a server reboot, or after recovering from a network problem. Infrequent message announcements can also affect a server's load balancing.

Smaller multicast time values cause the LAT driver to consume more network resources because it sends service announcement messages more frequently.

Multicast service announcement messages broadcast service node characteristic changes. When you change a service node's characteristics, the LAT driver notifies the servers in the LAN by including the changed information in the service node's announcement messages.

11.3.1.5 Service Node States

A service node can be active or inactive. When active, the service node periodically sends service announcement messages to terminal servers in the LAT network. An active service node can also accept connection requests from terminal servers. For example, you can issue a connect request from an interactive terminal.

When inactive, a service node driver does not send service announcement messages to or accept connection requests from terminal servers until you start the LAT protocol with the LATCP command START NODE or a call to the ELN$LAT_START_NODE procedure. By using an initial inactive state, an application program can set up a service node before the LAT driver establishes connections with terminal servers. For example, an application program can set the service node's characteristics or create ports for establishing connections with dedicated services or application devices.

11.3.2 Managing Service Node Services

A *service* is a resource offered by a VAXELN service node or a terminal server on the LAT network. A VAXELN service node can offer up to eight uniquely named services, with each service offering all of the node's resources.

You can manage a service node's services by calling LAT host service utility procedures from an application program. The procedures let you:

- Create and delete services
- Change service characteristics
- Advertise services

11.3.2.1 Creating and Deleting Services

To add services to and delete services from a service node's list of offerings, use the ELN$LAT_CREATE_SERVICE and ELN$LAT_DELETE_SERVICE procedures, respectively. Each service offered by a service node has a name, identification string, and rating. The terminal server uses the *service rating* to balance the loads of service nodes in a LAT network.

When you create a service, you must specify the port connected in a circuit to a LAT control port, a service name, a service identification string, and a Boolean value indicating whether you want the driver to use a service rating that you specify or the service rating that it generates. You must also specify a link argument; however, this argument is reserved for future use.

The following call to ELN$LAT_CREATE_SERVICE creates a service named RT_SERVICE:

```
VAR
  lat_ctrl_port : PORT;
  service_rating : LAT$SERVICE_RATING;
  link_count, status : INTEGER;
  link_names : LAT$LINK_NAME_LIST;
    .
    .
    .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
    .
    .
    .
  ELN$LAT_CREATE_SERVICE(CIRCUIT := lat_ctrl_port,
                         SERVICE_NAME := 'RTSERVICE',
                         SERVICE_IDENT := 'Real Time Service -- RTSERVICE',
                         STATIC_SERVICE_RATING := FALSE,
                         SERVICE_RATING := service_rating,
                         LINK_COUNT := link_count,
                         LINK_NAMES := link_names);
    .
    .
    .
END.
```

The value FALSE specified for the static service rating tells the driver to use a dynamically determined service rating and to ignore any user-specified service rating.

When you delete a service, you only need to specify the port connected in a circuit to the LAT control port and the name of the service you want to delete.

The following sections provide more information about VAXELN service names, identification strings, and ratings.

Service Names

Service names are strings that can consist of up to 16 of the following ASCII characters:

- Alphanumeric characters — A to Z, a to z, and 0 to 9
- A subset of the international character set — decimal character values 192(10) to 253(10)

- Punctuation characters — dollar sign ($), hyphen (-), period (.), and underscore (_)

The names of services that a service node offers must be unique. However, multiple service nodes in a LAT network can share a service name. Having multiple service nodes in a LAT network offer the same service provides for *failover*.

Failover is a terminal server service that takes over if a session is disrupted because a service node becomes unavailable. When a session is disrupted, the terminal server uses the automatic failover service to search for other nodes that offer the service that was being used by the disrupted session. If the server finds one or more such nodes, it tries to connect to the service on the node that is least busy.

When the LAT driver executes, it creates a default service representing the name of the VAXELN service node. If no name is defined for the service node, the driver uses a node name and service name of the form LAT-*nnnnnnnnnnnn*, where *nnnnnnnnnnnn* represents the hexadecimal string for the Ethernet controller's address.

Service Identification Strings

A service identification string is a string of up to 64 ASCII characters that describes a service offered by a service node. The service node includes service identification strings in its service announcement messages.

If you do not specify a service identification string, the default is the VAXELN system identification string.

Service Ratings

Service ratings provide a system load balancing feature. The LAT driver running on each service node that offers a particular service can dynamically calculate a service rating for that service. If a service's rating is calculated dynamically, the driver recalculates the rating every time the multicast timer expires. Thus, the rating reflects the availability of resources on the service node.

A service rating can range from 0 to 255, where 0 indicates that a service is not available and 255 indicates that a service is highly available. Dynamic service ratings are based on a service node's activity and processor type. Generally, services offered by nodes experiencing high levels of activity receive low ratings to inhibit new connections.

Terminal servers use service ratings to balance the load among service nodes that offer the same service; servers establish connections with a service on the least busy service node that offers that service.

11.3.2.2 Changing Service Characteristics

An application program can change a service's characteristics by calling the ELN$LAT_SET_SERVICE procedure. A call to this procedure must specify the port connected in a circuit to the LAT control port, a value for a new fields argument, and values for characteristics you want to change: name, identification string, and rating. The new fields argument identifies the characteristics you will be changing. You must also specify two link arguments. However, these arguments are reserved for future use.

Normally, the LAT driver generates adequate service ratings. However, you can override the driver's rating by assigning a static service rating to a service. For example, suppose a service needs to run on a service node that is not busy. An application program could let the LAT driver set up the service's initial service rating and then adjust the rating to inhibit new connections. The following call to ELN$LAT_SET_SERVICE changes a service's service rating:

```
VAR
  lat_ctrl_port : PORT;
  service_ident : LAT$IDENT_STRING;
  link_count, status : INTEGER;
  link_names : LAT$LINK_NAME_LIST;
    .
    .
    .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
    .
    .
    .
  { Lower service rating to inhibit new connections. }
  ELN$LAT_SET_SERVICE(CIRCUIT := lat_ctrl_port,
                      NEW_FIELDS := [LAT$SET_STATIC_RATING],
                      SERVICE_NAME := 'RTSERVICE',
                      SERVICE_IDENT := service_ident,
                      SERVICE_RATING := 10,
                      LINK_COUNT := 0,
                      LINK_NAMES := link_names);
     .
     .
     .
END.
```

When you specify a static service rating, you disable a dynamic service rating.

11.3.2.3 Advertising Services

All of the service nodes in a LAT network advertise their services by multicasting service announcement messages to all terminal servers in the LAN. A service node starts multicasting these messages as soon as it becomes active. You can activate a service node from an application program by calling the ELN$LAT_START_NODE procedure as follows:

```
VAR
  lat_ctrl_port : PORT;
  link_names : LAT$LINK_NAME_LIST;
  status : INTEGER;
     .
     .
     .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
     .
     .
     .
  ELN$LAT_START_NODE(CIRCUIT := lat_ctrl_port,
                     LINK_COUNT := 0,
                     LINK_NAMES := link_names);
     .
     .
     .
END.
```

A call to ELN$LAT_START_NODE must specify the port connected in a circuit to a LAT control port and two link arguments. The link arguments are reserved for future use.

As a terminal server receives service announcement messages, it checks whether the service nodes sending the messages share one of its enabled LAT network groups. If a service node and a terminal server share a group, the server accepts the message and adds the name of the service node and the names of the services the node offers to its services data base.

To shut down a VAXELN service node, call the ELN$LAT_STOP_NODE procedure. You should precede the call with a call to ELN$LAT_SET_NODE and identify the reason for the shut-down in its *node_ident* argument. For example:

```
VAR
  lat_ctrl_port : PORT;
  seconds : LAT$MULTICAST
  disable_grps, enable_grps : LAT$GROUPS;
  link_names : LAT$LINK_NAME_LIST;
  status : INTEGER;
   .
   .
   .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
   .
   .
   .
  ELN$LAT_SET_NODE(CIRCUIT := lat_ctrl_port,
                   NEW_FIELDS := [LAT$SET_IDENT],
                   NODE_NAME := 'RTNODE',
                   NODE_IDENT := 'RTNODE shutting down for PM...',
                   SECONDS := seconds,
                   ENABLE_GROUPS := endable_grps,
                   DISABLE_GROUPS := disable_grps);

  ELN$LAT_STOP_NODE(CIRCUIT := lat_ctrl_port,
                    LINK_COUNT := 0,
                    LINK_NAMES := LAT$LINK_NAME_LIST);
   .
   .
   .
END.
```

11.3.3 Retrieving LAT Port Characteristics

A VAXELN LAT port has the following characteristics:

- Port name
- Port type
- Queue status
- Remote server name
- Remote port name
- Service name
- Actual remote server name
- Actual remote port name

An application program can retrieve a LAT port's characteristics by calling the ELN$LAT_SHOW_PORT procedure. This procedure allocates a characteristics record from which the application program can retrieve the characteristics.

A call to ELN$LAT_SHOW_PORT must specify the port connected in a circuit to a LAT control port, the name of the LAT port whose characteristics record is to be accessed, a LAT port type, and the name of a user-specified procedure to be invoked by ELN$LAT_SHOW_PORT. For example:

```
VAR
  lat_ctrl_port : PORT;
    .
    .
    .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
    .
    .
    .
  ELN$LAT_SHOW_PORT(CIRCUIT := lat_ctrl_port,
                    PORT_NAME := 'LTA0',
                    PORT_TYPES := [LAT$APPLICATION, LAT$DEDICATED,
                                   LAT$INTERACTIVE],
                    SHOW_PORT := show_lat_port);
    .
    .
    .
END.

PROCEDURE show_lat_port OF TYPE LAT$SHOW_PORT_ROUTINE;
```

```
BEGIN
  WITH port_characteristics^ DO
    BEGIN
      WRITELN('Port name = ', name);
      WRITE('Port type = ');
      CASE port_type OF
        LAT$RESERVED_PORT : WRITELN('Reserved');
        LAT$APPLICATION : WRITELN('Application');
        LAT$DEDICATED : WRITELN('Dedicated');
        LAT$INTERACTIVE : WRITELN('Interactive');
        OTHERWISE WRITELN('Unknown');
      END;
   .
   .
   .
    END;
END;
```

The ELN$LAT_SHOW_PORT procedure invokes the user-specified procedure only if the driver finds a characteristics record for the specified LAT port. When the call to ELN$LAT_SHOW_PORT in the preceding example executes, the driver searches for a characteristics record for the LAT port named LTA0. If it finds one, the user-defined procedure *show_lat_port* is invoked.

You can also access a LAT port's mapping information by calling the port utility procedure ELN$LAT_SHOW_PORT_MAPPING. A call to this procedure specifies a port connected in a circuit to the LAT port's DDA port. Other arguments receive the LAT port's name, type, and queue status, and depending on the port's mapping, a remote terminal server name, a remote port name, and a service name.

The following sections give more information about LAT port names, queue statuses, remote server names, and remote port names. For information about service names, see Section 11.3.2.1.

11.3.3.1 LAT Port Names

VAXELN LAT port names are strings that can consist of up to 16 of the following ASCII characters:

- Alphanumeric characters — A to Z, a to z, and 0 to 9
- A subset of the international character set — decimal character values 192 to 253
- Punctuation characters — dollar sign ($), hyphen (-), period (.), and underscore (_)

A service node's LAT port names must be unique.

11.3.3.2 Queue Statuses

A LAT port's queue status indicates whether connection requests to a remote device port or service are to be queued. If the remote port is busy and queuing is enabled on the terminal server and the VAXELN service node, the remote request is queued. If queuing is disabled, the terminal server rejects connection requests when the device or service is busy.

11.3.3.3 Remote Server Names

A remote server name is a string of up to 255 characters that identifies a terminal server that supports an application device or service. You get a remote server name from the terminal server manager.

11.3.3.4 Remote Port Names

A remote port name is a string of up to 255 characters that identifies a terminal server port that supports an application device or service. You get a remote port name from the terminal server manager.

11.3.4 Retrieving Terminal Server Characteristics

When an application program establishes a virtual circuit with a remote terminal server, the server becomes known to the service node. The LAT driver creates a terminal server characteristics record for each known terminal server. A terminal server characteristics record stores the following characteristics:

- Name
- State (active or inactive)
- Address
- Number of active users
- Link name

An active terminal server is connected to a VAXELN service node. Inactive servers are known to the service node, but are not connected to the node.

An application program can retrieve the characteristics of terminal servers known to a service node by calling the ELN$LAT_SHOW_SERVERS procedure. This procedure allocates a characteristics record from which the application program can retrieve the characteristics.

A call to ELN$LAT_SHOW_SERVERS must specify the port connected in a circuit to a LAT control port, a Boolean value that indicates whether the driver is to return characteristics for active and inactive servers, and the name of a user-specified procedure to be invoked by ELN$LAT_SHOW_SERVERS. For example:

```
VAR
  lat_ctrl_port : PORT;
    .
    .
    .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
    .
    .
    .
  ELN$LAT_SHOW_SERVERS(CIRCUIT := lat_ctrl_port,
                       INACTIVE := TRUE,
                       SHOW_SERVER := show_lat_server);
    .
    .
    .
END.

PROCEDURE show_lat_server OF TYPE LAT$SHOW_PORT_ROUTINE;

BEGIN
  WITH server_characteristics^ DO
    BEGIN
      WRITELN('Server name = ', SERVER_NAME);
      WRITE('Server is ');
      IF active THEN WRITELN('active') ELSE WRITELN('inactive');
    .
    .
    .
    END;
    .
    .
    .
END;
```

The ELN$LAT_SHOW_SERVERS procedure invokes the user-specified procedure for all known servers or only active servers, depending on the value of the *inactive* argument. When the call to ELN$LAT_SHOW_SERVERS in the preceding example executes, the driver uses the *show_lat_server* procedure to return server characteristic records for known terminal servers.

11.3.5 Monitoring LAT Network Performance and Error Statistics

The LAT driver maintains performance and error counters for a service node and the terminal servers logically connected to a service node. The LAT driver stores the following counters in node, server, and device counter records:

Node Counters	Server Counters	Device Counters
Circuit timeouts	Invalid messages	Line name
Discarded output bytes	Invalid slots	Seconds since last zeroed
Last transmit failure	Out-of-sequence frames	Receive frames
No transmit buffer	Receive frames	Receive multicast frames
Protocol errors	Retransmissions	Receive errors
Protocol bit mask	Server name	Bytes received
Receive frames	Transmit frames	Multicast bytes received
Receive errors		Data overruns
Receive duplicates		Local buffer errors
Resource errors		Transmit frames
Retransmissions		Transmit multicast frames
Solicitation failures		Frames sent, multiple collisions
Transmit frames		Frames sent, single collision
Transmit errors		Frames sent, initially deferred
Unit timeouts		Bytes sent
		Multicast bytes sent
		Transmit errors

Node Counters	Server Counters	Device Counters
		Transmit collisions detect check failure
		Unrecognized frame destination
		User buffer unavailable
		Receive errors bit mask
		Transmit errors bit mask

An application program can monitor these counters by calling the ELN$LAT_CLEAR_COUNTERS and ELN$LAT_SHOW_COUNTERS procedures. Use the ELN$LAT_CLEAR_COUNTERS procedure to initialize the node counters. You can then call ELN$LAT_SHOW_COUNTERS at various points in the program to retrieve various counter values.

The ELN$LAT_SHOW_COUNTERS procedure allocates a record for the type of counter the application program is to access. A call to the procedure must specify the port connected in a circuit to a LAT control port, a counter type, a Boolean value that indicates whether the driver is to return counters for active and inactive servers (if you specify the server counter type), and the name of a user-specified procedure to be invoked by ELN$LAT_SHOW_COUNTERS. For example:

```
VAR
  lat_ctrl_port : PORT;
  link_name : LAT$LINK_NAME;
    .
    .
    .
BEGIN
  CREATE_PORT(lat_ctrl_port);
  CONNECT_CIRCUIT(lat_ctrl_port, DESTINATION_NAME := '$LAT_CONTROL');
    .
    .
    .
  ELN$LAT_SHOW_COUNTERS(CIRCUIT := lat_ctrl_port,
                        COUNTER_TYPE := LAT$SERVER,
                        INACTIVE := FALSE,
                        LINK := link_name,
                        SHOW_COUNTER := show_server_counters);
    .
    .
    .
END.
```

```
PROCEDURE show_server_counters OF TYPE LAT$SHOW_COUNTER_ROUTINE;

BEGIN
  WITH server_counters^ DO
    BEGIN
      WRITELN('Server name = ', SERVER_NAME);
    .
    .
    .
    END;
   .
   .
   .
END;
```

When the call to ELN$LAT_SHOW_COUNTERS in the preceding example executes, the driver uses the *show_server_counters* procedure to return server counter records for active terminal servers.

If you specify the counter type LAT$NODE, the ELN$LAT_SHOW_COUNTERS procedure invokes the user-declared procedure to return a service node counter record. Similarly, if you specify the counter type LAT$DEVICE, the procedure invokes the user-declared procedure to return device counter records.

11.4 Setting Up a Dedicated Service Environment

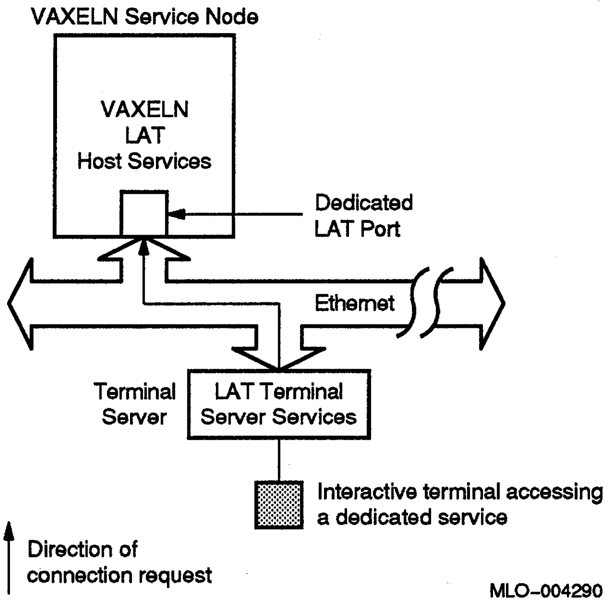
The VAXELN LAT host services provide support that lets a VAXELN service node offer user-created dedicated services, such as data entry and on-line status programs, to terminal server users. When you create a service on a VAXELN service node, that service offers ECL by default. To offer a dedicated service instead of ECL, you must associate the service with a dedicated port.

You initiate a connection to a dedicated service running on a VAXELN service node from a device attached to a terminal server, as shown in Figure 11–3.

To make a service other than ECL available to a terminal attached to a terminal server, you must do the following:

1. Connect to a LAT control port
2. Create a service
3. Create a dedicated LAT port

Figure 11–3: Dedicated Service Environment



4. Associate the dedicated LAT port with the service
5. Access the dedicated LAT port from an application program
6. Request notification of a terminal server connection (optional)

The rest of this section uses the sample application module *sample_lat_dedic_srvc* and callout text to illustrate these steps. The module executes as one job of a two-job application that offers the VAXELN Display Utility as a dedicated service to terminal server users. The sample module creates a LAT service named EDISPLAY and the dedicated LAT port LTA0. The module then associates the LAT port with the service EDISPLAY.

Once the port/service association is made, the module activates the LAT protocol. Using the protocol, the LAT driver multicasts service announcement messages to terminal servers in the LAT network, advertising the EDISPLAY service. While the driver multicasts these messages, the application module waits for a terminal server user to connect to the service. When a connection is established, the application module creates the EDISPLAY job and then waits for that job to execute. When the terminal server user exits the Display Utility, the application module waits for another connection request.

Example 11–1 assumes that the LAT driver is built into the VAXELN system with the LAT protocol inactive. It also assumes that the LAT application module *sample_lat_dedic_srvc* is built into the system with the initial state set to *RUN* and that EDISPLAY is built into the system with the initial state set to *NORUN*.

Example 11–1: LAT Dedicated Service

```
MODULE sample_lat_dedic_srvc;

INCLUDE $LAT_UTILITY, $KERNELMSG;

PROGRAM example(INPUT, OUTPUT);

VAR
  lat_control_port, dda_interface_port : PORT;
  job_port, edisplay_exit_port : PORT;
  link_names : LAT$LINK_NAME_LIST;
  stat : INTEGER;
  mid : MESSAGE;
  mptr : ^INTEGER;

BEGIN

  { Create a VAXELN message port, then connect that port in a circuit to
  { a LAT control port.
  {}

  CREATE_PORT(lat_control_port);                                   ❶
  CONNECT_CIRCUIT(lat_control_port,
                DESTINATION_NAME :=  '$LAT_CONTROL');

  { Create the LAT service that is to offer EDISPLAY. }

  ELN$LAT_CREATE_SERVICE(CIRCUIT := lat_control_port,              ❷
                         SERVICE_NAME := 'EDISPLAY',
                         SERVICE_IDENT := 'VAXELN Display Utility',
                         STATIC_SERVICE_RATING := FALSE,
                         SERVICE_RATING := 0,
                         LINK_COUNT := 0,
                         LINK_NAMES := link_names);

  { Create a dedicated LAT port. }
```

Example 11–1 Cont'd on next page

Example 11–1 (Cont.): LAT Dedicated Service

```
ELN$LAT_CREATE_PORT(CIRCUIT := lat_control_port,                   ❸
                    PORT_TYPE := LAT$DEDICATED,
                    PORT_NAME := 'LTA0');

{ Associate the dedicated LAT port LTA0 with the LAT service EDISPLAY. }

ELN$LAT_SET_PORT(CIRCUIT := lat_control_port,                      ❹
                 NEW_FIELDS := [LAT$SET_SERVICE],
                 PORT_NAME := 'LTA0',
                 QUEUE := TRUE,
                 REMOTE_SERVER_NAME := '',
                 REMOTE_PORT_NAME := '',
                 SERVICE_NAME := 'EDISPLAY',
                 LINK_NAME := '');

{ Activate the LAT protocol to advertise service. }

ELN$LAT_START_NODE(CIRCUIT := lat_control_port,                     ❺
                   LINK_COUNT := 0,
                   LINK_NAMES := link_names);

{ Access the LAT port. }

CREATE_PORT(dda_interface_port);                                    ❻
CONNECT_CIRCUIT(dda_interface_port,
                DESTINATION_NAME := 'LTA0$ACCESS',
                STATUS := stat);

{ Create a port to be notified when EDISPLAY exits. }

CREATE_PORT(edisplay_exit_port);

WHILE TRUE DO
  BEGIN

    { Wait for a terminal server user to connect to the EDISPLAY service. }

    ELN$LAT_WAIT_FOR_CONNECTION(CIRCUIT := dda_interface_port,       ❼
                                STATUS := stat);

    {
    {   If the wait fails because the terminal server user
    {   disconnected the session, reestablish a circuit connection
    {   with the LAT port's DDA port.
    {}
```

Example 11–1 Cont'd on next page

Example 11–1 (Cont.): LAT Dedicated Service

```
      IF stat = KER$_DISCONNECT THEN
        BEGIN
          DISCONNECT_CIRCUIT(dda_interface_port);
          CONNECT_CIRCUIT(dda_interface_port,
                          DESTINATION_NAME := 'LTA0$ACCESS',
                          STATUS := stat);
          ELN$LAT_WAIT_FOR_CONNECTION(CIRCUIT := dda_interface_port,
                                      STATUS := stat);
        END;

      { If an error occurs, terminate this job. }

      IF NOT(ODD(stat)) THEN
        BEGIN
          WRITELN('Exiting, status is: ', stat:1);
          EXIT(EXIT_STATUS := stat);
        END;

      { Run EDISPLAY on the VAXELN LAT port and wait for it to complete. }

      IF ODD(stat) THEN
        CREATE_JOB(job_port,                                              ❽
                   'EDISPLAY',
                   'LTA0:',         { Program argument 1 }
                   'LTA0:',         { Program argument 2 }
                   'LTA0:',         { Program argument 3 }
                   NOTIFY := edisplay_exit_port,
                   STATUS := stat);

      IF ODD(stat) THEN
        BEGIN

          { Wait for the EDISPLAY job to terminate. }

          WAIT_ANY(edisplay_exit_port);                                   ❾
          RECEIVE(mid, mptr, edisplay_exit_port, STATUS := stat);
          DELETE(mid, STATUS := stat);

          { Disconnect the session with the terminal server user. }

          ELN$LAT_DISCONNECT_PORT(CIRCUIT := dda_interface_port,          ❿
                                  STATUS := stat);
        END;
  END;
END.  { Program }
END;  { Module }
```

❶ **Connect to a LAT control port.** Create a VAXELN message port and connect that port in a circuit to a LAT control port. The sample module creates the message port *lat_control_port* and connects that port in a circuit to the local control port $LAT_CONTROL. For information about connecting to a LAT control port, see Section 11.2.1.

❷ **Create a service.** Create a service to be offered by the service node by calling the ELN$LAT_CREATE_SERVICE procedure. The sample module creates the service named EDISPLAY. The procedure call specifies FALSE for the static service rating argument. Thus, the LAT driver will calculate the service rating dynamically. For information about services, see Section 11.3.2.

❸ **Create a dedicated LAT port.** Create a dedicated LAT port by calling the ELN$LAT_CREATE_PORT procedure with the port type LAT$DEDICATED. The sample module creates a dedicated LAT port named LTA0. The LAT driver creates a DAP message port and a DDA message port for the LAT port and associates the ports with the names LTA0 and LTA0$ACCESS. For information about creating LAT ports, see Section 11.2.2.

❹ **Associate the dedicated LAT port with the service.** Associate the dedicated LAT port with the service by calling the ELN$LAT_MAP_PORT or ELN$LAT_SET_PORT procedure. Use ELN$LAT_SET_PORT to make the association using the LAT control interface. Use ELN$LAT_MAP_PORT to associate the port and service using the DDA interface.

A call to the ELN$LAT_SET_PORT procedure must specify the port connected in a circuit to a LAT control port, the name of a LAT port, and a value for a new fields argument. You can change the LAT port's queue status, remote server name, remote port name, and service name. The new fields argument identifies the LAT port characteristics you will be changing.

You must also specify a link argument. This argument is reserved for future use.

The call to ELN$LAT_SET_PORT in the sample module associates the LAT port LTA0 with the service EDISPLAY.

A call to the ELN$LAT_MAP_PORT procedure must specify the port connected in a circuit to a LAT port's DDA port and a value for a new fields argument. You can also change the LAT port's queue status, remote server name, remote port name, and service name. The new fields argument identifies the LAT port characteristics you will be changing.

The following procedure call makes the same association using the LAT port's DDA interface:

```
ELN$LAT_MAP_PORT(CIRCUIT := dda_interface_port,
                 NEW_FIELDS := [LAT$SET_SERVICE],
                 QUEUED_STATUS := TRUE,
                 REMOTE_SERVER_NAME := '',
                 REMOTE_PORT_NAME := '',
                 SERVICE_NAME := 'EDISPLAY');
```

❺ **Activate the LAT protocol.** Advertise the service node's services by activating the LAT protocol with a call to ELN$LAT_START_NODE. When you activate the protocol, the driver starts advertising services in announcement messages that it multicasts to the terminal servers in the LAT network. The call to ELN$LAT_START_NODE in the sample module causes the LAT driver to activate the LAT protocol. The driver then starts advertising the service EDISPLAY.

You can skip this step if the LAT protocol is already active.

❻ **Access the dedicated LAT port from the application program.** An application program running on the service node can access a dedicated LAT port by using a call to a language-dependent open statement or a call to CONNECT_CIRCUIT. Use a call to an open statement to perform file- and record-oriented I/O operations. Your application program accesses the LAT port's DAP port to process these I/O requests.

Use a call to CONNECT_CIRCUIT to connect to the LAT port's DDA port. You need to connect to the DDA port to perform operations that involve accessing serial line devices or managing a VAXELN LAT port connection to a remote terminal server port.

The sample module uses a call to CONNECT_CIRCUIT to establish a circuit connection with the DDA port LTA0$ACCESS. For information about connecting to a LAT port's DDA port, see Section 11.2.3.

❼ **Request notification of a terminal server connection.** A dedicated service application program can request notification of a terminal server connection by calling the ELN$WAIT_FOR_CONNECTION port utility procedure. A call to this procedure causes your program to wait for a connection to be established between a dedicated LAT port and a terminal server port. The application program unblocks when the terminal server connects to the service node.

A call to ELN$LAT_WAIT_FOR_CONNECTION procedure must specify a port connected in a circuit to the LAT port's DDA port. Since the sample module established a circuit between the port *dda_interface_port* and the DDA port LTA0$ACCESS, it can call ELN$WAIT_FOR_CONNECTION by specifying the port *dda_interface_port*.

❽ **Create a job for the service.** The sample module creates a job for EDISPLAY, specifying the LAT port name as program arguments.

❾ **Wait for the job to terminate.** Wait for the job to terminate by specifying the service's exit port in a WAIT_ANY procedure call. The sample module waits on the exit port *edisplay_exit_port*.

❿ **Disconnect the session between the service node and terminal server.** A session between a dedicated LAT port and a terminal server's device port terminates when one of the following occurs:

- The terminal server user disconnects the session (for example, by logging out).
- All open files to the dedicated LAT port are closed, and all DDA circuit connections to the dedicated LAT port are disconnected.
- The application program forces a disconnection by calling the ELN$LAT_DISCONNECT_PORT port utility procedure.

An application program can force a session to disconnect a dedicated LAT port from a remote terminal server port by calling the ELN$LAT_DISCONNECT_PORT port utility procedure. The procedure call must specify the port connected in a circuit to the LAT port's DDA port.

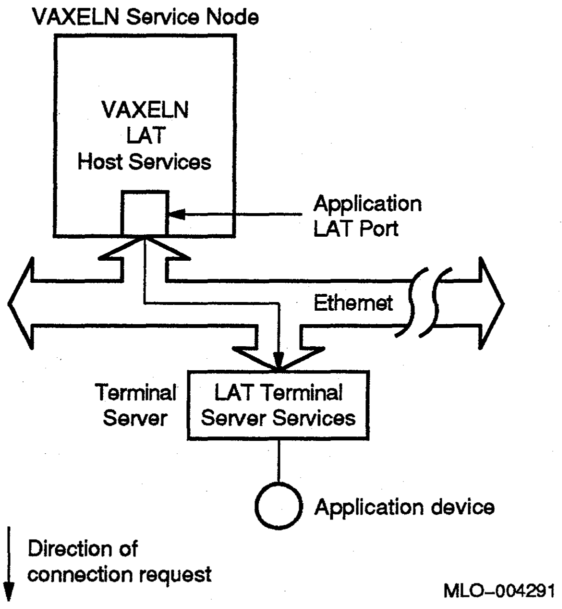
When a session terminates, all open files are closed and all DDA circuit connections between application programs and the dedicated LAT port on the service node are terminated.

11.5 Setting Up an Application Device Environment

The VAXELN LAT driver supports access to remote application devices attached to terminal servers. For example, application programs running on VAXELN service nodes in a LAT network can share a remote printer.

To access an application device, an application program associates a remote terminal server port with an application LAT port. As shown in Figure 11–4, once a program running on a service node makes the port association, it can initiate a connection to the terminal server to which the application LAT port is associated.

Figure 11–4: Application Device Environment



To communicate with a remote application device, you must do the following:

1. Create an application LAT port
2. Access the application LAT port from the application program
3. Associate the application LAT port with a terminal server device port or service
4. Issue a connection request to establish a session with the remote terminal server port

The rest of this section uses the sample application module *sample_lat_app_device* and callout text to illustrate these steps. The module executes as one job of a two-job application. This application uses a terminal attached to a terminal server as a device that displays VAXELN Display Utility output. The sample module creates the application LAT port LTA0. The module then accesses the LAT port by connecting to the LAT port's DDA port.

The module then associates the application LAT port with a port on a terminal server. Once the port/device association is made, the module activates the LAT protocol, allowing the service node and terminal server to communicate.

Example 11–2 assumes that the LAT driver is built into the VAXELN system with the LAT protocol inactive. It also assumes that the LAT application module *sample_lat_dedic_srvc* is built into the system with the initial state set to *RUN* and that EDISPLAY is built into the system with the initial state set to *NORUN*.

Example 11–2: LAT Application Service

```
MODULE sample_lat_app_device;

INCLUDE $LAT_UTILITY;

PROGRAM example(INPUT, OUTPUT);

VAR
  lat_control_port, dda_interface_port : PORT;
  job_port, edisplay_exit_port : PORT;
  link_names : LAT$LINK_NAME_LIST;
  stat, reject_reason : INTEGER;
  mid : MESSAGE;
  mptr : ^INTEGER;

BEGIN

  { Create a VAXELN message port, then connect that port in a circuit
  { to a LAT control port.
  {}

  CREATE_PORT(lat_control_port);                                        ❶
  CONNECT_CIRCUIT(lat_control_port,
                DESTINATION_NAME :=  '$LAT_CONTROL');

  { Create an application LAT port. }
```

Example 11–2 Cont'd on next page

Example 11–2 (Cont.): LAT Application Service

```
ELN$LAT_CREATE_PORT(CIRCUIT := lat_control_port,                    ❷
                    PORT_TYPE := LAT$APPLICATION,
                    PORT_NAME := 'LTA0');

{ Activate the LAT protocol. }

ELN$LAT_START_NODE(CIRCUIT := lat_control_port,                     ❸
                   LINK_COUNT := 0,
                   LINK_NAMES := link_names);

{ Access the LAT port. }

CREATE_PORT(dda_interface_port);                                    ❹
CONNECT_CIRCUIT(dda_interface_port,
                DESTINATION_NAME := 'LTA0$ACCESS',
                STATUS := stat);

{ Create a port to be notified when EDISPLAY exits. }

CREATE_PORT(edisplay_exit_port);

{ Associate the application LAT port LTA0 with the remote terminal
{ server port PORT_2.
{}

ELN$LAT_MAP_PORT(CIRCUIT := dda_interface_port,                     ❺
                 NEW_FIELDS := [LAT$SET_QUEUED_STATUS, LAT$SET_PORT],
                 QUEUED_STATUS := TRUE,
                 REMOTE_SERVER_NAME := 'LAT100',
                 REMOTE_PORT_NAME := 'PORT_2',
                 SERVICE_NAME := '');

{ Issue a connection request to connect the application LAT port
{ with a remote terminal server port.
{}

ELN$LAT_CONNECT_PORT(CIRCUIT := dda_interface_port,                 ❻
                     REJECT_REASON := reject_reason,
                     STATUS := status);

IF ODD(stat) THEN
  BEGIN

    { If the connection was established, execute the Display
    { Utility on the application LAT port.
    {}
```

Example 11–2 Cont'd on next page

Example 11–2 (Cont.): LAT Application Service

```
      CREATE_JOB(job_port,
             'EDISPLAY',
             'LTA0:',        { Program argument 1 }
             'LTA0:',        { Program argument 2 }
             'LTA0:',        { Program argument 3 }
              NOTIFY := edisplay_exit_port,
              STATUS := stat);

      IF ODD(stat) THEN
        BEGIN

          { An EDISPLAY job was created successfully.  Wait for it
          { to complete.
          {}

          WAIT_ANY(edisplay_exit_port);
          RECEIVE(mid, mptr, edisplay_exit_port, STATUS := stat);
          DELETE(mid, STATUS := stat);

          { Disconnect the session with the terminal server. }

          ELN$LAT_DISCONNECT_PORT(CIRCUIT := dda_interface_port, ❼
                                  STATUS := stat);
        END;
    END;
END.  { Program }
END;  { Module }
```

❶ **Connect to a LAT control port.** Create a VAXELN message port and connect that port in a circuit to a LAT control port. The sample module creates the message port *lat_control_port* and connects that port in a circuit to the local control port $LAT_CONTROL. For information about connecting to a LAT control port, see Section 11.2.1.

❷ **Create an application LAT port.** Create an application LAT port by calling the ELN$LAT_CREATE_PORT procedure with the port type LAT$APPLICATION. The sample module creates an application LAT port named LTA0. The LAT driver creates a DAP message port and a DDA message port for the LAT port and associates the ports with the names LTA0 and LTA0$ACCESS. For information about creating LAT ports, see Section 11.2.2.

❸ **Activate the LAT protocol.** Activate the LAT protocol with a call to ELN$LAT_START_NODE. When you activate the protocol, the driver starts advertising services in service announcement messages that it multicasts to the terminal servers in the LAT network. The call to ELN$LAT_START_NODE in the sample module causes the LAT driver to activate the LAT protocol.

You can skip this step if the LAT protocol is already active.

❹ **Access the application LAT port from the application program.** Use a call to CONNECT_CIRCUIT to connect the VAXELN message port to the LAT port's DDA port. The DDA port provides an interface for operations that access serial line devices or manage a VAXELN LAT port.

The sample module creates the VAXELN message port *dda_interface_port*, then uses a call to CONNECT_CIRCUIT to connect that port in a circuit to the DDA port LTA0$ACCESS. For information about connecting to a LAT port's DDA port, see Section 11.2.3.

❺ **Associate the application LAT port with the remote device.** Associate an application LAT port with a remote device port or service on a terminal server by calling the ELN$LAT_MAP_PORT or ELN$LAT_SET_PORT procedure. Use ELN$LAT_SET_PORT to make the association using the LAT control interface. Use ELN$LAT_MAP_PORT to associate the port and service using the DDA interface.

A call to the ELN$LAT_MAP_PORT procedure must specify the port connected in a circuit to a LAT port's DDA port and a value for a new fields argument. You can also change the LAT port's queue status, remote server name, remote port name, and service name. The new fields argument identifies the LAT port characteristics you will be changing.

The call to ELN$LAT_MAP_PORT in the sample module associates the LAT port LTA0 with the remote device port named PORT_2 on the terminal server LAT100.

A call to the ELN$LAT_SET_PORT procedure must specify the port connected in a circuit to a LAT control port, the name of a LAT port, and a value for a new fields argument. You can also change the LAT port's queue status, remote server name, remote port name, and service name. The new fields argument identifies the LAT port characteristics you will be changing.

Get the names of the remote terminal server and remote port by using the terminal server manager.

You must also specify a link argument. This argument is reserved for future use.

The following procedure call makes the same association from the LAT control port:

```
ELN$LAT_SET_PORT(CIRCUIT := lat_control_port,
                 NEW_FIELDS := [LAT$SET_QUEUE_STATUS, LAT$SET_PORT],
                 PORT_NAME := 'LTA0',
                 QUEUE := TRUE,
                 REMOTE_SERVER_NAME := 'LAT100',
                 REMOTE_PORT_NAME := 'PORT_2',
                 SERVICE_NAME := '',
                 LINK_NAME := '');
```

❻ **Connect to the application device.** You can issue a connection request from the application LAT port to the remote device port on a terminal server with a call to the ELN$LAT_CONNECT_PORT procedure. In the procedure call, you specify the port connected in a circuit to the LAT port's DDA port. The sample module connects the LAT port LTA0 to the device port PORT_2.

A call to ELN$LAT_CONNECT_PORT causes the LAT driver to send a request to a terminal server to establish a session between the service node and a terminal server device port or service. If the terminal server establishes a session, the LAT port is ready to be used for I/O and DDA operations. If the server rejects the connection request or if the request times out, the server returns an error code that identifies the reason for the connection failure.

The LAT driver initiates connection requests automatically when an application program performs read and write operations, if an application LAT port is associated with a terminal server and is not already connected in a session. However, if an automatic connection attempt fails, the I/O operation returns the status value ELN$DNR. It does not identify the reason for the connection failure.

❼ **Disconnect the session between the service node and terminal server.** An application program disconnects a session between an application LAT port and a terminal server's device port by calling the ELN$LAT_DISCONNECT_PORT port utility procedure. The procedure call must specify the port connected in a circuit to the LAT port's DDA port.

When a session terminates, all open files are closed and all DDA circuit connections between application programs and the application LAT port on the service node are terminated.

A session is also terminated if all open files are closed and all DDA circuit connections are disconnected.

11.6 Retrieving and Setting Terminal Characteristics

The VAXELN LAT driver supports the DDA interface procedures ELN$TTY_GET_CHARACTERISTICS and ELN$TTY_SET_CHARACTERISTICS. An application program can use a VAXELN LAT port's DDA port in a call to ELN$TTY_GET_CHARACTERISTICS to retrieve the following characteristics for that LAT port:

- Terminal type
- Speed
- Parity
- Parity type
- Display type
- Escape recognition
- Echo
- Passall
- Eight-bit
- Display type
- Character size
- Terminal synchronization
- Modem
- DDCMP

Similarly, a program can use a VAXELN LAT port's DDA port in a call to ELN$TTY_SET_CHARACTERISTICS to set the following subset of these characteristics, which includes escape recognition, echo, passall, eight-bit, display type, and terminal synchronization.

For more information about retrieving and setting terminal characteristics, see Section 14.4.5.2.

Chapter 12
System Security

The VAXELN Toolkit includes system security features that protect system resources and data from unauthorized use, examination, or modification. Since VAXELN is primarily for developing and running dedicated applications, the security features are disabled by default for programs running on a single system. You might use these features, however, to protect an application from inexperienced or malicious users.

This chapter provides an overview of the security features that the VAXELN Toolkit supplies (see Section 12.1) and describes the following:

- User names and identification codes, Section 12.2
- Authorization Service, Section 12.3
- User identities, Section 12.4
- File service security, Section 12.5

12.1 Security Features Overview

The VAXELN Toolkit provides security features that application designers must explicitly include and enable. If, for example, a VAXELN system is to be included as part of a larger network of systems, the system would normally include the security features.

Since VAXELN is not intended to provide a multiuser time-sharing environment, no protection is enforced among programs running on a single system. That is, although the VAX memory management ensures that incorrectly coded programs cannot accidentally modify the memory allocated to other programs, the VAXELN kernel and runtime services do not attempt to dictate which programs can run in kernel mode, alter priorities, stop and start program execution, or, in general, fairly distribute the resources of the single-node system.

The programs running on a system control the resources of the system. Therefore, if a VAXELN application is potentially vulnerable to inexperienced or even malicious users, you should ensure that the application and the system are protected. Also, if protection of system resources is required, users should not be allowed to run their own programs.

Many VAXELN systems are part of a larger network. Programs must protect the resources of these systems from use or abuse by other users of the network. In particular, programs that accept requests from other network nodes need to somehow determine the identity of the requestor. An example of a program with this requirement is the File Service, which needs to provide protection for the disk files that it services.

The most basic security feature of a VAXELN system, therefore, provides the capability for a program to determine the identity of a user issuing a network request. This feature is provided by an optional service called the Authorization Service. The Authorization Service maintains a data base of the users authorized to use a particular VAXELN system or network of systems. When an application program accepts a circuit connection to handle a request, the program can query the data base to determine the identity of the requestor.

Other VAXELN facilities use the Authorization Service to protect the resources and data that they control. The Network Service running on a particular node accepts incoming circuit connections only from authorized users in the Authorization Service's data base. The File Service provides read, write, and delete protection for files on disk volumes that it controls. The Authorization Service itself uses the data base to protect the data base. Likewise, application programs can use the service to protect their resources and data.

12.2 User Names and Identification Codes

Each process in a VAXELN system has an associated user name string and a user identification code (UIC). These two values are maintained by the kernel and are inherited by a process from the process or job that created it. A process can also set its own user name and UIC to desired values by calling the KER$SET_USER kernel procedure (see Section 12.4).

The UICs are integer values that provide a shorthand way of identifying a user or group of users. UICs can then be used by application programs to protect their resources. For example, the File Service stores a UIC with each file that is created. The File Service then uses the stored UIC, called the *owner UIC*, to determine whether a requestor should be allowed to access the file.

The VAXELN use of UICs is compatible with the VMS use. On VAXELN and VMS, UIC values are 32-bit longwords, partitioned into two 16-bit words. The least significant word is called the *member* number, and the most significant word is called the *group* number. UIC values are normally displayed in octal, in the format [*group-number,member-number*] — for example, [1,4], [11,32], [200,200]. The partitioning of the value into group/member fields allows groups of values to be associated with each other for protection. Also, group numbers less than or equal to octal 10 are considered part of the *system* groups. The use of UICs is explained in Section 12.5.

A process can determine its own user name and UIC by calling the KER$GET_USER kernel procedure. Since, as just described, the security features in VAXELN are based upon validating network requests, a process can also determine the user name and UIC of the process from which it has accepted a circuit connection. This capability is also provided by calling KER$GET_USER, although the port object connected in the circuit is then one of the arguments.

12.3 Authorization Service

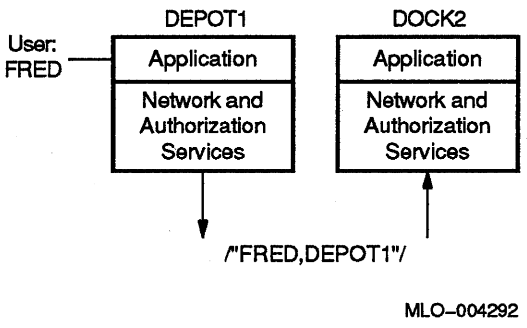
The Authorization Service is the key component of the VAXELN security facilities. It protects system resources and data by maintaining a data base of a system's authorized users and identifying users who issue network requests.

A target system can include local or network authorization services. When a system includes the network authorization services, it handles authorization for the nodes in a local area network that do not have their own service. At least one node in a local area network must include this service. If multiple nodes include the network authorization service and all nodes in the local area network use the same data base file, one target system acts as an *authorization server* and manages the data base while the other nodes serve as backups. By designating multiple authorization servers, you can preserve the application's security if the acting server shuts down.

The Authorization Service's primary task is to determine the identity of the requestor of a network connection request. The service gets the requestor's host system user name and node name and looks them up in the authorization data base. The service can also accept a specific user name and password, or *access control string*, and look them up in the data base.

Figure 12–1 and the accompanying text illustrate and explain how the service works.

Figure 12–1: Authorization Service Example


DEPOT1
DOCK2
User: FRED
Application
Network and Authorization Services
Application
Network and Authorization Services
/"FRED,DEPOT1"/
MLO–004292

In Figure 12–1, a user named FRED is executing a program on a VAXELN node named DEPOT1. FRED issues a request for a service on another node named DOCK2, so the Network Service on node DEPOT1 sends a connection request message that identifies FRED and DEPOT1 to DOCK2. The DOCK2 Network Service then sends a request to its Authorization Service to verify that user FRED on node DEPOT1 is authorized to use the services provided by node DOCK2. The Authorization Service replies to the Network Service with a Yes or No indication; if Yes, the Authorization Service returns the UIC with which the user is to be identified.

This type of authorization is termed *proxy access control*. Since FRED is authorized to use the resources of node DEPOT1, his DEPOT1 name is sent, by network proxy, to determine if he can use the resources of node DOCK2.

The other type of authorization provided by the Authorization Service is called *destination authorization*. It is used when a connection request or open file operation specifies a user name and password, or access control string, with the connection request. Destination authorization provides a means of assuming a new identity on the remote system.

Proxy access control and destination authorization are provided in a compatible manner by the VMS operating system. Other Digital operating systems support only the destination authorization provided with access control strings.

The CONNECT_CIRCUIT procedure allows you to specify a remote destination as a string by using the optional DESTINATION_NAME parameter. Like other DECnet systems, the node specification for CONNECT_CIRCUIT can include a user name and password, which can be optionally enclosed in quotes and separated from each other by a space.

To specify the remote destination by object name, use a string of one of the following forms:

'*nodenumber*::*objectname*'

'*nodenumber*"*username password*"::*objectname*'

'*nodenumber*"*username*"::*objectname*'

'*nodenumber*"[*ggg*,*mmm*] *password*"::*objectname*'

For example, the following call connects to object TESTOR on node number 3, using a user name of FRED and a password of SWIZZLE:

```
CONNECT_CIRCUIT(p, DESTINATION_NAME := '3"FRED SWIZZLE"::TESTOR');
```

To specify the remote destination by object number, use a string of one of the following forms:

'*nodenumber*::*objectnumber*'

'*nodenumber*"*username password*"::*objectnumber*'

'*nodenumber*"*username*"::*objectnumber*'

'*nodenumber*"[*ggg*,*mmm*] *password*"::*objectnumber*'

For example, the following call connects to object number 129 on node 4, using a user name of [10,150] and a password of QUAKE. This format is typically used only to connect to RSTS/E systems.

```
CONNECT_CIRCUIT(p, DESTINATION_NAME := '4"[10,150] QUAKE"::129');
```

To connect to a port in a VAXELN system, use a string of one of the following forms:

'nodename::objectname'

'nodename"username password"::objectname'

'nodename"username"::objectname'

'nodename"[ggg,mmm] password"::objectname'

'nodename::objectnumber'

'nodename"username password"::objectnumber'

'nodename"username"::objectnumber'

'nodename"[ggg,mmm] password"::objectnumber'

For example, the following call would connect to object TEST on node NODEA, using a user name of FRED and a password of ABC:

```
CONNECT_CIRCUIT(p, DESTINATION_NAME := 'NODEA"FRED ABC"::TEST');
```

Since the OPEN routine uses CONNECT_CIRCUIT to access remote files on other DECnet nodes, its FILE_NAME parameter can also include a user name and password if a node number is specified.

For example, the following call would open FILE99.DAT on node number 3, using a user name of FRED and a password of SWIZZLE:

```
OPEN(f, FILE_NAME := '3"FRED SWIZZLE"::FILE99.DAT');
```

12.3.1 Including the Authorization Service

The Authorization Service is supplied as a program image that can be included in a VAXELN system using the System Builder. To include this service in a system, do the following:

1. Select *Yes* for the **Authorization required** entry on the Network Node Characteristics Menu. When you select *Yes*, the Network

Service monitors inbound circuit connections and honors a connection only if it can authorize the user through the Authorization Service.

2. Select *Local* or *Network* for the **Authorization service** entry on the Network Node Characteristics Menu. When you select *Local*, the service is included in the system image but handles authorization for only the local target system. When you select *Network*, the service is included, and it handles authorization for any node in the local area network that does not have its own Authorization Service.

   The network Authorization Service uses VAXELN universal names. Thus, at least one system in the network must include the Name Service. Only one of the nodes can run the network Authorization Service, even if multiple nodes include the Name Service.
3. Specify an authorization data file for the **Authorization file** entry on the Network Node Characteristics Menu. The data file must exist on the same node as the Authorization Service or on a node that the service is authorized to access, for example, a node with its own local service. The default file is [0,0]AUTHORIZE.DAT on the local default disk.

When the Authorization Service starts running, it opens and reads the specified data file. If the data file is not found, the Authorization Service creates a new one. The file should be modified only by using the maintenance procedures described in Section 12.3.2.

Typically, the authorization data file is on a disk directly attached to the node running the Authorization Service. In such a case, when the file is first created by the service, it can be modified only by users running programs on the same node. That is, since the data file is empty, no remote users are authorized to access the node.

Once other users are authorized, if they have UICs in the system group, they can remotely maintain the authorization data file.

12.3.2 Authorization Service Utility Procedures

The Authorization Service provides the capability to maintain the authorization data base. Since the Authorization Service can run as a server in a local area network, it performs the maintenance functions, using messages and its own maintenance request protocol. To simplify the development of maintenance programs, the VAXELN

Toolkit includes a set of utility procedures that handle the protocol, eliminating the need for user programs to code the protocol explicitly.

The Authorization Service utility procedures are as follows:

Routine	Description
ELN$AUTH_ADD_USER	Adds a new user record to the authorization data base.
ELN$AUTH_MODIFY_USER	Modifies a user record in the authorization data base.
ELN$AUTH_REMOVE_USER	Removes a user record from the authorization data base.
ELN$AUTH_SHOW_USER	Returns the authorization data base information for the specified users.

To use the Authorization Service utility procedures, your program must be authorized with a system group UIC — that is, a UIC that is less than or equal to %X0008FFFF or [10,177777]. You must also include the $AUTHORIZE_UTILITY module in the compilation. For descriptions of these routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual.*

12.3.3 Establishing Circuits for Authorization Service Communication

An application program communicates with the Authorization Service using a VAXELN virtual circuit. The program must establish the circuit connection by creating a port and connecting that port to Authorization Service's AUTH$MAINTENANCE port. Once the connection is made, the program can call the Authorization Service utility procedures to set up and access the authorization base. The following Pascal example connects the port *authorization_port* in a circuit to the AUTH$MAINTENANCE port.

```
MODULE test_authorization;

INCLUDE $AUTHORIZE_UTILITY;

PROGRAM authorization_maintenance;
```

```
VAR
  authorization_port : PORT;
    .
    .
    .
BEGIN
    .
    .
    .
  CREATE_PORT(authorization_port);
  CONNECT_CIRCUIT(authorization_port,
                 DESTINATION_NAME := 'AUTH$MAINTENANCE');
    .
    .
    .
END;
END.
```

Once the connection between *authorization_port* and the authorization maintenance port is established, the program can call the Authorization Service utility routines, specifying *authorization_port* as an argument.

12.3.4 Adding Users to the Authorization Data Base

To add new user records to the authorization data base, call the ELN$AUTH_ADD_USER procedure. A call to ELN$AUTH_ADD_USER must specify the port connected in a circuit to the AUTH$MAINTENANCE port, a user name, a node name, a password, a user identification code, and user data.

The user name can specify a name, a null string, or the reserved word $ANY. If you specify $ANY, any user from the specified node that does not match one the explicit user names is authorized with the specified user identification code.

The node name argument specifies the name or number of the node on which the new user is authorized. You can specify a node name, node number, a null string, or the reserved name $ANY.

- If you specify a node name or number, the data base record represents a proxy access control, and the Authorization Service does not use the password.
- If you specify a null string, the data base record represents a destination authorization.

- If you specify $ANY, any user with the specified name from any node that does not match one of the explicit node names is authorized with the specified user identification code.

You can specify a password or null string for the password argument. If you add a destination authorization record to the data base (that is, if you specify a null string for the node name), the Authorization Service stores the password in the record. The Authorization Service stores all passwords in scrambled form so they cannot be read once they are stored.

You must specify a user identification code for each new user.

A user data argument lets you store unmodified user-defined data in a user record. If you do not need to store such data, you can specify a null string.

The following section of Pascal code adds a new record to the authorization data base for user FRED:

```
VAR
  authorization_port : PORT;
  node               : AUTH$NODENAME;
  user               : AUTH$USERNAME;
  uic                : INTEGER;
  node               : AUTH$NODENAME;
BEGIN
   .
   .
   .
  CREATE_PORT(authorization_port);
  CONNECT_CIRCUIT(authorization_port,
                 DESTINATION_NAME := 'AUTH$MAINTENANCE');
   .
   .
   .
  user := 'FRED';
  node := 'DEPOT1';
  uic := %X00010002;
  ELN$AUTH_ADD_USER(CIRCUIT := authorization_port,
                    USERNAME := user,
                    NODENAME := node,
                    PASSWORD := '',
                    UIC := uic,
                    USERDATA := '');
   .
   .
   .
END;
END.
```

The call to ELN$AUTH_ADD_USER adds user FRED to the data base with authorization on node DEPOT1 and the user identification code %X00010002. Since a node name is specified, the record represents a proxy access control. Thus, a null string is specified for the password argument. The call also omits user data by specifying a null string for that argument.

For information about establishing a circuit with the AUTH$MAINTENANCE port, see Section 12.3.3. For a description of the ELN$AUTH_ADD_USER procedure, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

12.3.5 Modifying Records in the Authorization Data Base

To modify user records in the authorization data base, call the ELN$AUTH_MODIFY_USER procedure. A call to ELN$AUTH_MODIFY_USER must specify the port connected in a circuit to the AUTH$MAINTENANCE port, a user name, and a node name. You must also specify a new fields argument that identifies the record fields that you intend to modify and new values for the user name, node name, password, user identification code, and user data, as appropriate.

The user name and node name arguments specify the name of the user whose record is to be modified and the name or number of the node on which that user is authorized.

The new fields argument identifies the user record fields you will be changing: user name, node name, password, user identification code, or user data. You can modify fields without changing other fields in the record. To change a record, you must indicate the appropriate fields in the value you specify for this argument and specify values for the fields you are changing. The call to ELN$AUTH_MODIFY_USER in the following section of code changes the node for user FRED:

```
VAR
  authorization_port : PORT;
  node               : AUTH$NODENAME;
  user               : AUTH$USERNAME;
  uic                : INTEGER;
  new_node           : AUTH$NODENAME;
BEGIN
   .
   .
   .
  CREATE_PORT(authorization_port);
  CONNECT_CIRCUIT(authorization_port,
                  DESTINATION_NAME := 'AUTH$MAINTENANCE');
   .
   .
   .
  user := 'FRED';
  node := 'DEPOT1';

  new_node := 'DEPOT2';
  uic := %X00010002;
  ELN$AUTH_MODIFY_USER(CIRCUIT := authorization_port,
                       USERNAME := user,
                       NODENAME := node,
                       NEW_FIELDS := [AUTH$NODENAME_FIELD],
                       NEW_USERNAME := '',
                       NEW_NODENAME := new_node,
                       NEW_PASSWORD := '',
                       NEW_UIC := '',
                       NEW_USERDATA := '');
   .
   .
   .
END;
END.
```

You can specify a new user name or the reserved name $ANY. If you specify $ANY, any user from the specified node that does not match one of the explicit user names is authorized with the specified user identification code. If you modify the user name, you must reset the password.

The value for the new node name argument can be a node name, a node number, a null string, or the reserved name $ANY. If you specify a node name of number, the data base record represents a proxy access control and the Authorization Service does not use the password. If you specify a null string, the data base record represents a destination authorization. If you specify $ANY, any user with the specified name from any node that does not match one of the explicit node names is authorized with the specified user identification code.

The hashing algorithm that the Authorization Service uses for passwords includes the user name. Thus, you must reset the password if you modify the user name.

You can reset a password to another password or to a null string. If you add a destination authorization record to the data base (that is, if you specify a null string for the node name), the Authorization Service stores the password with the record. The Authorization Service stores all passwords in scrambled form so they cannot be read once they are stored.

To establish a circuit with the AUTH$MAINTENANCE port, see Section 12.3.3. For a description of the ELN$AUTH_MODIFY_USER procedure, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

12.3.6 Removing User Records from the Authorization Data Base

To remove user records from the authorization data base, call the ELN$AUTH_REMOVE_USER procedure. A call to ELN$AUTH_REMOVE_USER must specify the port connected in a circuit to the AUTH$MAINTENANCE port, a user name, and a node name. The user name identifies the user record to be removed. The node name argument specifies the name or number of the node on which the user is no longer authorized. The call to ELN$AUTH_REMOVE_USER in the following section of Pascal code removes user FRED from the data base:

```
VAR
  authorization_port : PORT;
  node               : AUTH$NODENAME;
  user               : AUTH$USERNAME;
  uic                : INTEGER;
  node               : AUTH$NODENAME;
BEGIN
    .
    .
    .
  CREATE_PORT(authorization_port);
```

```
    CONNECT_CIRCUIT(authorization_port,
                    DESTINATION_NAME := 'AUTH$MAINTENANCE');
      .
      .
      .
    user := 'FRED';
    node := 'DEPOT1';
    uic := %X00010002;
    ELN$AUTH_REMOVE_USER(CIRCUIT := authorization_port,
                      USERNAME := user,
                      NODENAME := node);
      .
      .
      .
END;
END.
```

To establish a circuit with the AUTH$MAINTENANCE port, see Section 12.3.3. For a description of the ELN$AUTH_REMOVE_USER procedure, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

12.3.7 Retrieving Authorization Data Base Information

To retrieve information from the authorization data base, call the ELN$AUTH_SHOW_USER procedure. A call to ELN$AUTH_SHOW_USER must specify the port connected in a circuit to the AUTH$MAINTENANCE port, a user name, a node name, and the name of a show user procedure.

The user name specifies the name of the user for which data base information is to be accessed. If you specify the string '*', the procedure returns all the records in the data base.

The node name argument specifies the name or number of the node on which the user is authorized. You must specify a nonnull string if the proxy information for the specified users is requested, in which case the Authorization Service returns the proxy information.

The show user routine is a user-defined routine that the ELN$AUTH_SHOW_USER procedure invokes. The procedure invokes your routine only if it finds the specified user entry in the authorization data base. If you specify the string '*' for the user name, ELN$AUTH_SHOW_USER calls your routine once for each record in the data base.

The call to ELN$AUTH_SHOW_USERS in the following section of Pascal code instructs the Authorization Service to invoke the procedure *show_all_users* for all records in the authorization data base:

```
VAR
  authorization_port : PORT;
  uic                : INTEGER;
  node               : AUTH$NODENAME;
BEGIN
    .
    .
    .
  CREATE_PORT(authorization_port);
  CONNECT_CIRCUIT(authorization_port,
                  DESTINATION_NAME := 'AUTH$MAINTENANCE');
    .
    .
    .
  node := 'DEPOT1';
  uic := %X00010002;
  ELN$AUTH_SHOW_USER(CIRCUIT := authorization_port,
                     USERNAME := '*',
                     NODENAME := node,
                     SHOW_USER := show_all_users);
    .
    .
    .
END.

PROCEDURE show_all_users OF TYPE AUTH$SHOW_USER_ROUTINE;

BEGIN
  WRITELN('User name =', username^);
  WRITELN('Node name =', nodename^);
  WRITELN('UIC =', uic^);
END;
```

For information about establishing a circuit with the AUTH$MAINTENANCE port, see Section 12.3.3. For a description of the ELN$AUTH_SHOW_USER procedure, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

12.4 User Identities

The Network Service ensures that inbound connection requests are from authorized users. However, application programs that accept such requests should use calls to KER$GET_USER to query the user's identity and use the information to protect program resources.

Each process in a VAXELN system has a user identity that consists of a user name and a UIC. The user name can be a string of up to 20 characters, and the UIC is an integer. Using the KER$GET_USER and KER$SET_USER procedures, you can retrieve and set these values for a calling process.

The KER$GET_USER procedure returns the user identity of one of the following:

- The calling process
- A process connected in a circuit to a port owned by a process in the current job

To retrieve the user identity of a process whose port is connected to the calling process, you must specify that port in an optional circuit port argument. The port that you specify must be connected in a circuit that was established as follows:

- The process in the current job whose port is to be specified in the call to KER$GET_USER initiated the connection with a call to the CONNECT_CIRCUIT procedure.
- The process calling KER$GET_USER accepted the connection with a call to the ACCEPT_CIRCUIT procedure.

If the appropriate circuit connection is established, the kernel returns the user name and UIC associated with the process that initiated the connection to optional user name and UIC arguments. If the appropriate circuit connection is not established, KER$GET_USER returns invalid user information.

If the process that initiates the circuit connection is a remote process, you should ensure that the Authorization Service is built into your system. That is, you should select *Yes* for the **Authorization required** entry and *Network* for the **Authorization service** entry on the System Builder's Network Node Characteristics Menu. If you do not build the Authorization Service into the system, KER$GET_USER returns 0 to the UIC argument.

The Network Service ensures that inbound connection requests are from authorized users. However, application programs that accept such requests must use calls to KER$GET_USER to query the user's identity and use the information to protect program resources. The following example accepts an inbound connection request and checks that it is from a user in a system group, less than or equal to octal 10:

```
VAR
  np, p: PORT;
  username: VARYING_STRING(20);
  uic: INTEGER;
    .
    .
    .
  ACCEPT_CIRCUIT(np, CONNECT := p);
  KER$GET_USER(CIRCUIT := p,
              USERNAME := username,
              UIC := uic);
  IF (uic div %X10000) > %O10 THEN
    DISCONNECT_CIRCUIT(p)
  ELSE
    .
    .
    .
```

To set the user identity of the current process, call the KER$SET_USER procedure. Specify the user name and UIC that are to be associated with the process.

The UIC that the KER$SET_USER procedure sets is valid only on the local system. When you specify a remote destination port in a call to CONNECT_CIRCUIT, only the calling process's user name is sent to the destination system. The Authorization Service on the destination system adds that name to the authorization data base and associates the name with a new UIC. Calls to KER$GET_USER on the destination system then return the user name and new UIC as they are defined on that system.

If you include an access control string in the remote destination argument that you specify in a call to CONNECT_CIRCUIT, the user name specified in the argument is sent to the remote system rather than the user name set with KER$SET_USER. The argument in the following call to CONNECT_CIRCUIT authorizes the user name FRED and password ABC on the remote system NODEA:

```
CONNECT_CIRCUIT(DESTINATION_NAME := 'NODEA"FRED ABC"::TEST');
```

Calls to KER$GET_USER on the destination system get the UIC authorized by the destination system.

You can use the KER$SET_USER procedure to authorize a process's access requests to a remote system. Consider the following entries in the Authorization Service data base on node DOCK2.

Authorization Type	Host Node	UIC	Password
Proxy access control	DEPOT1	[1,2]	
Destination authorization	SAM	[1,3]	NOODLE

The first entry is a proxy access control because it includes a host node name. The second entry is a destination authorization because it includes a password instead of a node name.

Suppose program A on node DEPOT1 executes the following:

```
KER$SET_USER(username := 'FRED');
CONNECT_CIRCUIT(destination_name := 'DOCK2::TESTOR');
```

When program B executes the following code on node DOCK2, the Authorization Service on node DOCK2 uses the proxy access control entry to authorize the remote user:

```
CREATE_NAME(p, 'TESTOR');
ACCEPT_CIRCUIT(p);
KER$GET_USER(CIRCUIT := p,
             USERNAME := partner_user,
             UIC := partner_uic);
```

Program B receives a user name value of FRED in variable *partner_user* and a UIC value of [1,2] (%X00010002) in variable *partner_uic*.

Suppose, instead, that program A on node DEPOT1 executes the following:

```
CONNECT_CIRCUIT(DESTINATION_NAME := 'DOCK2"SAM NOODLE"::TESTOR');
```

When Program B executes the following code on node DOCK2, the Authorization Service on node DOCK2 uses the destination authorization entry to authorize the remote user.

```
CREATE_NAME(p, 'TESTOR');
ACCEPT_CIRCUIT(p);
KER$GET_USER(CIRCUIT := p,
             USERNAME := partner_user,
             UIC := partner_uic);
```

Program B receives a user name value of SAM in variable *partner_user* and a UIC value of [1,3] (%X00010003) in variable *partner_uic*.

12.5 File Service Security

The File Service uses the VAXELN features explained in this chapter to protect the disk volumes and files that the File Service manages. Since the File Service uses the Files–11 on-disk structure, it uses the standard Files–11 protection facilities. Those facilities are compatible with the VMS operating sysetm.

The standard Files–11 protection facilities are as follows:

- When a new file is created, one of its attributes is the primary UIC of the user requesting the creation. This UIC is called the *owner UIC* of the file. If the File Service is unable to determine the UIC of the user creating a new file (for example, no Authorization Service is available) the file owner UIC is set to the UIC of the disk volume owner.
- A new file also gets, as one of its attributes, a protection *mask* that describes how the File Service protects the file from the following categories of users:

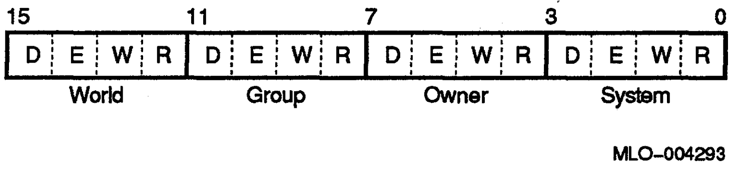
Category	Description
System	Users with UICs with a group number less than or equal to 8
Owner	Users with UICs that match the owner UIC
Group	Users with UICs with a group number that matches the owner UIC's group number
World	Users with UICs in none of the previous categories

The protection mask is a 16-bit word composed of four fields. Each of the four fields corresponds to one of the four categories of users. Each of the four fields consists of 1-bit indicators that specify the access allowed to the category: read, write, execute, and delete.

Figure 12–2 shows the protection mask.

If a bit is set in a category's field, users in that category are denied the corresponding access. For example, if bit 1 is set, then system users are denied write access.

Figure 12–2: Protection Mask



The Pascal programmer can specify the protection mask fields defined by the $FILE_UTILITY module. The C programmer typically specifies unsigned octal values. (For compatibility with UNIX systems, the C **creat** and **chmod** functions do not use the same format for the protection mask.)

- The owner and protection for a new file can be specified as parameters to the Pascal OPEN procedure and the C **creat** function. The protection for an existing file can be changed by using the Pascal PROTECT_FILE procedure and the C **chmod** and **chown** functions. If a new file is created and no protection mask is specified, the File Service sets the protection to the disk volume's default file protection.
- The owner UIC and protection mask for a new disk volume can be specified as a parameter to the ELN$INIT_VOLUME procedure.
- The default protection mask for files on a new disk volume can be specified as a parameter to the ELN$INIT_VOLUME procedure.
- If the File Service is unable to determine the UIC of a user requesting access to a file (for example, no Authorization Service is available) it allows unprotected access by the user. (See the description of the *Authorization required* Network Node Characteristic in Section 12.3.1, for a means of preventing this unprotected access.)

Chapter 13

File Service

The File Service is a set of services provided by the disk and tape drivers in a system that allows programs to perform file-oriented I/O on disks and tapes. The File Service is not used for terminal or printer I/O.

The File Service consists of a disk File Service and a tape File Service:

- The disk File Service provides Files–11 On-Disk Structure Level 2 file services. The disk File Service is compatible with the VMS Version 4.4 file system and with the RMS–32 system routines.
- The tape File Service is based on Version 3 of the ANSI standard for magnetic tapes. The tape File Service provides users with a convenient means of transporting files to and from VMS systems, since it is compatible with the VMS Version 4.4 file system.

For disk and tape devices supported by Digital, the File Service is already linked with the VAXELN drivers. If you are writing your own disk or tape drivers that will use the File Service, the appropriate shareable image must be linked, as explained in Section 13.12.

When several VAXELN systems are running on nodes in a local area network, only one node needs to have disk or magnetic tape hardware. An appropriate hardware configuration, running a system containing the File Service, thus can act as a *file server* for other jobs on the same node or on other nodes, handling all file storage and retrieval for the local area network.

With disks, for example, programs can identify files, regardless of their network locations, by using file specifications that give the File Service volume name for the storage device; node specifications are needed only when you use a file that is stored on an operating system other than a VAXELN system. Systems that support file access from remote nodes also include a separate job, the File Access Listener, to handle connection requests between nodes.

This chapter discusses the following:

- Device specifications, Section 13.1
- Volume names, Section 13.2
- File specifications, Section 13.3
- Mount procedure for multiple volumes with identical volume labels, Section 13.4
- Use of the DISK$DEFAULT_VOLUME device name, Section 13.5
- File Access Listener, Section 13.6
- Use of file service volumes from VMS, Section 13.7
- File service operations, Section 13.8
- File utility procedures, Section 13.9
- Disk utility procedures, Section 13.10
- Tape utility procedures, Section 13.11
- File Service interface for disk and tape drivers, Section 13.12
- Data Access Protocol, Section 13.13

13.1 Device Specifications

You must provide descriptions of the devices to be used by a VAXELN system when you build the system by editing the System Builder's Device Description Menu, as explained in the *VAXELN Development Utilities Guide*.

Each device name identifies a specific unit on a specific controller. Typically, the controller is specified by a letter and the unit by a number. For example, the device specification *DQA1* identifies controller A, unit 1, for an RB02 or RB80 disk attached to the Integrated Disk Controller of a VAX–11/730 processor.

Table 13–1 lists the storage device types used in VAXELN programming.

Table 13–1: Storage Device Types

Device Type	Meaning
DQ	VAX–11/730 Integrated Disk Controller (RB02 cartridge disks and RB80 fixed disks)
DD	TU58 cartridge drive in VAX console
DU	UDA50 UNIBUS interface to Storage Interconnect (SI) disks, RQDXn (MicroVAX) interface to RXnn diskettes and RDnn Winchester disks, RC25 fixed and removable disks, KDA50 Q-bus RAxx-series disk adapter
BD	KDB50 BI RAxx-series disk adapter
MU	TK50 streaming cartridge tape drive, TK70 streaming cartridge tape drive, or TU81 reel tape system

The device types in Table 13–1 are conventional names for these devices; you can use any names that you like, provided the usage is consistent in the System Builder and in user programs.

13.2 Volume Names

After you enter the device specifications for the drives used by the File Service, you can supply volume names, or volume labels, for disks or tapes that are to be mounted by the service when the system is started. Volume names are specified on the System Builder's Edit System Characteristics Menu (see the *VAXELN Development Utilities Guide*).

The volume name is paired with a device specification; the following System Builder menu argument establishes two such pairings:

```
"DUA1 TEST1", "DUA0 TEST2"
```

TEST1 is established as the volume to be mounted on drive DUA1 and TEST2 as the volume for DUA0. The first volume mounted in the system becomes the *default volume* for the File Service. That is, any file specification that lacks a volume name or device name refers to this volume. In systems with a single disk controller, the first volume specified in the list (here, TEST1) will be the first volume mounted.

In systems with multiple disk controllers, volumes are mounted in the order in which the disk driver jobs controlling their disks are created and initialized at system start-up. The order in which the driver jobs are created is determined by their job priority; the driver with the highest job priority is created first. If two or more drivers with the same priority exist, their jobs are created in the order in which they appear in the System Builder's program list, as shown in a full System Builder map. In this case, the default volume will be the first volume in the disk/volume name list that is associated with the first driver job to be initialized.

The controller name (here, DUA) is also supplied as an argument to the driver. The *VAXELN Development Utilities Guide* explains how the controller device is described to the System Builder and how the appropriate driver is built into the system.

The specified volumes are mounted automatically if the VAXELN system is built with the File Service. If no volume name is supplied for a specified device, the File Service tries to mount the volume that exists in the drive. If the specified volume name is not the same as the name specified when the volume was initialized, the File Service mounts the volume anyway and displays an informational message on the target machine's console terminal. However, the specified name is overridden by the volume name of the volume mounted in the drive. For example, if you specify the volume name TEST1, but the volume in the drive was initialized with the name TEST3, you would have to refer to the mounted volume by the name DISK$TEST3, not DISK$TEST1.

If no argument is supplied for a drive, no volume is mounted initially by the File Service, but a volume can be mounted dynamically with the ELN$MOUNT_VOLUME procedure or with the ELN$MOUNT_TAPE_VOLUME procedure, as appropriate. If the drive is a disk, it can also be used directly (for nonfile, or *logical*, I/O) by opening it for logical I/O with the Pascal OPEN procedure or corresponding C open functions. (Logical I/O treats the volume as if it were a single large file; logical I/O is explained in Chapter 14.)

NOTE

If you attempt to mount a VMS disk volume that was improperly dismounted — for exmaple, if the VMS system crashed — the File Service prints a warning message on the target machine's console. The volume should be remounted on VMS, which rebuilds it; then it can be mounted on the VAXELN system. You can successfully mount a VAXELN or VMS tape volume that was improperly dismounted and can read all of its files. If the tape structure was corrupted — for example, by a crash of the VAXELN system when a file was being written — additional files cannot be written to it.

13.3 File Specifications

When used in programs, such as in a call to the Pascal OPEN procedure, file specifications with volume labels are interpreted by the File Service as referring to a particular mounted disk or tape on the target machine. The format for specifying a volume label is as follows:

DISK$*name* or TAPE$*name*

If you supplied volume names with the System Builder's Edit System Characteristics Menu, *name* must match a volume name you defined. If you did not define volume names through the System Builder or if a different volume is in the drive, *name* must match the actual volume name.

The first time a volume is mounted — whether by the File Service or with the ELN$MOUNT_VOLUME or ELN$MOUNT_TAPE_VOLUME procedure — its DISK$ or TAPE$ name is established as a *universal* name by the File Service and uniquely identifies the volume to local area network nodes.

If another process in the application mounts a volume with the same volume name, the volume's DISK$ or TAPE$ name is established as a *local* name for that process's node. The use of local names allows, for example, a VAXELN system to initialize, mount, and write duplicate copies of a volume, all with the same volume name.

To illustrate the use of volume labels in file specifications, suppose that the following volume name definitions are entered on the Edit System Characteristics Menu:

```
"DQA1 TEST1", "DQA0 TEST2"
```

The volume specifications DISK$TEST1 and DISK$TEST2 in programs now refer to disks mounted on drives DQA1 and DQA0, respectively. Furthermore, DISK$TEST1 (DQA1) is the default disk volume; if no volume or device is specified in a file specification, the File Service refers to the specified directory, file name, and so forth on TEST1.

For example, the following Pascal procedure call creates a file on DISK$TEST2:

```
OPEN(myfile, FILE_NAME := 'DISK$TEST2:[data]analog.dat');
```

The corresponding example in C is as follows:

```
#include stdio
FILE *file_ptr
file_ptr=fopen("DISK$TEST2:[data]analog.dat","r");
```

Here, the file *analog.dat* in directory *data* is created and is represented by the program variable *myfile.*

NOTE

> If a volume is mounted, you can also refer to it with an explicit device name. For example, the following OPEN statement refers to the disk volume in drive DUA0:
>
> ```
> OPEN(f,FILE_NAME := 'DUA0:[TEST]TEST.DAT')
> ```
>
> The corresponding example in C is as follows:
>
> ```
> #include stdio
> FILE *file_ptr
> file_ptr=fopen("DUA0:[TEST]TEST.DAT","r");
> ```
>
> Device names are local to their network node.

If you access a file on a remote node by using file specification syntax other than VMS syntax, you must enclose the specification in quotation marks. For example, the following file specification file reference contains a question mark, which is not valid under VMS. Therefore, you must enclose the file reference in quotation marks:

```
2.9::"DATA?.DAT"
```

13.4 Procedure for Mounting Multiple Volumes with Identical Volume Labels

You can mount multiple disk volumes that have the same volume label. However, the presence or absence of the Network Service and Name Service can affect such an operation.

If the system does not include the Network Service and Name Service and you try to mount a volume that has the same volume name as an already mounted volume, the mount fails.

If the system includes the Network Service and Name Service (the System Builder defaults), and you try to mount a volume that includes the volume label of an already mounted volume, several outcomes are possible.

Consider a situation in which two disk volumes are to be mounted in the system. The first volume has the name TEST and will be mounted in the physical device DUA0. The second volume, also named TEST, will be mounted in the physical device DUA1.

Suppose a mount request is made specifying the device DUA0. When you mount the volume, the following names are created:

- DUA0. This is the first name created; it is placed in the local name table. If this name cannot be created, the mount operation fails.
- DISK$DEFAULT_VOLUME. This is the second name created; it is also placed in the local name table. If this name cannot be created, the system assumes that another disk volume is to be used as the default volume and has already been mounted in the system.
- DISK$TEST. The system tries to create this name in the universal name table. If this fails, the system tries to create the name in the local name table. For example, if the universal name DISK$TEST does not exist in the network, the name DISK$TEST is created in the universal name table.

You can use these names to access the mounted volume.

If you try to mount another volume that has the same volume label, assuming that the mount request is made to the DUA1 device, the following events occur:

1. The name DUA1 is created in the local name table.
2. The system's attempt to create the name DISK$DEFAULT_ VOLUME fails because the name already exists.
3. The system's attempt to create the name DISK$TEST in the universal name table fails, so the name is created in the local name table.

When the second mount operation is completed, the system knows the following names:

Name	Name Table	Device Represented
DUA0	Local	DUA0
DISK$TEST	Universal	DUA0
DISK$DEFAULT_VOLUME	Local	DUA0
DUA1	Local	DUA1
DISK$TEST	Local	DUA1

The preceding operations have the following consequences for use of volume labels:

- If you try to access a file by using the device specification DISK$TEST:, you get access to device DUA1 because local names are used first.
- If you call the ELN$DIRECTORY_OPEN procedure, it returns a volume name string that does not correctly refer to the volume accessed. For example, if you specify DUA0:[000000]*.*;* for the *search_name* argument in a call to ELN$DIRECTORY_OPEN, the procedure returns DISK$TEST as the volume name. If you include this volume name in the file name you specify with a file utility routine, such as ELN$DELETE_FILE, you access the volume DUA1, not DUA0.

  Therefore, you should use the device that you specify for the *search_name* argument, not the device that you specify for *volume_name*, in subsequent file operations.

13.5 DISK$DEFAULT_VOLUME Device Name

The name DISK$DEFAULT_VOLUME is used as the device name when a file is accessed without the device name specification. When you mount a volume, the system tries to create the name DISK$DEFAULT_VOLUME. If the name does not exist, it is created. If another volume is already mounted, the name already exists, and the second attempt to create the name fails.

If you dismount the volume that corresponds to DISK$DEFAULT_VOLUME, the name is deleted. A subsequent attempt to access files on DISK$DEFAULT_VOLUME fails. The next mount request recreates the name DISK$DEFAULT_VOLUME. The system does not check to see whether other volumes are already mounted in the system. The following sequence of ECL commands illustrates this behavior:

```
ECL>  MOUNT DUA0:
ECL>  DIRECTORY DISK$DEFAULT_VOLUME:

Directory DISK$VOLUME1:[000000]

000000.DIR       BACKUP.SYS       BADBLK.SYS       BITMAP.SYS
CONTIN.SYS       CORMIG.SYS       INDEXF.SYS       VOLSET.SYS
VOLUME1.DAT

Total of 9 files.

ECL>  MOUNT DUA1:
ECL>  DISMOUNT DUA0:
ECL>  DIRECTORY DISK$DEFAULT_VOLUME:
$DIRECT-E-OPENIN, error opening DISK$DEFAULT_VOLUME:[000000]*.*;
* as input
-ELN-F-DEV, error in device name or inappropriate device type
for operation

ECL>  DISMOUNT DUA1:
ECL>  MOUNT DUA1:
ECL>  DIRECTORY DISK$DEFAULT_VOLUME:

Directory DISK$VOLUME2:[000000]

000000.DIR       BACKUP.SYS       BADBLK.SYS       BITMAP.SYS
CONTIN.SYS       CORMIG.SYS       INDEXF.SYS       VOLSET.SYS
VOLUME2.DAT      VOLUME2.DAT

Total of 10 files.
```

13.6 File Access Listener

The file access listener (FAL) is built into VAXELN systems that support file access from remote nodes. You include the FAL in a VAXELN system by selecting **Yes** for the **File access listener** entry on the System Builder's Network Node Characteristics Menu, as explained in the *VAXELN Development Utilities Guide*.

The FAL handles connection requests, such as file openings, that involve different network nodes, including incoming requests from VMS nodes. Accordingly, the inclusion of the FAL in a VAXELN system also presumes that the Network Service is present.

Inclusion of the FAL does not necessarily mean that the File Service must be present. For example, a VAXELN network could use a system that includes a line printer, a line printer device driver, and the FAL as a *print server*. The FAL would accept I/O connection requests directed at the printer and establish the connection with the printer driver's message ports.

13.7 File Service Volumes from VMS

The File Service uses the same on-disk and tape file structure as the VMS operating system, and supports most VMS file-handling operations, such as COPY, DIFFERENCES, DIRECTORY, EDIT, and so forth.

For example, assume that node MILDEW is a VAX system with two disks, DQA0 and DQA1, and that MILDEW is running a VAXELN system with the File Service, Network Service, and FAL. Assume also that the following volume name definitions were entered on the System Builder's Edit System Characteristics Menu:

```
"DQA1 TEST1", "DQA0 TEST2"
```

You can copy a file from MILDEW with the following VMS command, which refers by default to DQA1 on MILDEW:

```
$  COPY MILDEW::[directory]filename.type  *.*
```

Or, you can copy with the following command, which refers to DQA0 on MILDEW:

```
$  COPY MILDEW::DISK$TEST2: [directory]filename.type  *.*
```

You can also use the device specification directly, as in the following command:

```
$  COPY MILDEW::DQA0:[directory] filename.type  *.*
```

13.8 File Service Operations

The File Service performs the following disk file and record I/O operations:

- Creating a new file or opening an existing file — for example, using the Pascal OPEN procedure or the C **open** function
- Retrieving information from the file — for example, using the Pascal READLN procedure or the C **gets** function
- Adding information to the file — for example, using the Pascal WRITELN procedure or the C **puts** function
- Closing a file — for example, using the Pascal CLOSE procedure or the C **close** function

When you are familiar with Pascal or C I/O, all you need to know about the File Service is how to initialize, mount, and dismount volumes and how to create directories.

The call formats and detailed argument descriptions for all file I/O routines, as well as for the file utility, disk utility, and tape utility procedures summarized in the following sections, are contained in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

13.9 File Utility Procedures

The file utility procedures provided by the File Service are summarized in this section. To use these procedures, you must include the $FILE_UTILITY module appropriate to the language you are using in the compilation of your program. (For more information, see the *VAXELN Development Utilities Guide*.

NOTE

> The VAXELN File Service supports all of the file utility procedures for disk and tape volumes. However, the ELN$CREATE_DIRECTORY, ELN$DELETE_FILE, ELN$RENAME_FILE, and ELN$PROTECT_FILE procedures are invalid for tapes. An error message is returned if you attempt to apply them to tape volumes.

13.9.1 ELN$COPY_FILE Procedure

The ELN$COPY_FILE procedure makes a duplicate of a specified file. A string of 1 to 255 characters gives the file specification of the source file. A second string of 1 to 255 characters gives the file specification of the destination file. You can supply file specification defaults for both the source and destination files.

Optional parameters return the resultant file name strings of both files, the mode (block or record) and the number of blocks or records copied. If an error exists in one of the files, an optional Boolean expression is returned, indicating which file contains the error.

NOTE

> The ELN$COPY_FILE procedure provides the only means of creating an ISAM or RELATIVE organization file in a VAXELN system, by copying an existing file of the organization.

13.9.2 ELN$CREATE_DIRECTORY Procedure

The ELN$CREATE_DIRECTORY procedure creates a directory on the specified file service disk volume. This procedure is invalid for tape volumes.

A string of 1 to 255 characters gives the file specification for the directory or subfile directory to be created. You can supply a file specification default for the directory. A file owner user identification code (UIC) can also be specified. An optional parameter returns the resultant file name string of the directory file created.

For example, the following Pascal command creates the directory DATA.DIR in the master file directory of the volume:

```
CREATE_DIRECTORY('DISK$TEST:[DATA]');
```

The directory must be created on a VAXELN disk volume; the procedure cannot create a directory on a volume that is not part of a VAXELN system. Also, the procedure creates only the last directory in the specification; any intermediate directories must already exist.

When you create a directory and do not specify an owner, the directory is assigned the same owner as the directory under which it is created. Furthermore, all files created under a directory are assigned the owner of that directory's owner, unless you specify otherwise in the call to OPEN. Unless you desire the default owner, you should specify an owner when creating directories and files. (Specifying 0 has the same effect as not specifying a value.)

13.9.3 ELN$DELETE_FILE Procedure

The ELN$DELETE_FILE procedure deletes a file from a mounted disk volume. This procedure is invalid for tape volumes.

A string of 1 to 255 characters gives the file specification, with either an explicit version number or a semicolon or period to indicate the most recent version. For example, *test.dat;23* designates version *23* is to be deleted; *test.dat;* and *test.dat.* designate the most recent version of the file. You can supply a file specification default for the file to be deleted. Another optional parameter stores the resultant file name string of the deleted file.

13.9.4 ELN$DIRECTORY_CLOSE Procedure

The ELN$DIRECTORY_CLOSE procedure closes a directory on a mounted disk volume. A variable supplies a pointer to the directory file variable.

The preferred method for obtaining directory listings is to use the ELN$DIRECTORY_OPEN procedure to open the directory, then loop, calling the ELN$DIRECTORY_LIST procedure until no more files are found. You should call ELN$DIRECTORY_CLOSE only if you do not want the program to continue the directory list loop until all the files are exhausted.

13.9.5 ELN$DIRECTORY_LIST Procedure

The ELN$DIRECTORY_LIST procedure obtains the next file name from a mounted disk directory. A variable supplies a pointer to the directory file. If more than one directory is traversed by ELN$DIRECTORY_LIST, the directory name will change. An optional variable supplies a pointer to a file attributes record that receives the file's attributes; if you specify this argument, you must supply the pointer that was returned by a previous call to ELN$DIRECTORY_OPEN.

13.9.6 ELN$DIRECTORY_OPEN Procedure

The ELN$DIRECTORY_OPEN procedure opens a directory on a mounted disk volume in preparation for a ELN$DIRECTORY_LIST operation, returning the volume name and directory name if the procedure is successful. A variable supplies a pointer to the directory file variable.

A string of 1 to 255 characters supplies the file specification of a directory for which to search. The general form of the character string is as follows:

node::*disk*:[*directory*]*filename.type;version*

The file name, type, and version can use the wildcard characters, percent sign (%) and askerisk (*), as in VMS file specifications. The % character matches any single character in the corresponding position; the * character matches any character or string in the indicated positions, including null strings.

For example, the following string matches any specification with a file name of at least four characters, the last being C and the fourth-from-last being A, and any file type or version.

```
DISK$TEST:[testdata]*A%%C.*;*
```

Wildcards are not allowed in volume names or, for VAXELN disks, in directory specifications.

If the directory is not on a VAXELN disk — for example, it is serviced by a VMS system — the asterisk (*), percent sign (%), and ellipsis (. . .) can be used in the directory specification. The ellipsis following a directory name matches all subdirectories contained in and including the named directory.

In addition, an optional string of 1 to 64 characters receives the resultant node specification or server process port name. An optional variable receives a pointer to the file attributes record allocated by ELN$DIRECTORY_OPEN; you can use this pointer in subsequent calls to ELN$DIRECTORY_LIST to receive a file's attributes.

13.9.7 ELN$PROTECT_FILE Procedure

The ELN$PROTECT_FILE procedure changes the protection of a disk file. This procedure is invalid for tape volumes.

A string of 1 to 255 characters gives the file specification. The procedure sets the file ownership user identification code (UIC), the protection code, or both for the specified file. You can supply a file specification default for the file. Another optional parameter returns the resultant file name string of the file.

13.9.8 ELN$RENAME_FILE Procedure

The ELN$RENAME_FILE procedure renames a disk file. This procedure is invalid for tape volumes.

A string of 1 to 255 characters gives the current file specification; no wildcard characters are permitted. (To rename several related files, use ELN$DIRECTORY_LIST to find them and ELN$RENAME_FILE to rename each one.) A second string of 1 to 255 characters gives the new file specification. You can supply file specification defaults for both the current file and the new file. Optional parameters return the resultant file name strings of both files.

The new volume name must be the same as the old one; that is, if the old specification includes a volume name, the new one must supply the same name or no name. Any parts of the current specification that are not supplied in this argument are obtained from the old file name.

13.9.9 ELN$SET_DEFAULT_FILESPEC Procedure

The ELN$SET_DEFAULT_FILESPEC procedure establishes a default file specification to be used within the current job. The default is applied to procedures that take a file specification as an input parameter. For example, if you set the default to DISK$YAHOO:[TEST]FILE1.DAT and if you call the OPEN procedure with the file name TEST, the resulting file reference for the OPEN is DISK$YAHOO:[TEST]TEST.DAT.

A string of 1 to 255 characters gives the default file specification for the current job. The string replaces the previous default specification each time you call ELN$SET_DEFAULT_FILESPEC within a job. You cannot use wildcards in the file specification.

13.10 Disk Utility Procedures

The disk utility procedures provided by the disk File Service are summarized in this section. To use these procedures, you must include in the compilation of your program the $DISK_UTILITY module appropriate for the language you are using (see the *VAXELN Development Utilities Guide*).

13.10.1 ELN$DISMOUNT_VOLUME Procedure

The ELN$DISMOUNT_VOLUME procedure dismounts a file service disk volume on the specified device. The procedure must be called on the same node that has the File Service. A dismounted disk can be opened and used for nonfile, or *logical block*, I/O.

A string of 1 to 30 characters names the device, for example, *DQA1* for drive 1 on disk controller DQA. The user must have read, write, execute, and delete (RWED) privileges to dismount a volume.

13.10.2 ELN$INIT_VOLUME Procedure

The ELN$INIT_VOLUME procedure initializes a disk for use as a Files–11 file-structured volume. Disks must be initialized once before they are used. You can initialize any volume on any node running a VAXELN system, provided the volume is not mounted or already open. The procedure must be called on the same node that has the File Service.

A string of 1 to 30 characters gives the device specification of the disk drive, for example, *DQA1* for drive 1 on disk controller DQA. The node must be specified explicitly for a drive on another node. A string of 1 to 12 characters gives the volume label for the disk.

An optional argument supplies the default extension quantity in blocks for the files on the disk volume. The extension quantity is applied when the size of a file is increased beyond its initial allocation by writing more records to the file.

Optional arguments supply a user name to be recorded on the volume and an integer identifying the UIC of the volume owner. The volume, file, and record protection for the volume are also specified by optional arguments. (See Chapter 12 for more information on protection.)

Other optional arguments designate the following:

- The number of directories that can be cached by the File Service by default
- The maximum number of files that can exist on a disk
- The number of entries that are preallocated for user directories
- The number of file headers allocated initially for the index file (the file for the volume's file structure)

- The number of mapping pointers to be allocated for file windows, which are used to describe the logical segments of the file for access
- The cluster size (the minimum allocation unit for the volume)
- The position of the index file (beginning, middle, or end)
- Whether data checking on read or write operations is enabled or disabled
- Whether the volume is shareable
- Whether the volume is a group volume
- Whether the volume is a system volume
- Whether the volume has information about where bad blocks are located

A required argument supplies a list of bad blocks. Bad blocks are areas on the volume that are known to be faulty and are marked by the procedure so that no data will be written on them. The bad-block list specifies a range of either logical or physical block numbers. You can specify a null list.

13.10.3 ELN$MOUNT_VOLUME Procedure

The ELN$MOUNT_VOLUME procedure mounts a disk for use as a file-structured volume. The procedure requires the device, its driver, and the File Service to be present in the same system from which it is called. The procedure does not return until the disk is mounted.

A string of 1 to 30 characters names the disk drive on which the volume is to be mounted, for example, *DQA1* for drive 1 on disk controller DQA.

An optional argument of 1 to 12 characters supplies the volume label. If the argument is omitted, the procedure mounts whichever volume is loaded in the indicated drive.

13.11 Tape Utility Procedures

The tape utility procedures provided by the tape File Service are summarized in this section. To use these procedures, you must include the $TAPE_UTILITY module appropriate for the language you are using in the compilation of your program (see the *VAXELN Development Utilities Guide*).

13.11.1 ELN$DISMOUNT_TAPE_VOLUME Procedure

The ELN$DISMOUNT_TAPE_VOLUME procedure dismounts a file service magnetic tape volume on the specified device. The procedure must be called on the same node that has the File Service. A string of 1 to 30 characters names the device — for example, *MUA0* for drive 0 on tape controller MUA. An optional argument designates whether the tape will be unloaded by the device.

13.11.2 ELN$INIT_TAPE_VOLUME Procedure

The ELN$INIT_TAPE_VOLUME procedure initializes a file service magnetic tape as a tape volume that conforms to American National Standards Institute (ANSI) standard X3.27–1978. Tapes must be initialized before they are used. The procedure requires the device, its driver, and the tape File Service to be present in the same system from which it is called. The procedure does not return until the tape is initialized.

A string of 1 to 30 characters gives the device specification of the tape drive, for example, *MUA0* for drive 0 on tape controller MUA. The node must be specified explicitly for a drive on another node. A string of 1 to 6 characters gives the volume label for the tape. An optional argument designates the density of data recorded on the tape.

13.11.3 ELN$MOUNT_TAPE_VOLUME Procedure

The ELN$MOUNT_TAPE_VOLUME procedure mounts a file service magnetic tape as a tape volume that conforms to American National Standards Institute (ANSI) standard X3.27–1978. The procedure requires the device, its driver, and the tape File Service to be present in the same system from which it is called. The procedure does not return until the tape is mounted.

A string of 1 to 30 characters names the tape drive on which the volume is to be mounted, for example, *MUA0* for drive 0 on tape controller MUA. An optional argument of 1 to 6 characters supplies the volume label. If the argument is omitted, the procedure mounts whichever volume is loaded in the indicated drive.

Optional arguments designate the block size of new files and whether the tape volume can be written to.

13.12 File Service Interface for Disk and Tape Drivers

This section is provided for anyone who is writing new disk or tape drivers that will use the File Service or who wants to study the drivers supplied with the development toolkit. You do not need this information for normal use of the VAXELN Toolkit.

The File Service consists of two separate shareable images: FILE.EXE, which is the disk File Service shareable image, and TAPE.EXE, which is the tape File Service shareable image. The appropriate shareable image is linked to each disk and tape driver installed in a VAXELN system and is activated by calling routines from the respective driver.

The following File Service initialization routines are available:

- The function ELN$FILE_INITIALIZE defines the actions *open*, *close*, *get*, and *put* for the specific disk device being driven.
- The function ELN$TAPE_INITIALIZE defines the actions *open*, *close*, *get*, *put*, *reposition*, *tapemark*, *erase*, and *return* for the specific tape device being driven.

Normally, one of these functions is called by the driver's master process as part of its initialization sequence. The arguments are functions, or action routines, that define the operations for the device. The function returns a *file context* variable that is used by the File Service.

Since most controllers support multiple units, typical drivers are multitasking programs that create a process to handle each drive. Therefore, after defining the action routines with ELN$FILE_INITIALIZE or ELN$TAPE_INITIALIZE, as appropriate, the driver creates a process for each attached drive.

The drive process is usually passed some kind of argument identifying the drive, such as a unit number. The initializing process then waits for a *start-up* event to be signaled, meaning that one drive is initialized and the initializing process can proceed with creating other drive processes. (Depending on the driver, the event value can be passed explicitly to the process or obtained in the drive process with an up-level reference.)

When all the drive processes have been started, the initializing process calls INITIALIZATION_DONE and proceeds with its other work.

Each drive process calls one of the following file service routines:

- The procedure ELN$FILE_SERVICE for disk device drivers

- The procedure ELN$TAPE_SERVICE for tape device drivers

In either case, the procedure's arguments are as follows:

- The start-up event value (*startup_event*)
- The file context (*file_context*)
- A string naming the drive (*drive_name*) (typical drivers take the controller name as a program argument and concatenate a digit to it to form the drive name)
- A *drive context* pointer, where the drive context (*drive_context*) is a structure defining the state of an individual drive and is usually initialized by the drive process

Forming these arguments and calling the procedure are the only actions required of the drive processes.

From this point on, the File Service is in effective control of the drive and performs all I/O operations on it, including handling protocol messages. The File Service signals the start-up event after performing its own initialization, allowing the master process to proceed with the creation of the other driver processes.

The source modules for user-written drivers are as follows:

- ELN$:DAP.PAS. This module contains the Pascal language declarations of the two disk routines described in this section and the declarations of the function types you can use to declare action routines for ELN$FILE_INITIALIZE. The two tape routines described in this section and the Pascal language declarations of the function types for the action routines of ELN$TAPE_INITIALIZE are in ELN$:TAPE.PAS. The action routines' types are prefixed with DISK$ or TAPE$, as appropriate. DISK$PUT_ACTION, for example, is the function type used to declare *put* actions for disk devices.

  The precompiled version of DAP.PAS is the module $DAP and the precompiled version of TAPE.PAS is $TAPE. If you are writing a disk or tape driver for use with the File Service, be sure to include the appropriate module in its compilation.
- $DAP in ELN$:VAXELNC.TLB. This module contains the definitions for disk drivers written in C. You include this module when you compile your driver source module by issuing a command of the following form:

  ```
  #include $DAP
  ```

After including the appropriate Pascal or C module in your compilation, link the compiled driver with ELN$:RTLSHARE.OLB, which contains the shareable image of the File Service.

NOTE

A user-written driver should be capable of having any of its functions called in the context of any process, and its data base should, therefore, either be statically allocated or be allocated on the heap.

13.13 Data Access Protocol

The data access protocol (DAP) is a method for exchanging data between processes in your system and record-oriented device driver programs or services. DAP is used by the Pascal and C runtime libraries to exchange I/O requests and results between the user's program and device drivers.

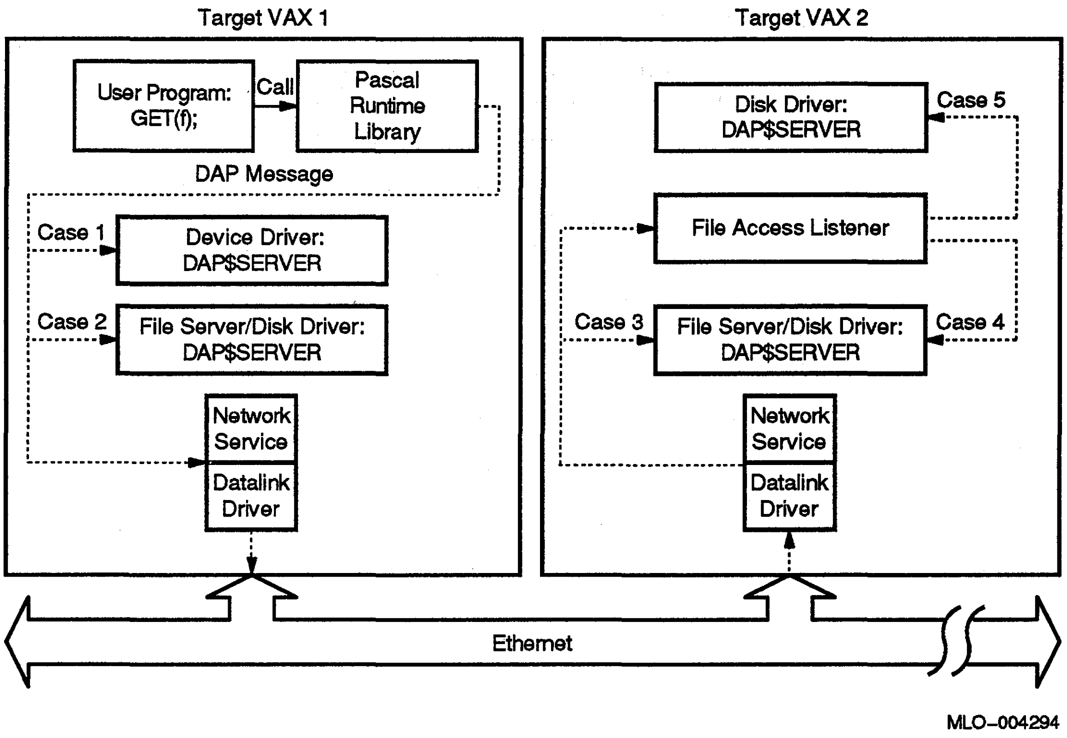
This section explains the use of the development system's DAP facilities for anyone who is writing file- or record-oriented device drivers, or for anyone who is studying the drivers that Digital supplies. You will not need this information for normal use of the VAXELN Toolkit or for writing disk or tape drivers that will not use the File Service. A typical occasion for using the DAP is to add support for a new type of disk controller.

Writing drivers with the DAP is usually simple because you have only to write definitions of a set of preestablished functions called *action routines*. Typically, you write definitions of *open*, *close*, *get data*, and *put data* that are appropriate for the device in question. The definition of each action routine in your program is accomplished with predeclared constants, data types, and functions, which are discussed in this section.

For practical information on the use of the DAP in driver writing, study the driver and definition sources supplied with your development system.

Figure 13–1 illustrates the message flow in a typical I/O operation.

Figure 13–1: DAP Message Transmission (Read Request)



In the example illustrated by Figure 13–1, a user program makes a *read* request, the Pascal GET procedure. When the runtime library is called, it generates a DAP message formulating the read request. There are then five cases that describe the destination and processing of the message, depending on the way the file was originally opened:

Case 1 The program has opened a local terminal for logical I/O, as in:

```
OPEN(f,FILE_NAME := 'TTA0:')
```

The message is sent directly to the terminal driver by translating the local name TTA0, which has called the function DAP$SERVER to define the actions for servicing DAP requests directed at its device. (Action routines are discussed in Section 13.13.2.)

Case 2 The program has opened a file on a mounted disk volume, as in:

```
OPEN(f, FILE_NAME := 'DISK$VDATA:[mydir]file.dat')
```

In this case, DISK$VDATA is a universal name established by the File Service, naming the port that receives DAP requests for the disk volume of the given name. The DAP request is thus received and processed by the File Service and the associated disk driver for that volume.

Case 3 The OPEN call is as in Case 2, but the volume name does not have a local translation. The Network Service receives the message and encloses it in an NSP message for transmission by the datalink drivers over the Ethernet to the node (here, VAX2) that has the named message port. The DAP message reemerges from node B's Network Service with the NSP envelope removed. The named port is defined in the job running node B's disk driver, and the read request is handled there.

Case 4 The OPEN call used an explicit node name to access a file on a mounted disk (DUA1), as in:

```
OPEN(f, FILE_NAME := 'VAX2::DUA1:[mydir]file.dat')
```

After transmission to node VAX2, the message is intercepted by that node's FAL and sent on to the File Service on that node. (In most respects, this case also applies if node VAX2 is a VMS node, although the node is then specified by number instead of by name; similarly, it could occur if node VAX1 is a VMS node at which a comparable OPEN call was made from a VMS program.)

Case 5 The OPEN call specified a node explicitly, to open a remote terminal, as in:

```
OPEN(f,FILE_NAME := 'VAX2::TTA0:')
```

This is the network version of Case 1; the terminal TTA0 on node VAX2 was opened for logical I/O.

In all cases, the device driver manipulates the device registers to perform the input or output. The device driver or File Service uses the function DAP$SERVER to handle the message. Figure 13–1 shows the flow of the read-request message; the requested record, in each case, flows back to the requesting program on the same path.

When the driver uses the data access protocol, the driver must be on the same network node as the device it controls, but the driver — and thus, the device — can be used by programs located anywhere in the local area network.

The DAP is supported by a set of precompiled modules (for Pascal only) and a set of declarations, including types, constants, and function types (action routines). The Pascal declarations are used in programs by including the module $DAP from RTLOBJECT.OLB in the compilation. The corresponding definitions for C are contained in the module $DAP in ELN$:VAXELNC.TLB.

13.13.1 DAP General Principles

In data communication, a *protocol* is a definition of a set of messages and, usually, the means of exchanging the messages.

The data access protocol defines two things:

- A set of messages. Each message has a predefined format and meaning, and definitions are provided in the DAP for messages of every kind likely to be relevant to talking to record-oriented devices: specifying a file and the kind of access requested, sending control information (commands to read, write, and so on) defining the characteristics of files and devices, and so forth.
- A method of starting a message exchange (action routines).

The DAP assumes that a communications path already exists for the messages, which, in VAXELN programming, is a circuit. (See Section 5.3.6.)

The low-level operations of locating the communicating processes and formatting, interpreting, and transmitting messages are done by runtime library routines. When writing a device driver, you can regard these routines as *black boxes*, since you do not have to call any of them explicitly, except DAP$SERVER.

In writing device drivers, the use of the DAP requires three steps:

1. Define a set of action routines appropriate to the device.
2. Establish circuits with any user processes that want to do something with the device.

3. Call the library function DAP$SERVER with parameters that supply the circuit — that is, the communication path between the device and the user process — and the set of action routines you have defined in the driver.

The management of messages and other low-level operations is then done implicitly by DAP$SERVER. Almost all other code in DAP device drivers is concerned with servicing device interrupts.

13.13.2 Action Routines and DAP$SERVER

An action routine defines your *choice* of DAP information that should be transmitted to perform a particular operation, such as reading a data record. The information is represented by a set of predeclared data types and constants.

DAP$SERVER is a predeclared function. The following Pascal declaration is included with module $DAP. (See also the source file DAP.PAS.)

```
FUNCTION dap$server(VAR circuit_port: port;
  FUNCTION   open_action OF TYPE dap$open_action;
  [OPTIONAL] FUNCTION rename_action OF TYPE dap$rename_action;
  [OPTIONAL] FUNCTION dir_open OF TYPE dap$dir_open;
  [OPTIONAL] FUNCTION dir_list OF TYPE dap$dir_list;
  [OPTIONAL] FUNCTION erase_action OF TYPE dap$erase_action;
  [OPTIONAL] FUNCTION get_action OF TYPE dap$get_action;
  [OPTIONAL] FUNCTION put_action OF TYPE dap$put_action;
  [OPTIONAL] FUNCTION find_action OF TYPE dap$find_action;
  [OPTIONAL] FUNCTION update_action OF TYPE dap$update_action;
  [OPTIONAL] FUNCTION rewind_action OF TYPE dap$rewind_action;
  [OPTIONAL] FUNCTION truncate_action OF TYPE dap$truncate_action;
  [OPTIONAL] FUNCTION flush_action OF TYPE dap$flush_action;
  [OPTIONAL] FUNCTION extend_action OF TYPE dap$extend_action;
  [OPTIONAL] FUNCTION display_action OF TYPE dap$display_action;
  [OPTIONAL] FUNCTION close_action OF TYPE dap$close_action;
  dap_buffer_size: integer := 0;
  context: integer := 0
  ): integer;

  SEPARATE;
```

The action routines, in turn, are represented by function types, for example:

```
FUNCTION dap$put_action(
  record_access : dap$b_rac;
  record_number : INTEGER;
  record_options : dap$l_rop;
  buffer : ^STRING(32767);
  buffer_length : INTEGER;
  context: integer;
  var record_file_address: dap$r_rfa;
  next_record: BOOLEAN)
  : dap$l_status;

FUNCTION_TYPE;
```

For the definitions of all DAP function types — that is, the action routines' parameters — and DAP$SERVER's parameters, see the file DAP.PAS.

NOTE

The preceding discussion applies to Pascal programs only. The equivalent interface is available to C programmers using the $DAP include module contained in ELN$:VAXELNC.TLB.

13.13.3 DAP Data Types

Each kind of action routine is associated with a set of data types representing the routine's parameters. In addition, the result type *dap$l_status*, shown in Section 13.13.2, represents the success/failure status of each action-routine call. For the definitions of the types of action-routine parameters and the result type *dap$l_status*, see the source file DAP.PAS, supplied with your development system.

13.13.4 DAP Constants

A large set of named constants are declared for use in DAP device drivers. For example, the named constant *dap$k_seq_acc* can be used as an open-file argument to indicate sequential access. For the list of names and their definitions, see the source file DAP.PAS, supplied with your development system. This same file defines the named constants representing action routine completion status, error status, control functions, and so forth.

Many of the status constants are defined in DAP.PAS with reference to other, lower-level named constants. The definitions of these constants are in the file DAPSTATUS.PAS.

13.13.5 DAP Wildcard Functions

The DAP$SERVER, upon receiving a retrieval, rename, or delete access function, checks the file specification parameter for any wildcard characters. If there are any, it recursively invokes itself to perform the function.

Chapter 14

VAXELN Device Drivers

The VAXELN Toolkit supplies drivers for a variety of devices. The supplied drivers include the following:

- Disk drivers, Section 14.1
- Tape driver, Section 14.2
- Printer drivers, Section 14.3
- Terminal drivers, Section 14.4
- Small Computer System Interface (SCSI) bus driver, Section 14.5
- Realtime device drivers, Section 14.6

This chapter discusses the features of the supplied drivers and explains how VAXELN applications can perform parallel I/O.

14.1 Disk Drivers

The VAXELN Toolkit includes device drivers for a number of mass storage devices. Table 14–1 lists these drivers with the devices they support.

Table 14–1: Disk Drivers

Driver	Supported Mass Storage Devices
BDDRIVER	Disk devices that use the VAXBI bus through a KDB50 VAXBI bus disk adapter, including the RA*nn* disks

Table 14–1 (Cont.): Disk Drivers

Driver	Supported Mass Storage Devices
DDDRIVER	TU58 (VAX–11/730 and VAX–11/750) console tape cartridges, which, operationally, resemble disk devices
DIDRIVER	RF*nn* disks attached to the MicroVAX 3300 and 3400 Integrated Disk Controller
DQDRIVER	RB02 and RB80 disks attached to the VAX–11/730 Integrated Disk Controller
DUDRIVER	Disk devices that use the UNIBUS through a UDA50 UNIBUS disk adapter, including the RA*nn* disk drives
	Disk devices that use the RQDX*n* interfaces on the MicroVAX, including RX*nn* diskettes and RD*nn* Winchester disks
	The RC25 controller for the Q-bus and UNIBUS
	Disk devices that use the KDA50 interface on the MicroVAX, including the RA*nn* disks
	Disk devices that use the KFQSA Q-bus controller, including the RF*nn* disks
DVSDRIVER	RX33, RD53, and RD54 disk devices attached to the MicroVAX 2000 disk subsystem
SCDRIVER	RZ*nn* Winchester disks, RX23 SCSI diskettes, RRD40 compact discs, and third-party Small Computer System Interface (SCSI) devices attached to a SCSI bus on MicroVAX, VAXstation, and rtVAXstation 3100 series systems.

To use the disk interfaces and drives on a VAXELN target processor, you must build the appropriate driver into the VAXELN system that is to run on that processor. If you use the supported disk types and drivers as supplied, you can regard the drivers, and the File Service, as self-contained programs that perform I/O for you. All you need to know in such cases is how to build the drivers into your systems. This information is provided in the *VAXELN Development Utilities Guide*.

NOTE

If you build DIDRIVER into a VAXELN system, you must specify an additional 256 pages for the system's system region size.

14.1.1 Logical I/O

When a disk is not mounted, you can access it directly by using language-specific I/O routines or statements. You open a disk for *logical I/O* (nonfile I/O) by specifying the disk's device name instead of a file name in a call to the Pascal OPEN procedure, C open functions, or FORTRAN OPEN statement. Operations that you perform on the open file variable apply to the disk volume itself, as if it were a single, large file with the first record (record number 1) starting at block 0 on the disk.

Logical I/O lets a program maintain and use its own information about the logical structure of records in a file. It is up to the program to interpret the structure of individual records, read from the disk, record the placement of records relative to one another, and perform other operations that the File Service normally handles.

NOTE

When you open a disk for logical I/O, no other job can access the disk.

You can write your own disk drivers that are compatible with this method and with the File Service. For information on writing disk drivers that are compatible with the File Service, and for general information on the Data Access Protocol (DAP) used by the language-specific I/O routines and statements, see Sections 13.12 and 13.13.

14.1.2 Disk Specifications

Table 14–2 lists specifications for the devices that the VAXELN disk drivers support.

Table 14–2: Disk Devices

Drive	Device Code	Media Type	Bytes/Disk	Disks/ Drive	Drives/ Controller	Driver Image
RA60	DU BD	Cartridge	205 Mbyte	1	4	DUDRIVER.EXE BDDRIVER.EXE
RA70	DU BD	Fixed disk	280 Mbyte	1	4	DUDRIVER.EXE BDDRIVER.EXE

Table 14–2 (Cont.): Disk Devices

Drive	Device Code	Media Type	Bytes/Disk	Disks/ Drive	Drives/ Controller	Driver Image
RA80	DU BD	Fixed disk	121 Mbyte	1	4	DUDRIVER.EXE BDDRIVER.EXE
RA81	DU BD	Fixed disk	456 Mbyte	1	4	DUDRIVER.EXE BDDRIVER.EXE
RA82	DU BD	Fixed disk	622 Mbyte	1	4	DUDRIVER.EXE BDDRIVER.EXE
RA90	DU BD	Fixed disk	1.2 Gbyte	1	4	DUDRIVER.EXE BDDRIVER.EXE
RB02	DQ	Cartridge	10 Mbyte	1	4	DQDRIVER.EXE
RB80	DQ	Fixed disk	119 Mbyte	1	1	DQDRIVER.EXE
RD31	DU	Fixed disk	20 Mbyte	1	2	DUDRIVER.EXE
RD32	DU	Fixed disk	42 Mbyte	1	2	DUDRIVER.EXE DVSDRIVER.EXE
RD51	DU	Fixed disk	10 Mbyte	1	2	DUDRIVER.EXE
RD52	DU	Fixed disk	31 Mbyte	1	2	DUDRIVER.EXE
RD53	DU	Fixed disk	71 Mbyte	1	2	DUDRIVER.EXE DVSDRIVER.EXE
RD54	DU	Fixed disk	150 Mbyte	1	2	DUDRIVER.EXE DVSDRIVER.EXE
RF30	DI	Fixed disk	150 Mbyte	1 1	6 1	DIDRIVER.EXE DUDRIVER.EXE[1]
RF71	DI	Fixed disk	400 Mbyte	1 1	6 1	DIDRIVER.EXE DUDRIVER.EXE[1]
RRD40[2]	DU	Compact disc	577 Mbyte	1	1	SCDRIVER.EXE
RX23	DU	Diskette	1.4 Mbyte	1	1	SCDRIVER.EXE
RX33	DU	RX33 diskette[3]	1.2 Mbyte	1 1	2 1	DUDRIVER.EXE DVSDRIVER.EXE
RX50	DU	Diskette	400 Kbyte	2	4	DUDRIVER.EXE
RZ22	DU	Fixed disk	52 Mbyte	1	1	SCDRIVER.EXE

[1]When used with the KFQSA Q-bus controller

[2]Read-only device

[3]The RX33 drive also supports RX50 diskettes.

Table 14–2 (Cont.): Disk Devices

Drive	Device Code	Media Type	Bytes/Disk	Disks/ Drive	Drives/ Controller	Driver Image
RZ23	DU	Fixed disk	104 Mbyte	1	1	SCDRIVER.EXE
RZ55	DU	Fixed disk	332 Mbyte	1	1	SCDRIVER.EXE
RZ56	DU	Fixed disk	665 Mbyte	1	1	SCDRIVER.EXE
TU58	DD	Tape cartridge	256 Kbyte	2	2	DDDRIVER.EXE

The RB02 and RB80 devices use the VAX–11/730 Integrated Disk Controller (RB730). You can attach a total of four drives to the controller and only one of them can be an RB80. RB02 cartridges are identical to RL02 cartridges, and the cartridges can be interchanged between these two drive types.

The TU58 cartridge is the console medium on VAX–11/730 and VAX–11/750 processors. The cartridge is treated as if it were a random-access disk with one cylinder, four tracks per cylinder, 128 512-byte blocks per track. It is controlled by processor registers.

RQDX*n* controllers interface up to four disk drives to the MicroVAX Q22-bus; up to two of these drives can be Winchester RD*nn* disks.

The RD32, RD53, RD54, and RX33 devices can use the MicroVAX 2000 Integrated Disk Controller. This controller interfaces up to three disk drives; up to two of these drives can be Winchester RD*nn* disks, and one can be an RX33 drive. Devices that use this controller use the driver image DVSDRIVER.EXE.

The RF30 and RF71 disks are integrated storage elements (ISEs) that can interface with the MicroVAX 3300 and MicroVAX 3400 Integrated Disk Controller or the KFQSA Q-bus controller. The disks use the DIDRIVER.EXE image to interface with the integrated disk controller. This controller communicates with up to six disks using the Digital Storage System Interconnect (DSSI) bus.

The RF30 and RF71 disks use the DUDRIVER.EXE image when interfacing with the KFQSA Q-bus controller. This controller also supports up to six RF*nn* disks. However, only one disk is supported per driver image. To support multiple RF*nn* disks, you must include a copy of the DUDRIVER.EXE image in your system for each disk.

Bad blocks are handled on disks and diskettes according to the device. RD*nn* disk devices support controller-initiated bad block replacement; that is, the RQDX*n* controller automatically handles bad blocks. However, the DVSDRIVER handles bad block replacement and vectoring for RD*nn* disks used with the MicroVAX 2000 Integrated Controller.

An RA*nn* disk interfaces to a VAX bus by using a disk adapter or controller.

- The UDA50 disk adapter interfaces RA*nn* disks to the VAX UNIBUS.
- The KDA50 disk controller interfaces the RA*nn* disks to the MicroVAX Q-bus.
- The KDB50 disk adapter interfaces the RA*nn* disks to the VAXBI bus.

The disk adapter or disk controller you use for an RA*nn* disk determines the driver image you should use. The driver image DUDRIVER.EXE is for UDA50 and KDA50 I/O; the image BDDRIVER.EXE is for KDB50 I/O.

RA*nn* and RC25 devices support host-initiated bad block replacement; that is, the driver automatically revectors bad blocks as they occur on the disks.

14.1.3 Disk Driver Interface to the File Service

The VAXELN disk drivers include the disk File Service, which supports the Files–11 on-disk structure that the VMS systems use. Therefore, you can move disk volumes to a VMS system and use them with VMS software. Also, most VMS file-handling commands can use disks mounted on VAXELN systems when the systems are part of a network that includes VMS systems.

A disk driver uses the File Service to perform the following operations on a disk:

Operation	Description
Open	Prepares a device and its driver for program I/O. The File Service performs this operation when you mount a disk volume or when the first user program accessing the disk for logical I/O calls the Pascal OPEN procedure, C open functions, or FORTRAN OPEN statement.
Get	Reads data from a disk. The File Service performs this operation when language-specific input routines or statements retrieve information from a disk volume.
Put	Writes data on a disk. The File Service performs this operation when language-specific output routines or statements add information to a disk volume.
Close	Terminates I/O exchange with a user program. The File Service performs this operation when you dismount a disk volume or when the last user program accessing the disk for logical I/O calls the Pascal CLOSE procedure, C close functions, or FORTRAN CLOSE statement.

14.1.4 Recovery from Power Failure

When disks are on line and mounted, they are brought back on line and remounted automatically following a power failure. The device driver reinitializes the disk controller. The File Service operations that were in progress when the power failed are retried, and the disks can be used again without manual intervention.

Spinning down an RC25 controller and later spinning it back up is equivalent to a power-failure recovery. The actions just described apply in this case.

14.1.5 Direct Device Access for Disk Devices

Direct device access (DDA) provides an interface that VAXELN applications can use to read data from and write data to local disks directly, avoiding the overhead incurred by the data access protocol (DAP). The DDA disk interface also provides for physical memory transfers by allowing applications to transfer data to and from an allocated system region. The interface consists of the runtime routines ELN$DISK_READ and ELN$DISK_WRITE, which read blocks of data from and write blocks of data to a local disk drive using the DDA protocol.

When the kernel initializes a disk driver, it creates a DDA port and a corresponding local port name of the form *drive-name*$ACCESS for each drive. For example, if DUDRIVER controls the drive named DUA1, the kernel creates the local name DUA1$ACCESS.

To use the DDA disk interface routines, an application must first open the appropriate file or device (to gain access to an unmounted disk) and establish a VAXELN virtual circuit with a disk driver. The application uses the circuit to communicate with the driver. To establish the circuit connection, the application must create a port and connect that port to the disk drive's DDA port. Once the connection is made, the program can call ELN$DISK_READ and ELN$DISK_WRITE to read and write data.

For descriptions of the ELN$DISK_READ and ELN$DISK_WRITE routines, see *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*. Section 14.1.5.1 explains how to establish a circuit for the DDA disk interface. Sections 14.1.5.2 to 14.1.5.5 explain how to use the interface to do the following:

- Perform direct read and write operations, Section 14.1.5.2
- Read logical blocks from an unmounted disk, Section 14.1.5.3
- Read logical blocks from a mounted disk, Section 14.1.5.4
- Transfer data to physical addresses, Section 14.1.5.5

14.1.5.1 Establishing Circuits for the DDA Disk Interface

An application program communicates with a disk driver using a VAXELN virtual circuit. The program must establish the circuit connection by creating a port and connecting that port to a disk drive's DDA port. Once the connection is made, the program can call the ELN$DISK_READ and ELN$DISK_WRITE routines to read and write data. The following example connects the port *drive1_port* in a circuit to the DDA port named DUA1$ACCESS:

```
MODULE test_drive;

INCLUDE $DDA_UTILITY;

PROGRAM test_read_write;
```

```
VAR
  drive1_port, dda_port : PORT;
    .
    .
    .
BEGIN
  CREATE_PORT(drive1_port);
  CONNECT_CIRCUIT(drive1_port, DESTINATION_NAME := 'DUA1$ACCESS');
    .
    .
    .
END.
END;
```

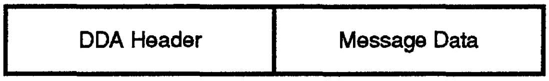
Once the connection between *drive1_port* and the DDA port is established, the program can call ELN$DISK_READ and ELN$DISK_WRITE, specifying *drive1_port* as an argument.

14.1.5.2 Reading Data from and Writing Data to a Local Disk

An application program can read data from and write data to a local mounted or unmounted disk by calling the ELN$DISK_READ and ELN$DISK_WRITE routines. These routines communicate with a disk driver by using a user-defined message. They send the message to the disk driver and wait for and receive a response. In the case of read operations, the message sent contains a read request and the message returned contains the data being read. In the case of write operations, the message sent contains the data to be written and the message returned contains the status of the write operation.

Alternatively, an application can transfer data to an allocated system region. An application must use this method if it needs to lock the data buffer at a specific physical address. If an application specifies a physical address, ELN$DISK_READ and ELN$DISK_WRITE transfer the data directly to or from the system region at that address. The message sent or received contains a DDA header.

Before calling ELN$DISK_READ or ELN$DISK_WRITE, you must create a message to be used for the data transfers. The message must be large enough to handle the largest possible transfer request and accommodate a DDA header of size DDA$_HEADER_SIZE. The following figure shows such a message:



MLO–004168

If the application is to transfer data to and from an allocated system region, the message must be large enough to accommodate just the header.

Calls to ELN$DISK_READ and ELN$DISK_WRITE must specify the port connected in a circuit to the disk drive's DDA port, the number of bytes of data to be read or written (read or write size), and the starting logical block number on the disk where the read or write operation is to begin.

A bytes transferred argument receives the number of bytes of data actually read or written.

You must also specify the identifier and pointer for the previously created message. These input/output arguments represent the message sent to and received from the driver. ELN$DISK_READ uses the specified message to send a read request to the driver and to receive the data read. ELN$DISK_WRITE uses the message to send the data being written and to receive the completion status of the write operation. If you specify an allocated system region, the routines use the message to transfer only the DDA header.

If an application uses the message arguments — for example, to gain access to the message data, it must ensure that it uses the current values, and if necessary, points to the message data. The message arguments are input/output arguments and the kernel may not map the sent and received messages to the same P0 virtual address space. For example, this might happen if another process in the job runs and either uses some memory or returns memory to the system while the read or write operation is in progress. If this occurs, the message pointer value that the kernel returns might differ from the pointer value of the message that was sent. Similarly, the value of the message identifier might change.

To read data from or write data to the message data buffer, an application must also set up a pointer to the data portion of the message. An application can do this by doing one of the following:

- Declaring a 2-field aggregate to represent the message and declaring a pointer to that aggregate
- Setting up a pointer to the message data directly

An example of how an application might use an aggregate to represent a DDA message that has a data buffer of size 4096 bytes (eight disk blocks) follows:

```
TYPE
  RECORD = dda_message;
           dda_header : BYTE_DATE(DDA$_HEADER_SIZE);
           data_buffer : ARRAY[1..1024] OF INTEGER;
           END;

VAR
  msg_ptr : ^dda_message;
```

After declaring the pointer to the aggregate, the application can specify the message pointer in the calls to CREATE_MESSAGE, ELN$DISK_READ, and ELN$DISK_WRITE. The application can also use the pointer value to gain access to data read or to fill in data to be written. The following code shows how an application might gain access to the data buffer after a read operation. The assignment statement adjusts the pointer such that it points to the twelfth array element in the data buffer.

```
  ELN$DISK_READ(status,
                access_port,
                READ_SIZE := bytes_to_read,
                BLOCK_NUMBER := starting_lbn,
                BYTES_TRANSFERRED := bytes_read,
                MSG_OBJ := msg_object,
                MSG_PTR := msg_ptr);

  data_ptr := msg_ptr^.data_buffer[12]
```

If you prefer not to use the an aggregate representation, you might set up a pointer to the message data buffer before a write operation or after a read operation as follows:

```
data_ptr::INTEGER := msg_ptr::INTEGER + DDA$_HEADER_SIZE;
```

For information about transferring data to an allocated system region see Section 14.1.5.5.

14.1.5.3 Reading Logical Blocks from an Unmounted Disk

You can use the DDA disk interface to transfer data to and from a mounted or unmounted local disk. Example 14–1 shows an example of how you might use the interface to read logical blocks of data from an unmounted disk.

Example 14–1: Reading Logical Blocks from an Unmounted Disk

```
MODULE disk_read_unmounted;

INCLUDE $dda_utility;

PROGRAM disk_read_unmounted(INPUT, OUTPUT);

TYPE
  byte = -128..127;
  block = PACKED ARRAY [1..512] OF BYTE;

VAR
  total_blocks : INTEGER;
  device_file : FILE OF block;
  msg_obj : MESSAGE;
  msg_ptr : ^ANYTYPE;
  bytes_to_read, starting_lbn : INTEGER;
  data_ptr : ^ANYTYPE;
  access_port : PORT;
  drive_name, remote_port_name, file_name : VARYING_STRING (32);
  status, bytes_read : INTEGER;
  s_time, e_time, d_time : LARGE_INTEGER;
  s_time_asc, e_time_asc, d_time_asc : VARYING_STRING(23);

BEGIN { Main program }

  CREATE_PORT(access_port);                                 ❶

  WRITE('Enter the drive name [DUA1]: ');
  READLN(drive_name);
  IF drive_name = '' THEN drive_name := 'DUA1';
  remote_port_name := drive_name + '$ACCESS';

  file_name := drive_name + ':';
  OPEN(device_file,
       FILE_NAME := file_name,
       HISTORY := HISTORY$READONLY);

  CONNECT_CIRCUIT(access_port, DESTINATION_NAME := remote_port_name);
```

Example 14–1 Cont'd on next page

Example 14–1 (Cont.): Reading Logical Blocks from an Unmounted Disk

```
REPEAT
  WRITE('How many blocks are to be read: ');
  READLN(total_blocks);
  bytes_to_read := total_blocks * 512;
  starting_lbn := 0;
  bytes_read := 0;

  CREATE_MESSAGE(msg_obj,                                              ❷
             msg_ptr::^BYTE_DATA(DDA$_HEADER_SIZE + bytes_to_read)
             );

  GET_TIME(s_time);
  ELN$DISK_READ(status,                                                ❸
                access_port,
                READ_SIZE := bytes_to_read,
                BLOCK_NUMBER := starting_lbn,
                BYTES_TRANSFERRED := bytes_read,
                MSG_OBJ := msg_obj,
                MSG_PTR := msg_ptr);
  GET_TIME(e_time);
  WRITELN('ELN$DISK_READ status: ', status);
  d_time := e_time - s_time;
  d_time_asc := TIME_STRING(-(d_time));
  WRITELN('Time for ', bytes_read, ' byte transfer -- ',
          d_time_asc);
  WRITELN('Transfer Rate = ',
          bytes_read DIV (d_time::INTEGER DIV 10000), ' Kb/s');
 data_ptr::INTEGER := msg_ptr::INTEGER + DDA$_HEADER_SIZE;             ❹

 {
 {  Use the data read.
 {}

  DELETE(msg_obj);                                                     ❺

UNTIL total_blocks = 1;

{ Clean up }

DISCONNECT_CIRCUIT(access_port);                                       ❻
DELETE(access_port);
CLOSE(device_file);
```

Example 14–1 Cont'd on next page

Example 14–1 (Cont.): Reading Logical Blocks from an Unmounted Disk

```
END; { of main program }
END.
```

❶ **Create a port, open the device, and connect to the drive's DDA port.** Create a VAXELN message port and connect it in a circuit to the drive's DDA port. The sample module creates the message port *access_port* and connects it in circuit to the DDA port *remote_port_name*, where *remote_port_name* is a specified drive name and the string $ACCESS. The sample module also uses the specified drive name to open the device. The call to OPEN ensures that the device driver sets up the appropriate structures for the data transfer.

❷ **Create a message object.** Call CREATE_MESSAGE to create the message that is to be sent to the DDA port. The sample module creates the message *msg_obj*. The message's size is calculated based on the number of blocks that the user specifies.

❸ **Read data from the disk drive.** Call ELN$DISK_READ to read data from the disk drive. You must specify the port connected in a circuit to the drive's DDA port, the read size, the starting logical block number, a variable that is to receive the number of bytes read, the message identifier, and the message pointer. The call to ELN$DISK_READ in the sample module reads data of size *bytes_to_read*, starting at block *starting_lbn*, using port *access_port*. The number of bytes read is returned to *bytes_read*. The message is transferred using the message *msg_obj*.

❹ **Set up a pointer to the message data.** Set up a pointer to the data portion of the message and use the data read.

❺ **Clean up resources for the read operation.** Clean up resources for this read operation by deleting the message and its associated data buffer.

❻ **Clean up resources and exit.** When the user enters 1 for the number of blocks to read, clean up resources by disconnecting the access port from the drive's DDA port, deleting the access port, and closing the device file. When the cleanup is complete, exit.

14.1.5.4 Reading Logical Blocks from a Mounted Disk

You can use the DDA disk interface to transfer data to and from a mounted local disk. Example 14–2 shows an example of how you might use the interface to read a contiguous file from or write a contiguous file to a mounted disk.

Example 14–2: Reading Logical Blocks from a Mounted Disk

```
MODULE disk_read_mounted;

INCLUDE $kernelmsg, $elnmsg, $dda_utility, $file_utility;

PROGRAM disk_read_mounted(INPUT, OUTPUT);

TYPE
  byte = -128..127;
  block = PACKED ARRAY [1..512] OF BYTE;

VAR
  dda_file : FILE OF block;
  filesize : INTEGER;
  attr_rec : ^FILE$ATTRIBUTES_RECORD;
  msg_obj : MESSAGE;
  msg_ptr : ^ANYTYPE;
  total_bytes, starting_lbn : INTEGER;
  data_ptr : ^ANYTYPE;
  access_port : PORT;
  drive_name, remote_port_name, file_name : VARYING_STRING (32);
  status, bytes_xfr : INTEGER;
  s_time, e_time, d_time : LARGE_INTEGER;
  s_time_asc, e_time_asc, d_time_asc : VARYING_STRING(23);

[INLINE] PROCEDURE populate_data_buffer(flag : INTEGER);

{++
{ This procedure writes and checks an easily recognizable pattern on
{ each block.
{--}

VAR
  vbn, offset : INTEGER;
  pointer : ^INTEGER;
```

Example 14–2 Cont'd on next page

Example 14–2 (Cont.): Reading Logical Blocks from a Mounted Disk

```
BEGIN
  FOR vbn := 1 TO filesize DO
    BEGIN
      FOR offset := 1 TO 128 DO
 BEGIN
          pointer::INTEGER :=
            data_ptr::INTEGER + ((vbn - 1) * 512 ) + (offset - 1) * 4;
          CASE flag OF
            0: pointer^ := 0;
            1: pointer^ := vbn;
            2: IF pointer^ <> vbn THEN
               BEGIN
                 WRITELN('Verify failed @ VBN, offset ', vbn, offset);
                 RAISE_EXCEPTION(KER$_BAD_VALUE);
               END;
   END; { CASE }
 END; { for offset }
    END;  { for vbn }
END; { procedure }

BEGIN { Main program }

  CREATE_PORT(access_port);                                            ❶

  WRITE('Enter the drive name [DUA1]: ');
  READLN(drive_name);

  WRITE('Enter file  size in blocks: ');
  READLN(filesize);

  IF drive_name = '' THEN drive_name := 'DUA1';                        ❷
  remote_port_name := drive_name + '$ACCESS';

  file_name := drive_name + ':' + '[000000]dda_file.img';

  OPEN(dda_file,                                                       ❸
       FILE_NAME := file_name,
       HISTORY := HISTORY$NEW,
       RECORD_LENGTH := 512,
       RECORD_TYPE := RECORD$FIXED,
       ACCESS_METHOD := ACCESS$DIRECT,
       CONTIGUOUS := TRUE,
       FILESIZE := filesize);

  IF filesize > 0 THEN                                                 ❹
    BEGIN
      LOCATE(dda_file, filesize);
      PUT(dda_file);
    END;
```

Example 14–2 Cont'd on next page

Example 14–2 (Cont.): Reading Logical Blocks from a Mounted Disk

```
CLOSE(dda_file);

OPEN(dda_file,                                                        ❺
     FILE_NAME := file_name,
     HISTORY := HISTORY$OLD,
     FILE_ATTRIBUTES := attr_rec);

IF attr_rec = NIL THEN
  RAISE_EXCEPTION(KER$_BAD_STATE)
ELSE IF attr_rec^.starting_block_number = 0 THEN
  RAISE_EXCEPTION(KER$_BAD_STATE);

CONNECT_CIRCUIT(access_port,                                          ❻
                DESTINATION_NAME := remote_port_name);

total_bytes := filesize * 512;
starting_lbn := attr_rec^.starting_block_number;
bytes_xfr := 0;

create_message(msg_obj,                                               ❼
               msg_ptr::^byte_data(dda$_header_size + total_bytes));

data_ptr::INTEGER := msg_ptr::INTEGER + DDA$_HEADER_SIZE;

{ Initialize the message data buffer for a write operation. }

populate_data_buffer(1);

GET_TIME(s_time);
ELN$DISK_WRITE(status,                                                ❽
               access_port,
               WRITE_SIZE := total_bytes,
               BLOCK_NUMBER := starting_lbn,
               BYTES_TRANSFERRED := bytes_xfr,
               MSG_OBJ := msg_obj,
               MSG_PTR := msg_ptr);
GET_TIME(e_time);
WRITELN('ELN$DISK_WRITE status: ', status);
d_time := e_time - s_time;
d_time_asc := TIME_STRING(-(d_time));
WRITELN('Time for ', bytes_xfr, ' byte transfer -- ', d_time_asc);
WRITELN('Write Transfer Rate = ',
        bytes_xfr DIV (d_time::INTEGER DIV 10000), ' Kb/s');

data_ptr::INTEGER := msg_ptr::INTEGER + DDA$_HEADER_SIZE;

{ Clear message data buffer before read operation. }

populate_data_buffer(0);
```

Example 14–2 Cont'd on next page

Example 14–2 (Cont.): Reading Logical Blocks from a Mounted Disk

```
  GET_TIME(s_time);
  ELN$DISK_READ(status,                                              ❾
                access_port,
                READ_SIZE := total_bytes,
                BLOCK_NUMBER := starting_lbn,
                BYTES_TRANSFERRED := bytes_xfr,
                MSG_OBJ := msg_obj,
                MSG_PTR := msg_ptr);
  GET_TIME(e_time);
  WRITELN('ELN$DISK_READ status : ', status);
  d_time := e_time - s_time;
  d_time_asc := TIME_STRING(-(d_time));
  WRITELN('Time for ', bytes_xfr, ' byte transfer -- ', d_time_asc);
  WRITELN('Read Transfer Rate = ',
          bytes_xfr DIV (d_time::INTEGER DIV 10000), ' Kb/s');

  data_ptr::INTEGER := msg_ptr::INTEGER + DDA$_HEADER_SIZE;

  { Check the message data read. }

  populate_data_buffer(2);

  DELETE(msg_obj);                                                   ❿
  DISCONNECT_CIRCUIT(access_port);
  CLOSE(dda_file);
  DELETE(access_port);

END; { of main program }
END.
```

❶ **Create a port to be connected to the drive's DDA port.** Use a call to CREATE_PORT to create a VAXELN message port. This port is to be connected to the drive's DDA port. The sample module creates the message port *access_port*.

❷ **Get the name of the drive's DDA port.** Get the name of the drive's DDA port. The name of the DDA port in the sample module is *remote_port_name*, where *remote_port_name* is a specified drive name and the string $ACCESS.

❸ **Create a file.** Use a call to OPEN to create a file. The sample module creates an empty contiguous file defined as having 512-byte, fixed-length records.

❹ **Extend the file to the correct size.** Extend the file to the correct size by locating the last block and writing to it. The sample module uses the LOCATE routine to write to the last record. The file is then closed to ensure that the correct EOF marker is set.

❺ **Reopen the file and get the starting block number.** Reopen the file so that data can be written to it. The sample module reopens the file and retrieves its file attributes record to get the starting logical block number and the file's size. The sample checks for the correct starting block number to prevent the disk from being destroyed. If the file is not contiguous, the starting block will be zero. In this case the application should stop immediately.

❻ **Connect to the drive's DDA port.** Use a call to CONNECT_CIRCUIT to connect the previously created message port to the drive's DDA port. The sample module connects the port *access_port* in a circuit to the DDA port *remote_port_name*, where *remote_port_name* is a specified drive name and the string $ACCESS.

❼ **Create a message object.** Call CREATE_MESSAGE to create the message that is to be sent to the DDA port. The sample module creates the message *msg_obj*. The message's size is calculated based on a specified number of blocks.

The module calls the routine *populate_data_buffer* with the argument 1. The routine writes the value 1 to each location in block 1, the value 2 to each location in block 2, and so forth.

❽ **Write data to the disk drive.** Call ELN$DISK_WRITE to write data to the disk drive by calling ELN$DISK_WRITE. You must specify the port connected in a circuit to the drive's DDA port, the write size, the starting logical block number, a variable that is to receive the number of bytes written, the message identifier, and the message pointer. The call to ELN$DISK_WRITE in the sample module writes data of size *total_bytes*, starting at block *starting_lbn*, using port *access_port*. The number of bytes written is returned to *bytes_xfr*. The message is transferred using the message object *msg_obj*.

When the routine returns, *msg_obj* receives the message identifier and *msg_ptr* receives a pointer to the message that is returned by the driver. The sample module uses the returned pointer to set up a pointer to the message data.

The module then calls the routine *populate_data_buffer* with an argument of 0. This routine call initializes all locations in the message data buffer to 0.

❾ **Read data from the disk drive.** Call ELN$DISK_READ to read data from the disk drive. You must specify the port connected in a circuit to the drive's DDA port, the read size, the starting logical block number, a variable that is to receive the number of bytes read, the message identifier, and the message pointer. The call to ELN$DISK_READ in the sample module reads data of size *total_bytes*, starting at block *starting_lbn*, using port *access_port*. The number of bytes read is returned to *bytes_xrf*. The message is transferred using the message *msg_obj*.

Note that the message that was used for the write operation is also used for the read operation. When the routine returns, *msg_obj* receives the message identifier and *msg_ptr* receives a pointer to the message that is returned by the driver. The sample module uses the returned pointer to set up a pointer to the message data.

The module then calls the routine *populate_data_buffer* with an argument of 2. This routine call checks the data read against the data that was written.

⑩ **Clean up resources.** Clean up resources by deleting the message object and its associated buffer, disconnecting the access port from the drive's DDA port, closing the data file, and deleting the access port.

14.1.5.5 Transferring Data to a System Region

You can transfer data to a region of memory at a specified physical address by specifying a system region address in the call to ELN$DISK_READ or ELN$DISK_WRITE. The address you specify must be aligned on a page boundary and must point to the starting location of a system region buffer that was previously allocated by a call to KER$ALLOCATE_SYSTEM_REGION. When you specify a system region address, the data is transferred directly to or from the system region at that address; the message associated with the specified message object is used only for the DDA header.

Example 14–3 shows an example of how you might use the DDA disk interface to read logical blocks of data from an unmounted disk using system virtual address space. The sample module must run in kernel mode.

Example 14–3: Transferring Data to a System Region

```
MODULE xfr_to_physical_adr;

INCLUDE $dda_utility, $KERNEL;

PROGRAM xfr_to_physical_adr(INPUT, OUTPUT);

CONST
  pages_in_ebuild = 6144;
  phy_addr = pages_in_ebuild * 512;
  free_pages_left = 10240 - pages_in_ebuild;

TYPE
  byte = -128..127;
  block = PACKED ARRAY [1..512] OF BYTE;

VAR
  s0_addr : ^ANYTYPE;
  total_blocks : INTEGER;
  device_file : FILE OF BLOCK;
  msg_obj : MESSAGE;
  msg_ptr : ^ANYTYPE;
  bytes_to_read, starting_lbn : INTEGER;
  data_ptr : ^ANYTYPE;
  access_port : PORT;
  drive_name, remote_port_name, file_name : VARYING_STRING (32);
  status, bytes_read : INTEGER;
  s_time, e_time, d_time : LARGE_INTEGER;
  s_time_asc, e_time_asc, d_time_asc : VARYING_STRING(23);

BEGIN { Main program }

  CREATE_PORT(access_port);                                     ❶

  WRITE('Enter the drive name [DUA1]: ');
  READLN(drive_name);
  IF drive_name = '' THEN drive_name := 'DUA1';
  remote_port_name := drive_name + '$ACCESS';

  file_name := drive_name + ':';
  OPEN(device_file,
       FILE_NAME := file_name,
       HISTORY := HISTORY$READONLY);

  CONNECT_CIRCUIT(access_port,
                  DESTINATION_NAME := remote_port_name);

  CREATE_MESSAGE(msg_obj,                                       ❷
   msg_ptr::^byte_data(DDA$_HEADER_SIZE));

  REPEAT
```

Example 14–3 Cont'd on next page

Example 14–3 (Cont.): Transferring Data to a System Region

```
    WRITE('How many blocks are to be read: ');
    READLN(total_blocks);

    bytes_to_read := total_blocks * 512;
    starting_lbn := 0;
    bytes_read := 0;

    KER$ALLOCATE_SYSTEM_REGION(,                                   ❸
                               s0_addr,
                               bytes_to_read,
                               PHYSICAL := phy_addr );

    data_ptr := s0_addr;

    GET_TIME(s_time);
    ELN$DISK_READ(status,                                          ❹
                  access_port,
                  READ_SIZE := bytes_to_read,
                  BLOCK_NUMBER := starting_lbn,
                  BYTES_TRANSFERRED := bytes_read,
                  MSG_OBJ := msg_obj,
                  MSG_PTR := msg_ptr,
                  SYS_REG_ADDR := s0_addr);
    GET_TIME(e_time);
    WRITELN('ELN$DISK_READ status: ', status);
    d_time := e_time - s_time;
    d_time_asc := TIME_STRING(-(d_time));
    WRITELN('Time for ', bytes_read, ' byte transfer -- ', d_time_asc);
    WRITELN('Transfer Rate = ',
            bytes_read DIV (d_time::INTEGER DIV 10000), ' Kb/s');

    {
    { Use the data read from address s0_addr.
    {                   .
    {                   .
    {                   .
    {}

    KER$FREE_SYSTEM_REGION(, bytes_to_read, s0_addr);             ❺

  UNTIL total_blocks = 1;

  DELETE(msg_obj);                                                 ❻
  DISCONNECT_CIRCUIT(access_port);
  CLOSE(device_file);
  DELETE(access_port);
```

Example 14–3 Cont'd on next page

Example 14–3 (Cont.): Transferring Data to a System Region

```
END; { of main program }
END.
```

❶ **Create a port, open the device, and connect to the drive's DDA port.** Create a VAXELN message port and connect that port in a circuit to the drive's DDA port. The sample module creates the message port *access_port* and connects it in a circuit to the DDA port *remote_port_name*, where *remote_port_name* is a specified drive name and the string $ACCESS. The sample module also uses the specified drive name to open the device. The call to OPEN ensures that the device driver sets up the appropriate structures for the data transfer.

❷ **Create a message object.** Create the message that is to be sent to the DDA port by calling CREATE_MESSAGE. The sample module creates the message *msg_obj*. The message's size is equal to the size of the DDA header.

❸ **Allocate necessary system region.** Use a call to the KER$ALLOCATE_SYSTEM_REGION routine to allocate memory in system virtual address space. The memory allocated is physically and virtually contiguous and comes from the system region built into the system. A pointer to the first location of the allocated memory is returned to *s0_addr*. The region size is calculated based on the number of blocks specified in *bytes_to_read*.

❹ **Read data from the disk drive.** Read data from the disk drive by calling ELN$DISK_READ. You must specify the port connected in a circuit to the drive's DDA port, the read size, the starting logical block number, a variable that is to receive the number of bytes read, the message identifier, and the message pointer. The call to ELN$DISK_READ in the sample module reads data of size *bytes_to_read*, starting at block *starting_lbn*, using port *access_port*. The number of bytes read is returned to *bytes_read*. The data is transferred to system region memory, starting at the address specified by *s0_addr*. The message object *msg_obj* is used for the DDA header. The routine returns the identifier and pointer values for the message returned by the driver.

❺ **Free the system region.** Use a call to the KER$FREE_SYSTEM_REGION routine to free memory that was previously allocated with the KER$ALLOCATE_SYSTEM_REGION routine. The call to KER$FREE_SYSTEM_REGION in the sample module frees the number of bytes read starting at the address specified by *s0_addr*.

❻ **Clean up resources.** After one block is read, clean up resources by deleting the message object and its associated buffer (from the last read operation), disconnecting the access port from the drive's DDA port, closing the device file, and deleting the access port. When the cleanup is complete, the module exits.

14.1.6 Virtual-Memory Disk Driver

The VAXELN Toolkit includes a virtual-memory driver (VMDRIVER) that lets you create a virtual RAM disk structure in system memory and use the disk as you would an actual disk drive. You can use the VM disk as a scratch disk for the life of the system. Multiple readers and writers can share the disk and it can participate in network file operations.

The VMDRIVER runs as a job in a VAXELN system. You build the driver into a system by entering the driver's characteristics on the System Builder's Program Description Menu.

Once the VM disk is initialized, you cannot extend it. The memory pages used for the disk are allocated from contiguous addresses in system virtual address space. If insufficient virtual or physical memory is available for the disk, the driver raises the exception appropriate for the missing resource. If the debugger is present in the system, it gains control and shows the specific error message; otherwise, the driver is deleted from the system.

The module in Example 14–4 initializes, mounts, and writes to the virtual-memory disk:

Example 14–4: Using the Virtual-Memory Driver

```
MODULE vm_sample;

INCLUDE $DISK_UTILITY, $FILE_UTILITY;

PROGRAM vm_sample(INPUT, OUTPUT, data_file);

CONST
  cluster_size =    1;
  record_size  = 1024; { Bytes }
  file_size    = 1000; { Blocks }

TYPE
  block_record = PACKED ARRAY[1..record_size] OF CHAR;

VAR
  i, j, m : INTEGER;
  bad_block_list : DSK$_BADLIST(0);
  data_file : FILE OF block_record;
  number_records : INTEGER;
  buffer_size : INTEGER;
  more_data : block_record;
  status : INTEGER;
  cstat : INTEGER;
  cerror : BOOLEAN;
  rdest : VARYING_STRING(255);
  rsource : VARYING_STRING(255);
  file_name : VARYING_STRING(30);
  old_file : VARYING_STRING(255);
  target_file : VARYING_STRING(255);
  volume_name : VARYING_STRING(12);

BEGIN
  file_name := 'DATA.DAT';
  volume_name :='VDISK';
  number_records := 500;

  WRITELN('Initializing virtual disk volume...');
```

Example 14–4 Cont'd on next page

Example 14–4 (Cont.): Using the Virtual-Memory Driver

```
ELN$INIT_VOLUME(DEVICE := 'VM',
                VOLUME := volume_name,
                DEFAULT_EXTENSION := 10,
                USERNAME := 'VAXELN',
                WINDOWS := 7,
                CLUSTER_SIZE := cluster_size,
                INDEX_POSITION := DSK$_BEGINNING,
                DATA_CHECK := DSK$_NOCHECK,
                SHARE := FALSE,
                GROUP := FALSE,
                SYSTEM := FALSE,
                VERIFIED := FALSE,
                BAD_LIST := bad_block_list::DSK$_BADLIST(0);
                STATUS := status);

WRITELN('Mounting virtual disk volume...');

ELN$MOUNT_VOLUME(DEVICE := 'VM',
                 STATUS := status);

buffer_size := 4096;
WRITELN('Opening file on virtual disk...');

OPEN(data_file,
     FILE_NAME := file_name,
     HISTORY := HISTORY$NEW,
     RECORD_LENGTH := record_size,
     RECORD_LOCKING := FALSE,
     ACCESS_METHOD := ACCESS$SEQUENTIAL,
     RECORD_TYPE := RECORD$FIXED,
     CARRIAGE_CONTROL := CARRIAGE$NONE,
     DISPOSITION := DISPOSITION$SAVE,
     SHARING := SHARE$NONE,
     APPEND := TRUE,
     BUFFERING := TRUE,
     BUFFERSIZE := buffer_size,
     EXTENDSIZE := 0,
     FILESIZE := file_size,
     TRUNCATE := FALSE,
     STATUS := status);

more_data[1] := 'A';
more_data[record_size] := 'Z';
```

Example 14–4 Cont'd on next page

Example 14–4 (Cont.): Using the Virtual-Memory Driver

```
  FOR i := 1 TO number_records DO
    WRITE(data_file, more_data);
  CLOSE(data_file);
  target_file := '11.111"name passwd"::log$nam:data.dat';
  old_file := file_name;
  ELN$COPY_FILE(old_file, target_file, cstat, cerror, , ,
                rsource, rdest);
END;
END.
```

For information on how to build the virtual-memory driver into a VAXELN system, see the *VAXELN Development Utilities Guide*.

14.2 Tape Driver

The VAXELN Toolkit includes the tape driver MUDRIVER for TK50 and TK70 magnetic streaming cartridge tape devices and the TU81 reel tape system. This driver also supports all other byte-structured magnetic tape mass storage control protocol (TMSCP) tape drives.

To use the tape interface and drive on a VAXELN target processor, you must include the driver in the VAXELN system that runs on that processor. If you use the supported tape types and driver as supplied, you can regard the driver, and the File Service, as a self-contained program that performs I/O for you. All you need to know in such cases is how to include the driver in your systems. This information is provided in the *VAXELN Development Utilities Guide*.

14.2.1 Logical I/O

Tape file operations use the ANSI file structure. Since you cannot directly read from or write to this type of structure, you cannot use logical I/O with tapes as you can with disks.

14.2.2 Tape Specifications

Table 14–3 lists specifications for the devices that the VAXELN tape driver supports.

Table 14–3: Tape Specifications

Drive	Device	Type	Driver Image
TK50	MU	Streaming cartridge	MUDRIVER.EXE
TK70	MU	Streaming cartridge	MUDRIVER.EXE
TU81	MU	Reel tape system	MUDRIVER.EXE

14.2.3 Tape Driver Interface to the File Service

The VAXELN tape driver, MUDRIVER, includes the tape File Service, which supports the ANSI tape file structure. ANSI is the tape file structure used by VMS. Therefore, you can move tape volumes to a VMS system and use them with VMS software. Also, tapes mounted on VAXELN systems can be used by most VMS file-handling commands when the VAXELN systems are part of a network with VMS systems.

The tape driver uses the File Service to perform the following operations on a tape:

Operation	Description
Open	Prepares a device and its driver for program I/O. The File Service performs this operation when you mount a tape volume or the first time a user program accesses the device.
Get	Asynchronously reads the next block from the tape and returns a context to the read operation.
Put	Asynchronously writes the next block to the tape and returns a context to the write operation.
Reposition	Asynchronously repositions the tape and returns a context to the reposition operation. The File Service performs this operation when a new file is accessed.

Operation	Description
Tapemark	Asynchronously writes a tape mark to the tape and returns a context to the tape mark operation. The File Service performs this operation when a file or the tape is closed.
Return	Provides the status of the completed action of the context given.

14.2.4 Recovery from Power Failure

When a power failure occurs, tapes that are on line and mounted are automatically brought back on line, remounted, rewound to the beginning, and repositioned at the last known position. The device driver reinitializes the tape controller. The File Service operations that were in progress when the power failed are retried, and the tapes can be used again without manual intervention.

14.2.5 Recovery from Errors

Tape mass storage control protocol (TMSCP) devices can detect errors and recover. The only data errors reported are unrecoverable errors, which the driver forwards to the File Service.

14.3 Printer Drivers

The VAXELN Toolkit includes three device driver images that support LP11-type line printers. Table 14–4 lists these drivers with the devices they support.

Table 14–4: Printer Drivers

Driver	Supported Printer Device
LCDRIVER	Printers attached to the parallel printer port of a DMF–32 board
LIDRIVER	Printers attached to the parallel printer port of a DMB32 communications adapter

Table 14–4 (Cont.): Printer Drivers

Driver	Supported Printer Device
LPVDRIVER	Printers attached to an LPV11 printer interface

You can use the parallel port on a DMF–32 for a line printer or for parallel I/O, but not both simultaneously (see Section 14.4.9).

14.3.1 Accessing Printer Devices

You can open a printer device for output by specifying its device name instead of a file specification to the language-specific procedures that open files. Operations on the opened file then apply to the printer.

To use line printer output on a VAXELN target system, you must build the appropriate driver in the VAXELN system that runs on that processor. Several systems in a network can use the printer configured for one node. For instructions on including a line printer driver in a VAXELN system, see the *VAXELN Development Utilities Guide*.

A printer driver generally has one program parameter: the device controller name that you supply with the System Builder. The driver creates the printer unit's local name by appending 0 to the controller name. If you load the driver using a System Builder program description, you can specify a second program argument. The driver uses this argument to create the unit's universal name; again, the driver appends 0 to the argument.

For example, if you specify LPA as the name of a printer controller when you build your system, you can use the local name LPA0: in place of a file specification when opening a file on that node. If you also supply a *universal-name* argument, such as PRINTER, you can use the name PRINTER0 to access the printer from any node.

Alternatively, you can access a printer on a remote node by supplying a node specification in the file specification. However, the use of universal names is more transparent.

If you are printing a file that was opened or created with FORTRAN carriage control, the driver interprets the first character of every line as a carriage control character.

LCDRIVER also initializes the DMF–32 parallel interface for line printer operation, which means that the same DMF–32 cannot be used for parallel I/O.

14.3.2 Printer Driver Characteristics

The source files LCDRIVER.PAS, LPVDRIVER.PAS, and LIDRIVER.PAS define the printer driver characteristics. These characteristics are defined as Pascal named constants. To change a driver's behavior, modify the appropriate constant definitions, recompile the source file, and relink to generate a new driver image.

Table 14–5 summarizes the driver characteristics.

Table 14–5: Printer Driver Characteristics

Characteristic	Description
Maximum record length	The maximum length of single records written to the line printer. The standard value is 512 bytes, or characters.
Lines per page	The number of consecutive lines written on a page before a page eject. The standard value is 66 lines. A user-generated page eject resets the count.
Form-feed/line-feed conversion	A Boolean value that specifies whether the American Standard Code for Information Interchange (ASCII) character FF (form feed) is converted to an equivalent sequence of LFs (line feeds) in the output. The default is FALSE. Use TRUE for printers that do not have a mechanical form-feed feature.
Page width	The maximum number of characters on a printed line. The standard value is 132 characters.
Line wrapping	A Boolean value that specifies whether lines longer than the specified page width are wrapped automatically. The default is FALSE.

Table 14–5 (Cont.): Printer Driver Characteristics

Characteristic	Description
Lowercase-to-uppercase conversion	A Boolean value that specifies whether lowercase characters are converted to uppercase when printed. The default is FALSE. To have all letters printed in uppercase, change the value to TRUE.
Nonprinting character handling	A Boolean value that specifies whether nonprinting characters are allowed in the output. The default is TRUE.
Insertion of CR before LF	A Boolean value that specifies whether the ASCII character CR (carriage return) is inserted before every occurrence of LF (line feed) in the output. (Some printers assume a CR when an LF is printed.) The default is FALSE.

14.4 Terminal Drivers

The VAXELN Toolkit includes device drivers for performing program I/O with console terminals and terminals attached to asynchronous serial communication line interfaces. Table 14–6 lists these drivers with the devices they support.

Table 14–6: Terminal Drivers

Driver	Supported Terminal Devices
CONSOLE	Target processor's console terminal
CXDRIVER	CXA16 and CXB16 devices, which interface up to 16 asynchronous serial lines to a Q-bus on a VAX processor
DECW$CONSOLE	Console emulator for VAXELN DECwindows applications
DECW$TE	VT3*nn* terminal emulator for VAXELN DECwindows applications

Table 14–6 (Cont.): Terminal Drivers

Driver	Supported Terminal Devices
DHVDRIVER	CXY08 device, which interfaces up to 8 asynchronous serial lines to an Industrial VAX processor
	DHQ11 and DHV11 devices, which interface up to 8 asynchronous serial lines to a MicroVAX processor
DHTDRIVER	DHT32 device, which interfaces up to 8 asynchronous serial lines to a MicroVAX 2000 processor
	DSH32 device, which interfaces up to 8 asynchronous serial lines to MicroVAX 2000 and 3100 processors
DZSDRIVER	MicroVAX 2000 integrated serial-line controller, which interfaces up to 4 asynchronous serial lines to a MicroVAX 2000 processor
DZVDRIVER	DZQ11 and DZV11 devices, which interface up to 4 asynchronous serial lines to a MicroVAX processor
DMBDRIVER	DMB32 devices, which interface up to 8 asynchronous serial lines to a VAX processor
LTDRIVER	Serial-line devices attached to terminal servers
RTDRIVER	Remote Terminal Utility, which lets you access a VAXELN target from a remote host and enter commands as if you were connected to a local terminal
SCNDRIVER	User-implemented console or terminal device using the Signetics DUART chip in an rtVAX 300 configuration
YCDRIVER	DMF–32 device, which interfaces up to 8 asynchronous serial lines to a VAX processor

For information about the DECW$CONSOLE and DECW$TE drivers, see the *VAXELN Guide to DECwindows*. Chapter 11 discusses the LAT driver. For information about the Remote Terminal Facility, see Section 9.6.

The VAXELN serial-line drivers are self-contained programs. To use the supported serial-line device types, build the corresponding driver program into your VAXELN system, using the System Builder's Terminal Description Menu or Console Characteristics Menu, as appropriate. If you include a terminal driver or the console driver in a VAXELN system, you may need to increase the pool size allocated for the system to at least 512 blocks. For more information about building terminal drivers and the console driver into VAXELN systems, see the *VAXELN Development Utilities Guide*.

NOTE

> To use the DHVDRIVER for the CXY08 or DHQ11 controller, you must set the onboard mode switch to DHV mode. The factory-default setting for these controllers is DHU mode. If you use a CXY08 or DHQ11 controller for I/O operations while the device is in DHU mode, I/O inconsistencies may occur.

All data transmissions involving terminals are full-duplex transmissions with the same speed, or baud rate, for sending and receiving. In addition, you can use all the supported serial-line interfaces to communicate between remote VAXELN and VMS systems, as discussed in Section 14.4.4.

Other VAXELN drivers support the printer and parallel I/O features of the DMF–32 device. For information about these features, see Sections 14.3 and 14.4.9.

The modules $TERMCLASS and $DDCMP_V2 in library RTLOBJECT.OLB contain several useful support routines for programming terminal drivers. You can use these declarations in your own terminal drivers by including the modules when you compile your driver programs. For details, see terminal driver source files such as DZVDRIVER.PAS, TERMCLASS.PAS, and YCDRIVER.PAS.

Sections 14.4.1 to 14.4.9 discuss a variety of topics concerning the terminal drivers that Digital supplies. Specifically, they discuss terminal I/O, the type-ahead buffer and output synchronization, and direct device access. They also explain how to do the following:

- Terminate lines of input
- Set up point-to-point Digital Data Communications Message Protocol (DDCMP) communication
- Establish circuits for serial-line communication
- Retrieve and set terminal characteristics
- Read data from and write data to serial lines
- Set serial lines to the spacing state
- Use control characters
- Monitor the use of out-of-band characters
- Use modem control

- Use escape and control sequences
- Perform parallel I/O

14.4.1 Terminal I/O

You read input from and write output to a terminal by sending messages to message ports that the Console Driver or other terminal driver creates.

The Console Driver handles transmissions between the program and the console terminal; asynchronous line drivers handle transmissions between the program and one or more terminals attached to asynchronous serial interfaces. For instructions on including terminal drivers in your systems, see the *VAXELN Development Utilities Guide*.

The runtime code for VAXELN procedures, such as the Pascal READ and WRITE procedures, formulates and transmits the necessary messages implicitly when you call these procedures with reference to a terminal.

14.4.2 Type-Ahead and Synchronization

Input characters that you type before a read request are buffered in a *type-ahead* buffer. The type-ahead feature lets you, for example, answer a prompt without waiting for it to appear and usually prevents the loss of characters typed by a fast typist. Input characters remain in the type-ahead buffer until the drivers receive a read request from a program in the application. They are not echoed until then.

If the type-ahead buffer fills up before the drivers get a read request, the drivers sound the bell on the terminal.

The drivers synchronize their output with the terminal by using the XON and XOFF control characters. Therefore, for most applications, you should enable the terminal's AUTO XON/XOFF setting.

14.4.3 Terminating Lines of Input

You terminate lines of input by pressing the Return key or by typing Ctrl/Z or any other character with an ASCII code less than 32 (decimal), except those that have special interpretations as control characters (see Section 14.4.6). When escape recognition is enabled, an entire valid escape sequence is treated as a line terminator. The escape sequence is not echoed and is returned to the program writing the input. This is the only case in which a line terminator also constitutes program input.

14.4.4 Setting Up Point-to-Point DDCMP Communication

You can use the following interfaces for error-free, though not transparent (as Ethernet is), communication between remote VAXELN and VMS systems:

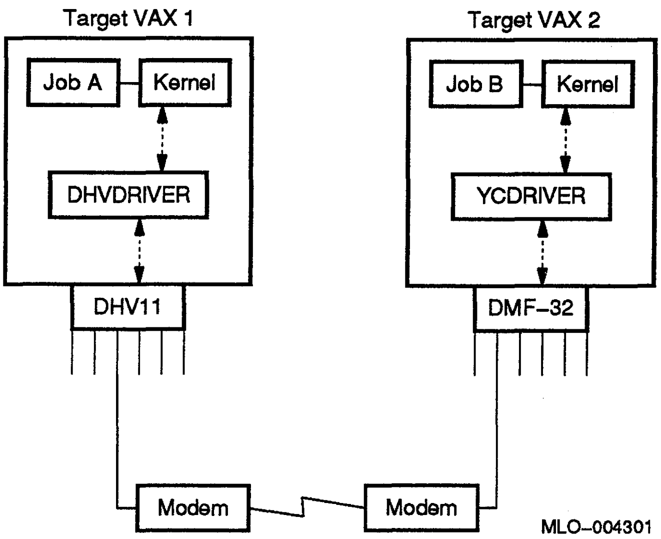
CXA16/CXB16
CXY08,
DHQ11
DHT32
DHV11
DMB32
DMF–32
DZQ11
DZV11

You can establish a virtual circuit between jobs on remote machines over a serial line. To set up such a circuit, let each line act as a full-duplex asynchronous point-to-point Digital Data Communications Message Protocol (DDCMP) communications link.

The DDCMP is a datalink control procedure that ensures a reliable data communications path between communications devices connected by data links. You specify the DDCMP option line-by-line when you build your system, using the System Builder.

Figure 14–1 shows a typical VAXELN serial DDCMP link.

Figure 14–1: A VAXELN Serial DDCMP Link



A job starts the DDCMP protocol on a line by connecting a circuit to the driver handling the line; the job stops the protocol by disconnecting from the circuit. In Figure 14–1, Job B receives messages sent by Job A and Job A receives messages sent by Job B. For example, if Job A uses line TTA2 on a DHV11 interface, part of the Pascal program would be the following:

```
VAR
  data_port : PORT;
  msg : MESSAGE;
  str : ^STRING(512);
   .
   .
   .
  CREATE_PORT(data_port);
  CONNECT_CIRCUIT(data_port, DESTINATION_NAME := 'TTA2');
  CREATE_MESSAGE(msg, str);
   .
   .
   .
  SEND(msg, data_port);
  WAIT_ANY(data_port);
  RECEIVE(msg, str, data_port);
```

On the other end, Job B's program for using line TTX1 on a DMF–32 interface would look like the following:

```
VAR
  data_port : PORT;
  msg : MESSAGE;
  str : ^STRING(512);
   .
   .
   .
  CREATE_PORT(data_port);
  CONNECT_CIRCUIT(data_port, DESTINATION_NAME := 'TTX1');
  CREATE_MESSAGE(msg, str);
   .
   .
   .
  SEND(msg, data_port);
  WAIT_ANY(data_port);
  RECEIVE(msg, str, data_port);
```

The message data can be any data type and can have a length of 1 to 1024 bytes.

The CONNECT_CIRCUIT procedure starts the DDCMP protocol running; the DISCONNECT_CIRCUIT procedure stops it. If the driver determines that the line is down due to excessive errors or retransmissions, it disconnects the circuit. Because this is a full-duplex communications line, both jobs can send messages simultaneously.

The following limitations apply to DDCMP communication:

- Messages received are guaranteed to be received in proper order and error-free. However, due to the nature of the DDCMP protocol, flow control is not as complete or as transparent as for normal circuits. For example, if a job sends enough messages to fill the destination port before the receiving job can call the RECEIVE procedure to receive them, the driver refuses additional messages.

- If the receiver does not receive the messages within a timeout period of approximately 20 seconds (accounting for retransmissions and acknowledgments) the sending driver stops the protocol and disconnects the circuit. To prevent this, the two jobs should synchronize their transmissions so as not to exceed each other's port. The transmission lines are full-duplex, and messages can be overlapped for higher throughput. However, you should avoid prolonged uncontrolled sending of messages.
- Only one virtual circuit is allowed for each line.

14.4.5 Direct Device Access for Serial-Line Devices

Direct device access (DDA) provides an interface for controlling and monitoring serial-line device characteristics at runtime. The VAXELN terminal drivers use this interface. Likewise, user-written programs, including terminal drivers, can use this interface to retrieve and set serial-line characteristics. Additionally, programs in systems that include modem control — DHVDRIVER, DMBDRIVER, or YCDRIVER — can use the interface to monitor modem events.

When the kernel initializes a serial-line terminal driver, it creates a port and a corresponding local port name of the form *line-name*$ACCESS for each serial line. For example, if DHVDRIVER controls the line named TTA1, the kernel creates the local name TTA1$ACCESS. For systems that include an attached console, the kernel names the console port CONSOLE$ACCESS.

The DDA serial-line device interface consists of a set of terminal utility procedures. To use these routines, a program must first connect a circuit to a serial line's DDA port named *line-name*$ACCESS. After the driver accepts the circuit, the program can call the procedures. A program can perform simultaneous DDA operations by connecting multiple circuits to the serial line's DDA port. A program can maintain multiple circuits with a terminal driver for each serial line.

The terminal utility procedures are as follows:

Routine	Description
ELN$TTY_ASSERT_BREAK	Requests that a serial line be set to the spacing state.
ELN$TTY_CANCEL_MODEM_EVENTS	Cancels a request to be notified when a serial line's modem state changes.
ELN$TTY_CANCEL_OOB_CHARACTERS	Cancels a request to be notified when a serial line receives an out-of-band character.
ELN$TTY_GET_CHARACTERISTICS	Returns a serial line's characteristics.
ELN$TTY_READ	Requests that data be read from a serial line.
ELN$TTY_RECEIVE_MODEM_EVENTS	Receives a datagram from the terminal driver containing information about a serial line's modem state changes.
ELN$TTY_RECEIVE_OOB_CHARACTER	Receives a datagram from the terminal driver notifying you that the serial line has received an out-of-band character.
ELN$TTY_SET_CHARACTERISTICS	Sets a serial line's characteristics.
ELN$TTY_SIGNAL_MODEM_EVENTS	Sends a request to the terminal driver to be notified when a serial line's modem state changes.
ELN$TTY_SIGNAL_OOB_CHARACTERS	Sends a request to the terminal driver to be notified when a serial line receives an out-of-band character.
ELN$TTY_WRITE	Requests that data be written to a serial line.

For descriptions of these procedures, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

14.4.5.1 Establishing Circuits for Serial-Line Communication

An application program communicates with a terminal driver using a VAXELN virtual circuit. The program must establish the circuit connection by creating a port and connecting that port to a serial line's DDA port. Once the connection is made, the program can call the terminal utility procedures to get and set terminal characteristics, read and write data, and so forth. The following code fragment connects the port *line1_port* in a circuit to the DDA port named TTA1$ACCESS:

```
MODULE test_terminal;

INCLUDE $DDA_UTILITY;

PROGRAM test_term_characteristics;

VAR
  line1_port, dda_port : PORT;
    .
    .
    .
BEGIN
  CREATE_PORT(line1_port);
  TRANSLATE_NAME(dda_port, 'TTA1$ACCESS', NAME$LOCAL);
  CONNECT_CIRCUIT(line1_port, DESTINATION_PORT := dda_port);
    .
    .
    .
END.
END;
```

Once the connection between *line1_port* and the DDA port is established, the program can call the terminal utility procedures, specifying *line1_port* as an argument.

14.4.5.2 Retrieving and Setting Terminal Characteristics

The VAXELN terminal drivers store a serial line's characteristics in a terminal characteristics record. The record fields define the characteristics listed in Table 14–7.

Table 14–7: Terminal Driver Characteristics

Characteristic	Description
Controller type	The type of asynchronous serial-line controller in use.

Table 14–7 (Cont.): Terminal Driver Characteristics

Characteristic	Description
Speed	For terminals other than the console, the baud rate for transmission and reception is on the indicated line. Valid baud rates include the following:
	50, 134, 600, 2000, 4800, 19200
	75, 150, 1200, 2400, 7200, 38400
	110, 300, 1800, 3600, 9600
	The default for the console and hard-copy terminals is 1200 baud. (The console is assumed to be a hard-copy terminal by default.) The default for CRT terminals is 9600 baud. You must set the terminal to the same speed by using its set-up mode. Not all serial-line devices support the setting of this characteristic.
Parity	A Boolean value that specifies whether parity checking is enabled for the line. The default is FALSE. To enable parity checking, specify TRUE. You must set the terminal to the same value by using its set-up mode. Not all serial-line devices support the setting of this characteristic.
Parity type	For terminals other than the console, the value is DDA$_PARITY_SPACE, DDA$_PARITY_ODD, DDA$_PARITY_EVEN, DDA$_PARITY_MARK, or DDA$_PARITY_IGNORE, which specifies the type of parity checking used by the connected terminal. The default is DDA$_PARITY_EVEN. You must set the terminal to the same parity type by using its set-up mode. Not all serial-line devices support the setting of this characteristic.

Table 14–7 (Cont.): Terminal Driver Characteristics

Characteristic	Description
Display type	The type of terminal in use. HARDCOPY specifies that the terminal is a hard-copy device, such as an LA120 printing terminal; this is the default for the console terminal. SCOPE specifies that the device is a video terminal; this is the default for terminals other than the console. SCOPE causes the DELETE key to backspace and rub out a deleted character; HARDCOPY makes it rewrite a deleted character enclosed in backslashes (\deleted-character\\). DDCMP lines ignore this setting.
Escape recognition	A Boolean value that specifies whether the terminal driver is to check that the format of escape sequences conforms to the American National Standards Institute (ANSI) format. The default is TRUE. To disable escape recognition, specify FALSE. Section 14.4.7 describes the correct formats. In general, if you enable escape recognition for a terminal, you should set the terminal's escape-sequence format to ANSI by using the terminal's set-up mode. DDCMP lines ignore the escape recognition setting.
Echo	A Boolean value that specifies whether the terminal displays (echoes) input lines it receives. The default is TRUE. If the terminal is to display only characters that the software writes to it, specify FALSE. DDCMP lines ignore this setting.
Passall	A Boolean value that specifies whether the terminal driver passes all characters — including tabs, form feeds, control characters, and XON/XOFF — directly from the terminal, without interpretation or translation. The default is FALSE, meaning that special interpretations apply to certain characters (see Section 14.4.6). DDCMP lines ignore this setting.

Table 14–7 (Cont.): Terminal Driver Characteristics

Characteristic	Description
Eight-bit	A Boolean value that specifies whether the attached terminal uses 8-bit ASCII characters. The default is FALSE, in which case the high-order bits of all input characters are masked to 0. To prevent the terminal driver from masking the high-order bit of an input character to 0, specify FALSE. This characteristic determines how software interprets input characters; the bits-per-character setting in a terminal's set-up mode governs the number of bits the terminal displays or prints. This setting is ignored for DDCMP lines.
Character size	The number of bits that comprise a character. Valid sizes are 5, 6, 7, and 8. Not all serial-line devices support the setting of this characteristic.
TTYSYNC	A Boolean value that specifies whether the terminal driver is to respond to XON/XOFF flow control (Ctrl/S and Ctrl/Q) sent from the device to synchronize output written by the system. TRUE is the default. To disable XON/OFF flow control, specify FALSE.
Modem	For terminals other than the console, a Boolean value that specifies whether a serial line is connected to a modem or cable that supplies standard (EIA) modem control signals. The default is FALSE; modem control signals are ignored. You can use modems only with CXY08, DHQ11, DHV11, DMB32, and DMF–32 devices. With the DMF–32 device, only the first two of its eight lines can be used for modems. (See Section 14.4.8.) Not all serial-line devices support the setting of this characteristic.
DDCMP	A Boolean value that specifies whether the line uses DDCMP for asynchronous communication with another system. The default is FALSE; the line acts as a regular terminal line. If the line is to act as a point-to-point full-duplex DDCMP line, specify TRUE. Not all serial-line devices support the setting of this characteristic.

Table 14–7 (Cont.): Terminal Driver Characteristics

Characteristic	Description
Passthru	A Boolean value that specifies whether the terminal driver passes all characters except XON/XOFF directly from the terminal, without interpretation or translation. The default is FALSE, meaning that special interpretations apply to certain characters (see Section 14.4.6). DDCMP lines ignore this setting.

NOTE

If you change the escape recognition, echo, or display type terminal driver characteristic, the change does not take effect during the current read operation. However, the change will take effect for the next read operation. Unless you are using DECW$CONSOLE, DECW$TE, or RTDRIVER, all other terminal driver characteristics that you set by calling ELN$TTY_SET_CHARACTERISTICS take effect immediately, regardless of whether a read operation is in progress.

You can retrieve a terminal's serial-line characteristics by:

- Issuing the ECL command SHOW TERMINAL
- Including a call to the ELN$TTY_GET_CHARACTERISTICS procedure in an application program

Similarly, you can set all or a subset of the serial-line characteristics in the following situations:

- When you build your VAXELN system. If you include a terminal driver in your system, you can specify the characteristics of the terminal on each serial line by editing the System Builder's Terminal Characteristics Menu. You can specify the console terminal's characteristics on the Console Characteristics Menu.
- When using ECL. You can change the character size, echo, 8-bit, escape, parity, passall, speed, scope, and terminal synchronization characteristics by issuing the SET TERMINAL command with the appropriate qualifiers.

- At runtime. You can modify a serial line's characteristics dynamically at runtime by including a call to the ELN$TTY_SET_CHARACTERISTICS procedure in an application program.

The ELN$TTY_GET_CHARACTERISTICS procedure allocates a terminal characteristics record that the application program can access to retrieve terminal characteristics. A call to ELN$TTY_GET_CHARACTERISTICS must specify the port connected in a circuit to the serial line's DDA port and a pointer that points to the serial line's characteristics record. For example:

```
VAR
  line1_port, dda_port : PORT;
  char_record          : ^DDA$_TERMINAL_CHARACTERISTICS;
   .
   .
   .
BEGIN
   .
   .
   .
  { Establish a circuit connection with the serial line's DDA port. }
  CREATE_PORT(line1_port);
  TRANSLATE_NAME(dda_port, 'TTA1$ACCESS', NAME$LOCAL);
  CONNECT_CIRCUIT(line1_port, DESTINATION_PORT := dda_port);
   .
   .
   .
  ELN$TTY_GET_CHARACTERISTICS(CIRCUIT := line1_port,
                              LINE_CHAR_PTR := char_record);
  WITH char_record^ DO
    BEGIN
      WRITELN('Device type = ', DEV_TYPE);
      WRITELN('Revision level = ', REVISION);
   .
   .
   .
    END;
END.
```

This section of code allocates a serial line's characteristics record and then accesses the fields containing the serial line's device type and the revision level of the characteristics record.

An application program can change a serial line's characteristics by calling ELN$TTY_SET_CHARACTERISTICS. The call ELN$TTY_SET_CHARACTERISTICS in the following section of code changes a serial line's passall characteristic to TRUE.

```
VAR
  line1_port, dda_port : PORT;
  char_record          : ^DDA$_TERMINAL_CHARACTERISTICS;
    .
    .
    .
BEGIN
    .
    .
    .
  { Establish a circuit connection with the serial line's DDA port. }
  CREATE_PORT(line1_port);
  TRANSLATE_NAME(dda_port, 'TTA1$ACCESS', NAME$LOCAL);
  CONNECT_CIRCUIT(line1_port, DESTINATION_PORT := dda_port);
    .
    .
    .
  ELN$TTY_GET_CHARACTERISTICS(CIRCUIT := line1_port,
                              LINE_CHAR_PTR := char_record);
    .
    .
    .
  char_record^.PASSALL := TRUE;
  ELN$TTY_SET_CHARACTERISTICS(CIRCUIT := line1_port,
                              LINE_CHAR_PTR := char_record);
    .
    .
    .
END.
```

Before the terminal driver sets a line's characteristics, it checks the values that you supply in the terminal characteristics record to ensure their compatibility with the driver. If you supply an incompatible value, the driver returns an error status and does not set any characteristics. You must then resubmit the request with compatible values. For example, if you specify a line speed that is not available to the driver, the driver returns the ELN$_INVALSPEED status and does not set any characteristics. In this case, you would resubmit the request with a valid line speed.

You should determine whether a serial line's characteristics are set appropriately for your application before modifying the characteristics.

You can also use the ELN$TTY_GET_CHARACTERISTICS and ELN$TTY_SET_CHARACTERISTICS procedures to retrieve and set a terminal's modem characteristics if your VAXELN system includes modem support. For information about using modem control, see Section 14.4.8.1.

The following information sources might also be useful:

- For information about establishing a circuit with a serial line's DDA port, see Section 14.4.5.1.
- For information about specifying terminal characteristics at build time or about using the ECL commands SET TERMINAL and SHOW TERMINAL, see the *VAXELN Development Utilities Guide*.
- For descriptions of the ELN$TTY_SET_CHARACTERISTICS and ELN$TTY_GET_CHARACTERISTICS procedures, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

14.4.5.3 Reading Data from and Writing Data to a Serial Line

Your application programs can read data from and write data to a serial-line device by calling the ELN$TTY_READ and ELN$TTY_WRITE procedures. These procedures read and write characters without interpretation.

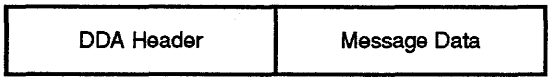
ELN$TTY_READ honors input flow control. However, you can disable input flow control when you build your system or at runtime. You disable flow control at build time by selecting **Yes** for the **Pass all** entry on the System Builder's Terminal Description Menu. You disable it at runtime by using the ELN$TTY_SET_CHARACTERISTICS procedure to change the values in the *passall* and *TTYSYNC* fields of the serial line's terminal characteristics record to TRUE and FALSE, respectively.

Calls to the ELN$TTY_READ and ELN$TTY_WRITE procedures must specify the port connected in a circuit to the serial line's DDA port, a buffer, the transfer request size (in bytes), and a variable that receives the number of bytes of data transferred. The buffer receives the data to be read or contains the data to be written. In calls to ELN$TTY_READ, the buffer size can represent the maximum number of characters that can be read or the number of characters to be read.

You can also specify the ELN$TTY_READ and ELN$TTY_WRITE procedures with arguments that specify a message object and its pointer. ELN$TTY_READ also provides an extended status argument that you can use for reporting character errors.

The message object arguments are input/output arguments that simplify read and write requests. By default, ELN$TTY_READ and ELN$TTY_WRITE create messages, send the messages to the serial-line driver, wait for and receive response messages, and delete the received messages. In the case of read operations, the messages are used to transfer data fragments that are copied to the specified buffer. In the case of write operations, the messages are used to transfer data fragments that are copied from the specified buffer. Rather than having the routines create and delete multiple messages to transfer data, you can create a message to be used in subsequent transfer requests.

To use the message arguments, you must create a message that is large enough to handle the largest possible transfer request and accommodate a DDA header of size DDA$_HEADER_SIZE, prior to calling the ELN$TTY_READ or ELN$TTY_WRITE routine. The following figure shows such a message:



MLO–004168

You can then specify the message's identifier and pointer in the call to ELN$TTY_READ or ELN$TTY_WRITE. ELN$TTY_READ uses messages to send a read request to the driver and to receive the data read. ELN$TTY_WRITE uses messages to send the data being written and to receive the completion status of the write operation.

If an application uses the message arguments — for example, to gain access to the message data, it must ensure that it uses the current values. The message arguments are input/output arguments and the kernel may not map the sent and received messages to the same P0 virtual address space. For example, this might happen if another process in the job runs and either uses some memory or returns memory to the system while the read or write operation is in progress. If this occurs, the message pointer value that the kernel returns might differ from the pointer value of the message that was sent. Similarly, the value of the message identifier might change.

When calling ELN$TTY_READ, you can also specify read options, the minimum number of characters required to complete a read request, a read terminator mask, and a timeout value.

Read options specify a specific type of read operation. The read options are defined as follows:

Option	Description
1	Read a minimum number of characters up to the maximum value
2	Read until a specified timeout value expires
4	Read until a specified character is read

If you do not specify a read option, the procedure reads characters without interpretation while honoring input flow control.

If the default action for ELN$TTY_READ is not appropriate for your application, use the read options to tailor the procedure's action to your needs. You can define a minimum read size by specifying option 1 and an argument representing the minimum number of characters required to complete a read request. The minimum read size that you specify must be less than the buffer size.

You can use the minimum read size option to flush a terminal driver's typeahead buffer. To do this, specify option 1 and a minimum read size of 0. This combination will cause ELN$TTY_READ to read as many bytes as are available in the driver's typeahead buffer, up to a maximum equal to the specified buffer size. If necessary, repeat the read operation until the number of bytes read equals 0 and the status value is odd (success).

You can define a terminator mask by specifying option 4 and an argument that specifies a read terminator mask array. Each element in the mask corresponds to a character in the DEC Multinational Character Set. Setting the value associated with an element to TRUE indicates that the character is to terminate the read operation. If you specify option 4 without specifying a value for the argument, the read terminates when the specified number of characters are read or a specified timeout value expires.

NOTE

When you specify option 4, a read operation may terminate due to two conditions: success (ELN$_SUCCESS) and the receipt of a terminating character (ELN$_TERM_RECV). In this case, the ELN$TTY_READ procedure returns the status value ELN$_TERM_RECV.

If you specify option 2 and a timeout value argument, the ELN$TTY_READ procedure reads data until a time interval expires. You specify the time interval as a time value as shown in the following example:

```
tmo := TIME_VALUE('0 00:02:00.00');
```

If you specify option 2 without specifying a timeout value, the read terminates immediately with as many characters as are available (from the type-ahead buffer) up to the buffer size or a specified terminating character.

You can specify multiple read options by supplying a read option value that is the sum of the desired options. For example, to initiate a read operation that is to use a terminator mask and a timeout value, specify 6 (the sum of options 2 and 4) for the read option.

If an error occurs on a serial-line device during a read operation, the driver terminates the operation and does the following:

- Checks whether the error condition is ELN$_PARITY, ELN$_BREAK_DETECTED, or ELN$_FRAME_ERROR. These conditions are associated with error characters. If one of these conditions occurs, the corresponding error character is stored in the first (low-order) byte of the optional extended status argument.
- Stores the number of good characters read in the number of bytes read argument. If the error occurs while the first character is being read, the value of the number of bytes read argument is 0 even if the specified minimum read size is greater than 0.

Thus, you should check for error conditions and check the value of the number of bytes argument before using data that the ELN$TTY_READ procedure reads.

The program in Example 14–5 reads data from a serial line and writes the characters that are read using the default transfer mechanism. Example 14–6 shows how you might read data to and write data from a serial-line device using the user-defined message transfer mechanism. The discussions that follow refer to the callouts in the examples.

Example 14–5: Reading and Writing Serial-Line Data

```
MODULE readit;

INCLUDE $ELNMSG, $PASCALMSG,
        $KERNELMSG, $GET_MESSAGE_TEXT,
        $DDA_UTILITY, $DDA;

VAR
  dda_packet : ^DDA$_PACKET;       { I/O packet }
  dda_msg : MESSAGE;
  dda_port : PORT;
  app_job_port : PORT;
  status : INTEGER := ELN$_SUCCESS;
  stat : INTEGER := ELN$_SUCCESS;
  options : INTEGER := 0;
  buffer_size : INTEGER;
  buffer : STRING(80);
  nbr_bytes_read : INTEGER := 0;
  nbr_bytes_written : INTEGER := 0;
  terminator_mask : DDA$_BREAK_MASK;
  tmo_value : LARGE_INTEGER := 0;
  min_read_size : INTEGER := 0;
  error_status : DDA$_EXTENDED_READ_STATUS := ZERO;

PROGRAM readit(INPUT,OUTPUT);

VAR
  i : INTEGER := 0;

BEGIN
  JOB_PORT(app_job_port);                                          ❶
  CONNECT_CIRCUIT(app_job_port,
                DESTINATION_NAME := 'TTA0$ACCESS');

  WHILE TRUE DO
    BEGIN
      options := 0;                                                ❷
      buffer_size := 50;
      ELN$TTY_READ(status,
                   app_job_port,
                   buffer_size,
                   buffer,
                   nbr_bytes_read,
                   options,
                   terminator_mask,
                   tmo_value,
                   min_read_size,
                   error_status);
```

Example 14–5 Cont'd on next page

Example 14–5 (Cont.): Reading and Writing Serial-Line Data

```
    WRITELN('Status from first read is: ', status:1);
    WRITELN('Number of characters read is: ', nbr_bytes_read:1);
    WRITELN('Data read was: ', buffer::string(nbr_bytes_read));

    IF ((status = ELN$_PARITY) OR                                          ❸
        (status = ELN$_FRAME_ERROR) OR
        (status = ELN$_BREAK_DETECTED))
    THEN
       WRITELN('Error character is: ', error_status.error_character);

    buffer_size := nbr_bytes_read;                                         ❹
    ELN$TTY_WRITE(status,
                  app_job_port,
                  buffer_size,
                  buffer,
                  nbr_bytes_written);

  END;  { WHILE TRUE }

  DISCONNECT_CIRCUIT(app_job_port);                                        ❺
END;
END.
```

❶ **Connect to a DDA port.** Get the application's job port and connect it in a circuit to a DDA port. The sample module gets the application job port *app_job_port* and connects it in a circuit to the DDA port TTA0$ACCESS.

❷ **Read the data.** Use a call to ELN$TTY_READ to read the data. The call to ELN$TTY_READ must specify the port connected in a circuit to a serial line's DDA port, a buffer, the size of the buffer, and a variable that receives a value indicating the amount of data read. You can also specify options, terminator mask, timeout value, minimum read size, extended status size, message object, and message pointer arguments.

The sample module indicates that no read options will be used, defines a buffer size of 50 bytes, and then issues a call to ELN$TTY_READ. The call to ELN$TTY_READ reads data into a buffer of size 50 bytes and uses *error_status* to receive extended status information. Because no read options are specified, ELN$TTY_READ reads characters without interpretation while honoring input flow control.

❸ **Check for a read operation error.** Check for a parity error, frame error, or break. If one of these conditions occurs, the low byte of the extended status argument will contain the character in error. The sample module checks whether *status* receives ELN$_PARITY, ELN$_FRAME_ERROR, or ELN$_BREAK_DETECTED. If one of these conditions occurs, the sample uses extended status argument *error_status* to write the character in error.

❹ **Write the data.** Use a call to ELN$TTY_WRITE to read the data. The call to ELN$TTY_WRITE must specify the port connected in a circuit to a serial line's DDA port, a buffer, the size of the buffer, and a variable that receives a value indicating the amount of data that is written. You can also specify message object and message pointer arguments. The call to ELN$TTY_WRITE in the sample module writes the data that is in *buffer* to the serial-line device.

❺ **Disconnect the circuit to the DDA port.** Use a call to DISCONNECT_CIRCUIT to disconnect the circuit to the DDA port. The sample module disconnects the circuit between *app_job_port* and the DDA port TTA0$ACCESS.

The numbered callouts that follow refer to the callouts in Example 14–6.

Example 14–6: Reading and Writing Serial-Line Data Using a User-Defined Message

```
MODULE dda_read_with_msg;

INCLUDE $dda_utility;

PROGRAM dda_msg_read(INPUT, OUTPUT);

CONST
  request_size = 5;   { Read in 5 bytes with each read }

VAR
  line_buffer : STRING(request_size);
  my_port : PORT;
  size_read,
  size_written,
  stat : INTEGER;
  message_obj : MESSAGE;
  message_ptr : ^STRING(DDA$_HEADER_SIZE + request_size);

BEGIN
```

Example 14–6 Cont'd on next page

Example 14–6 (Cont.): Reading and Writing Serial-Line Data Using a User-Defined Message

```
  JOB_PORT(app_job_port);                                          ❶
  CONNECT_CIRCUIT(my_port,
    DESTINATION_NAME := 'CONSOLE$ACCESS');

  CREATE_MESSAGE(message_obj,                                      ❷
               message_ptr::^STRING(DDA$_HEADER_SIZE + request_size));

  ELN$TTY_READ(STATUS := stat,                                     ❸
               CIRCUIT := app_job_port,
               BUFFER_SIZE := request_size,
               BUFFER := line_buffer,
               NBR_BYTES_READ := size_read,
               OPTIONS := 0,
               MIN_READ_SIZE := 0,
               MSG_OBJ := message_obj,
               MSG_PTR := message_ptr);

  IF (size_read > 0) THEN                                          ❹
    ELN$TTY_WRITE(STATUS := stat,
                  CIRCUIT := app_job_port,
                  BUFFER_SIZE := size_read,
                  BUFFER := line_buffer,
                  NBR_BYTES_WRITTEN := size_written,
                  MSG_OBJ := message_obj,
                  MSG_PTR := message_ptr);

  DELETE(message_obj);                                             ❺
  DISCONNECT_CIRCUIT(app_port);                                    ❻

END. {program}
END; {module}
```

❶ **Connect to a DDA port.** Get the application's job port and connect it in a circuit to a DDA port. The sample module gets the application job port *app_job_port* and connects it in a circuit to the DDA port CONSOLE$ACCESS.

❷ **Create a message object.** Create the first message that is to be used to transfer data between the application and the serial-line driver by calling CREATE_MESSAGE. The sample module creates the message *message_obj*. The message has a size of 5 bytes plus 512 bytes for the DDA header as indicated by the message pointer argument *message_ptr*.

❸ **Read the data.** Use a call to ELN$TTY_READ to read the data. The call to ELN$TTY_READ must specify the port connected in a circuit to a serial line's DDA port, a buffer, the size of the buffer, and a variable that receives a value indicating the amount of data read. You can also specify options, terminator mask, timeout value, minimum read size, extended status size, message object, and message pointer arguments.

The call to ELN$TTY_READ in the sample module reads data into a buffer of size 5 bytes (*request_size*), using the user-defined message for the data transfer. The *message_obj* and *message_ptr* arguments specify the user-defined message to be used. The routine returns the message identifier and message pointer values of the message returned by the driver. The call to ELN$TTY_WRITE in step 4 specifies these values to reuse the message. Because no read options are specified, ELN$TTY_READ reads characters without interpretation while honoring input flow control.

❹ **Write the data.** Use a call to ELN$TTY_WRITE to read the data. The call to ELN$TTY_WRITE must specify the port connected in a circuit to a serial line's DDA port, a buffer, the size of the buffer, and a variable that receives a value indicating the amount of data that is written. You can also specify message object and message pointer arguments.

In the sample module, ELN$TTY_WRITE echoes the data that is read back to the serial line. ELN$TTY_WRITE writes the number of characters read (*size_read*) from the buffer *line_buffer*, using the user-defined message for the data transfer. The *message_obj* and *message_ptr* arguments specify the message that was returned by the call to ELN$TTY_READ in step 3. The ELN$TTY_WRITE routine returns the message identifier and message pointer values of the message returned by the driver.

❺ **Delete the message.** Use a call to the DELETE procedure to delete the last message received from the driver.

❻ **Disconnect the circuit to the DDA port.** Use a call to DISCONNECT_CIRCUIT to disconnect the circuit to the DDA port. The sample module disconnects the circuit between *app_job_port* and the DDA port CONSOLE$ACCESS.

For information about establishing a circuit with a serial line's DDA port, see Section 14.4.5.1. For descriptions of the ELN$TTY_READ and ELN$TTY_WRITE procedures, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

14.4.5.4 Setting a Serial Line to the Spacing State

An application program that must get the attention of a device attached to a serial line can do so by calling the ELN$TTY_ASSERT_BREAK procedure. This procedure sets a device's serial line to the spacing state. While in the spacing state, the serial line waits for a default or user-specified period of time and asserts a break. The wait ensures that all characters are transmitted to the device.

All VAXELN terminal drivers except CONSOLE, DECW$CONSOLE, DECW$TE, and RTDRIVER support the ELN$TTY_ASSERT_BREAK procedure. When you call the procedure, you must specify the port connected in a circuit to the serial line's DDA port. Optionally, you can specify break options, a break duration period, and a break delay period.

Break options specify the type of break that the serial line is to transmit. The break options are defined as follows:

Option	Description
1	Short break (235 milliseconds)
2	Long break (3.5 seconds)
4	User-specified break duration
8	User-specified break delay

If you do not specify a break option, the serial line transmits a short break after transmitting all current output characters.

If the predefined short and long breaks are not appropriate for your application, you can define your own break. You can define a break duration by specifying option 4 and a break duration period. Likewise, you define a break delay by specifying option 8 and a break delay period. You specify the duration and delay periods as time intervals as shown in the following example:

```
delay := TIME_VALUE('0 00:02:00.00');
```

You can specify multiple break options by supplying a break option value that is the sum of the desired options. For example, to define break duration and delay periods, specify the value 12 (the sum of options 4 and 8) for the break option, as shown in the following example:

```
VAR
  line1_port, dda_port : PORT;
  stat : INTEGER;
    .
    .
    .
BEGIN
    .
    .
    .
  { Establish a circuit connection with the serial line's DDA port. }
  CREATE_PORT(line1_port);
  TRANSLATE_NAME(dda_port, 'TTA1$ACCESS', NAME$LOCAL);
  CONNECT_CIRCUIT(line1_port, DESTINATION_PORT := dda_port);
    .
    .
    .
  ELN$TTY_ASSERT_BREAK(STATUS := stat,
                       CIRCUIT := line1_port,
                       OPTIONS := 12,
                       DURATION := TIME_VALUE('0 00:02:00.00'),
                       DELAY := TIME_VALUE('0 00:00:00.00'));
    .
    .
    .
END.
```

When you specify multiple break options, the terminal driver applies the following precedence to determine the type of break to use:

1. Short break
2. Long break
3. User-defined break

NOTE

When you specify break option 4 or 8, you should also specify a break duration or delay period, as appropriate. If you specify option 4 and omit or specify 0 for the break duration period, the duration period is unpredictable. If you specify break option 8 and omit or specify 0 for the break delay period, the terminal driver does not impose a delay.

For information about establishing a circuit with a serial line's DDA port, see Section 14.4.5.1. For a description of the ELN$TTY_ASSERT_BREAK procedure, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual*.

14.4.5.5 Monitoring the Use of Out-of-Band Characters

VAXELN application programs can use the following terminal utility procedures to monitor the receipt of out-of-band characters:

- ELN$TTY_SIGNAL_OOB_CHARACTERS
- ELN$TTY_RECEIVE_OOB_CHARACTER
- ELN$TTY_CANCEL_OOB_CHARACTERS

A program can instruct a terminal driver to perform special actions based on the use of a specified character in the DEC Multinational Character Set.

Before an application program can call ELN$TTY_SIGNAL_OOB_CHARACTERS, ELN$TTY_RECEIVE_OOB_CHARACTER, and ELN$TTY_CANCEL_OOB_CHARACTERS, the program must establish a circuit with the serial line's DDA port (see Section 14.4.5.1). Once you establish the circuit, the application program can call ELN$TTY_SIGNAL_OOB_CHARACTERS to request that the terminal driver notify the program when a serial-line device receives an out-of-band character. In the routine call, you specify the port connected in a circuit to the DDA port, user data, the response port that is to receive the out-of-band character, and the out-of-band characters for which notification is requested.

The ELN$TTY_SIGNAL_OOB_CHARACTERS routine signals the receipt of an out-of-band on a serial line or the console only if the line is not in the PASSALL or PASSTHRU state; if the line is in one of these states, the routine ignores the out-of-band character. However, if the line is in a temporary PASSTHRU state, the routine will signal the receipt of an out-of-band character.

You specify the out-of-band characters by setting values in an out-of-band character mask. Each element in the mask corresponds to a character in the DEC Multinational Character Set. Setting the value associated with an element to TRUE indicates that the terminal driver is to notify the application when the driver receives that character.

You can also specify out-of-band character options in a call to ELN$TTY_SIGNAL_OOB_CHARACTERS. The options provide more control over the terminal driver's actions and are defined as follows:

Option	Description
1	Signal only once. The request for out-of-band character notification is canceled after the first datagram is sent.
2	Include the out-of-band character in the input stream.

If you do not specify an option, your application program receives all characters that match the specified out-of-band characters and the characters are not placed in the input stream. To specify both options, specify 3, the sum of the two options.

After you call the ELN$TTY_SIGNAL_OOB_CHARACTERS, you must wait on the response port as shown in the following call to WAIT_ANY:

```
WAIT_ANY(response_port
```

A subsequent call to ELN$TTY_RECEIVE_OOB_CHARACTER can then receive an out-of-band character from the terminal driver when the driver receives such a character. The call to this procedure must specify the response port that you specified in the call to ELN$TTY_SIGNAL_OOB_CHARACTERS, user data, and a variable that is to receive the out-of-band character.

The terminal driver sends a notification, in the form of a datagram, to the response port each time it receives an out-of-band character for which notification was requested. If you specified out-of-band character option 1 in the call to ELN$TTY_SIGNAL_MODEM_EVENTS, the driver sends one datagram. Otherwise, the driver sends a separate datagram to your program each time an out-of-band character is received until you cancel the request with a call to ELN$TTY_CANCEL_OOB_CHARACTERS. A call to this procedure must specify the port connected in a circuit to the serial line's DDA port.

A user data argument (in calls to ELN$TTY_SIGNAL_OOB_CHARACTERS and ELN$TTY_RECEIVE_OOB_CHARACTER) lets you pass unmodified user-defined data between your program and the terminal driver. You might use this argument to distinguish different serial lines reporting out-of-band characters to the specified response port.

Example 14–7 shows how you might use the ELN$TTY_SIGNAL_OOB_CHARACTERS, ELN$TTY_RECEIVE_OOB_CHARACTER, and ELN$TTY_CANCEL_OOB_CHARACTERS procedures to monitor the transmission of out-of-band characters over a serial line.

Example 14–7: Monitoring the Use of Out-of-Band Characters

```
MODULE oob;
INCLUDE $elnmsg, $pascalmsg,
        $kernelmsg, $get_message_text,
        $dda_utility, $dda;

VAR
  dda_packet        : ^DDA$_PACKET;          { I/O packet }
  dda_msg           : MESSAGE;
  dda_port          : PORT;              { Port to connect to }
  my_port           : PORT;              { Port for circuit connection}
  status            : INTEGER := ELN$_SUCCESS;
  response_port     : PORT;
  user_data         : INTEGER := 0;
  oob_char_mask     : DDA$_OOB_CHAR_MASK := ZERO;
  oob_char_received : CHAR;

PROGRAM oob (INPUT,OUTPUT);

VAR
  i : INTEGER;

BEGIN

  { Establish a circuit connection with the serial line's DDA port. }
  TRANSLATE_NAME(dda_port, 'TTA1$ACCESS', NAME$LOCAL);
  JOB_PORT(my_port);                 { DDA circuit connection ID }
  CREATE_PORT(response_port,
              limit := 20);          { Response port }
  CONNECT_CIRCUIT(my_port, DESTINATION_PORT := dda_port);

  oob_char_mask[ord('3')] := TRUE;  { Choose '3' and '4' as the        }
  oob_char_mask[ord('4')] := TRUE;  { characters for which we want to }
                                    { be notified.                     }

  { Make the request. }

  ELN$TTY_SIGNAL_OOB_CHARACTERS(status,
                                my_port,
                                user_data,
                                response_port,
                                oob_char_mask);

  { Wait for notification from the driver. }

  WHILE NOT(done) DO
  BEGIN
    WAIT_ANY(response_port);

    { Get the out-of-band character. }
```

Example 14–7 Cont'd on next page

Example 14–7 (Cont.): Monitoring the Use of Out-of-Band Characters

```
    ELN$TTY_RECEIVE_OOB_CHARACTER(status,
                                  response_port,
                                  user_data,
                                  oob_char_received);

    { The received character will either be a 3 or a 4. }

    WRITELN('The received character is: = ',oob_char_received);
  END;

  ELN$TTY_CANCEL_OOB_CHARACTERS(status,
                                my_port);
RETURN:
  DISCONNECT_CIRCUIT(my_port);
END;
END.
```

For information about establishing a circuit with a serial line's DDA port, see Section 14.4.5.1. For a description of the ELN$TTY_CANCEL_OOB_CHARACTERS, ELN$TTY_RECEIVE_OOB_CHARACTER, and ELN$TTY_SIGNAL_OOB_CHARACTERS procedures, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual.*

14.4.6 Using Control Characters

Unless you enable the passall, passthru, or DDCMP terminal characteristic, control characters identify special actions to be performed by the driver rather than actual characters to be sent to a program. Control characters have ASCII codes from 0 to 31 or equal to 127 (DELETE). Table 14–8 lists the control characters with corresponding ASCII codes and action taken.

You generate the characters designated Ctrl/*x*, where *x* is a letter, by holding down the Ctrl key while pressing key *x*.

In some cases, when the echo characteristic is enabled, a Ctrl/*x* character is echoed as a circumflex followed by the letter *x* — for example, ^U for Ctrl/U.

Table 14–8: Control Characters

Terminal Key or Name	Code	Action
Bell	7	Sound the terminal's bell or buzzer.
BACK SPACE	8	Back up the cursor one character. This does not delete the previous character from the input.
TAB, CTRL/I	9	Advance to the next horizontal tab stop. The terminal controls the tab placement.
LINE FEED, CTRL/J	10	Advance to the next line, without a carriage return.
CTRL/K	11	Advance to the next vertical tab stop. The terminal controls the tab placement.
CTRL/L	12	Advance to next page or display (form feed) and terminate the current input line.
NO SCROLL,[1] CTRL/Q	17	Resume transmitting output from the program.
CTRL/R	18	Redisplay the current input line.
NO SCROLL,[1] CTRL/S	19	Suspend transmitting output from the program.
CTRL/U	21	Erase the current input line.
CTRL/Z	26	Designate end-of-file to the program and terminate the current input line.
DELETE	127	Delete the previous character or, if escape recognition is in effect, the partial escape sequence from the input.
ESC	27	Begin escape sequence if the escape recognition characteristic is enabled. Otherwise, echo a dollar sign ($), perform a carriage return and line feed, and terminate the current input line.

[1]The key NO SCROLL, on VT100 and similar terminals, alternates between Ctrl/S (for the first and other odd-numbered keystrokes) and Ctrl/Q.

Table 14–8 (Cont.): Control Characters

Terminal Key or Name	Code	Action
ENTER,[2] RETURN	13	Perform a carriage return and line feed and terminate the current input line.

[2]The key ENTER, on the keypad of VT100 and similar terminals, is normally the same as RETURN.

14.4.7 Using Escape and Control Sequences

When the escape recognition characteristic is enabled and you are using a regular terminal line, you can read escape sequences from a terminal with terminal driver checking syntax. In all cases, whether or not escape recognition is enabled, you can write escape sequences to perform actions specific to the terminal. (For example, the VT100-, VT200-, and VT300-series terminals let you control the cursor's movement with escape sequences.)

The driver checks the syntax of escape sequences only on input and only when escape recognition is enabled. Only ANSI-format escape sequences, such as those used with VT100-, VT200-, and VT300-series terminals, are recognized on input. See the hardware documentation for your terminal for the set of escape sequences used with that terminal.

When escape recognition is enabled, the terminal driver checks any sequence of input characters beginning with the ESC character (ASCII code 27) to determine whether the syntax is correct. An invalid sequence, including the ESC character itself, is effectively removed from the input. Pressing the Delete key in the middle of an escape sequence deletes the entire sequence from the input.

The valid syntax is determined by an ANSI standard as follows (no space should separate the syntax elements):

ESC *character-sequence final-character*

character-sequence
: A sequence of zero or more characters, each of which has an ASCII code in the range 32 to 47. This range consists of the space character and 15 punctuation marks.

final-character
A single character that has an ASCII code in the range 48 to 127, which includes uppercase and lowercase letters, digits, and an assortment of punctuation marks.

The following alternative forms are permitted:

ESC ; *character-sequence final-character*

ESC ? *character-sequence final-character*

ESC O *character-sequence final-character*

With ESC O, the final character can have an ASCII code in the range 64 to 127. The character sequence is the same in all cases. (The 8-bit character SS3 [$8F_{16}$] can introduce an escape sequence in place of ESC O.)

ANSI *control sequences* are also valid. In these sequences, the character sequence and final character are preceded by a left bracket ([ ) and a sequence of parameter specifiers (no space should separate the syntax elements):

ESC [*param-sequence char-sequence final-char*

The 8-bit character CSI [$9B_{16}$] can be used to introduce an escape sequence, in place of ESC [.

param-sequence
Zero or more parameter specifiers, each of which has an ASCII code in the range 48 to 63. For example, for some control sequences on VT100-, VT200-, and VT300-series terminals, this is a sequence of digit characters separated by semicolons.

char-sequence
A sequence of zero or more characters, each of which has an ASCII code in the range 32 to 47.

final-char
A single character, which has an ASCII code in the range 64 to 127.

For example, the following control sequence erases from the current cursor position to the end of the line on a VT100 terminal:

```
ESC[0K
```

The 0 is a parameter and the K is the final character.

The following sequence turns on the bold and reverse video character attributes on a VT100 terminal:

```
ESC[1;7m
```

The 1 and 7 are parameters separated by a semicolon, and m is the final character.

14.4.7.1 Using VT52-Type Escape Sequences

The VT52 terminal uses escape sequences that do not comply fully with the ANSI format. VT100-, VT200-, and VT300-series terminals let you designate, in the terminal's set-up mode, that they will use VT52 escape sequences instead of the larger ANSI set supported on that terminal type.

You should use ANSI escape sequences whenever possible. However, most VT52 escape sequences are compatible with the ANSI syntax and can be recognized if the terminal is set up in VT52 mode.

For example, the following valid sequence erases from the cursor to the end of the screen on a VT52 terminal:

```
ESCJ
```

In ANSI terms, J is the *final-character* and there is no *character-sequence*.

In contrast, the following control sequence, for positioning the cursor to line 2, column 2, is *invalid*:

```
ESC!!
```

Here, the sequence is invalid in ANSI syntax because the *final-character* (!) does not have an ASCII code in the range 48 to 127.

14.4.8 Using Modem Control

Modems let you connect telephone or other remote lines with the terminal interface to access the target computer from remote terminals. The terminal drivers DHVDRIVER, DMBDRIVER, and YCDRIVER support modem control for modems such as the DF03 and DF224 in full-duplex, autoanswer mode. Of the eight asynchronous lines on a DMF–32, only the first two lines can be connected to modems.

You can include modem support in a VAXELN system by selecting *Yes* for the **Modem** entry on the System Builder's Terminal Description Menu when you build your system. For more information, see the *VAXELN Development Utilities Guide*.

A modem is controlled by a set of signals it exchanges with a target computer. The terminal driver transmits and interprets these signals. To be usable, the modem must support all signals listed in Table 14–9.

Table 14–9: Modem Control Signals

Signal Name	Source	Action
TxD (transmitted data)	Computer	Identifies data originated by the computer and transmitted through the modem to one or more remote terminals.
RxD (received data)	Modem	Identifies data generated by the modem, in response to signals received from a remote terminal, and sent to the computer.
RTS (request to send)	Computer	If present, RTS tells the modem to enter transmission mode; if absent, the modem leaves transmission mode after data transmission is complete.
CTS (clear to send)	Modem	If present, CTS tells the computer that the modem is ready to transmit data; if absent, it tells the computer that the modem is not ready.
DSR (data set ready)	Modem	If present, DSR tells the computer that the modem is ready to operate. That is, the modem is connected to the line properly and is ready to exchange more signals. If absent, it tells the computer that the modem is not ready.
CARRIER	Modem	If present, CARRIER tells the computer that the signal received on the data channel line is within the limits specified for the modem. If absent, it tells the computer that the received signal is not within these limits.

Table 14–9 (Cont.): Modem Control Signals

Signal Name	Source	Action
DTR (data terminal ready)	Computer	If present, DTR tells the modem that the computer is ready to operate, prepares the modem for connection to the telephone line, and maintains this connection after it is made. DTR can be present whenever the computer is ready to transmit or receive data; if it is absent, the modem disconnects itself from the line.
RING	Modem	If present, RING tells the computer that a calling signal is being received by the modem (for example, a remote telephone user has dialed the computer's telephone number).

When modem control is enabled for a terminal line, the line is monitored continually by the interface hardware for the RING signal. If the driver detects the CARRIER and DSR signals, the ring is answered whether or not a read request is pending for the line. If the line's CARRIER signal is lost, the driver waits 2 seconds for it to reappear. If it does not, the driver returns an error to any current or future read request.

14.4.8.1 Retrieving and Setting Modem Characteristics

If you build terminal modem support into your VAXELN system, you can use the ELN$TTY_GET_CHARACTERISTICS and ELN$TTY_SET_CHARACTERISTICS procedures to retrieve and set the modem characteristics listed in Table 14–10.

Table 14–10: Modem Characteristics

Characteristic	Description
Modem control	The type of modem control in effect. The modem can be controlled by the terminal driver or the user. The driver controls the modem by default. Not all serial-line devices support the setting of this characteristic.

Table 14–10 (Cont.): Modem Characteristics

Characteristic	Description
RING	A Boolean value that specifies whether the terminal's modem RING indicator is set. You cannot set this characteristic.
CD (carrier detect)	A Boolean value that specifies whether the terminal's modem CD indicator is set. You cannot set this characteristic.
CTS (clear to send)	A Boolean value that specifies whether the terminal's modem CTS indicator is set. You cannot set this characteristic.
DSR (data set ready)	A Boolean value that specifies whether the terminal's modem DSR indicator is set. You cannot set this characteristic.
DTR (data terminal ready)	A Boolean value that specifies whether the terminal's modem DTR indicator is set. To set this characteristic, the modem control characteristic must be set to DDA$_MODEM_CONTROL_USER, and the **Modem** entry on the System Builder's Terminal Description Menu must be set to *Yes*. Not all serial-line devices support the setting of this characteristic.
RTS (request to send)	A Boolean value that specifies whether the terminal's modem RTS indicator is set.

For information about using the ELN$TTY_GET_CHARACTERISTICS and ELN$TTY_SET_CHARACTERISTICS procedures, see Section 14.4.5.2.

14.4.8.2 Monitoring Modem Events

Programs in systems that support modem control (that is, systems that include the terminal driver DHVDRIVER, DMBDRIVER, or YCDRIVER and terminal modem support) can use the terminal utility procedures ELN$TTY_SIGNAL_MODEM_EVENTS, ELN$TTY_RECEIVE_MODEM_EVENTS, and ELN$TTY_CANCEL_MODEM_EVENTS to monitor modem events.

Before an application program can call these procedures, the program must establish a circuit with the serial line's DDA port (see Section 14.4.5.1). Once you establish the circuit, the application program can call ELN$TTY_SIGNAL_MODEM_EVENTS to request that the terminal driver notify the program when a serial line's modem state changes. In the routine call, you specify the port connected in circuit to the DDA port, user data, and the response port that is to receive the modem state change data. You must then wait on the response port as shown in the following call to WAIT_ANY:

```
WAIT_ANY(response_port)
```

A subsequent call to ELN$TTY_RECEIVE_MODEM_EVENTS can then receive modem state change information from the terminal driver. The call to this routine must specify the response port that you specified in the call to ELN$TTY_SIGNAL_MODEM_EVENTS, user data, and a pointer that indicates the record that is to receive the modem state change information.

The terminal driver sends a notification, in the form of a datagram, to the response port each time the modem's state changes. The modem state change information is then stored in a record that identifies the following:

- The revision level of the modem event information record
- The type of modem control in effect (driver or user)
- If the driver is controlling the modem, the modem's current state (connected or disconnected)
- Whether the terminal's modem RING indicator is set
- Whether the terminal's modem CD indicator is set
- Whether the terminal's modem CTS indicator is set
- Whether the terminal's modem DSR indicator is set
- Whether the terminal's modem DTR indicator is set
- Whether the terminal's modem RTS indicator is set

The driver continues to send the modem event information to your program until you cancel the request with a call to ELN$TTY_CANCEL_MODEM_EVENTS. A call to this procedure must specify the port connected in a circuit to the serial line's DDA port.

A user data argument (in calls to ELN$TTY_SIGNAL_MODEM_EVENTS and ELN$TTY_RECEIVE_MODEM_EVENTS) lets you pass unmodified user-defined data between your program and the terminal driver. You might use this argument to distinguish between serial lines reporting modem state changes to the specified response port.

Example 14–8 shows how you might use the ELN$TTY_SIGNAL_MODEM_EVENTS, ELN$TTY_RECEIVE_MODEM_EVENTS, and ELN$TTY_CANCEL_MODEM_EVENTS to monitor a serial line's modem events.

Example 14–8: Monitoring Modem Events

```
MODULE get_modem_events;

INCLUDE $DDA_UTILITY;

PROGRAM getmodem;

VAR
  line1_port         : PORT;
  dda_port           : PORT;
  modem_events_port  : PORT;
  modem_event_ptr    : ^DDA$_MODEM_EVENT_INFORMATION;
  user_data          : INTEGER;
   .
   .
   .
BEGIN
  { Establish a circuit connection with the serial line's DDA port. }

  CREATE_PORT(line1_port);
  CREATE_PORT(modem_events_port);
  TRANSLATE_NAME(dda_port, 'TTA1$ACCESS', NAME$LOCAL);
  CONNECT_CIRCUIT(line1_port,
                  DESTINATION_PORT := dda_port);
   .
   .
   .
  { Request to be signaled when modem events occur on line 1. }

  ELN$TTY_SIGNAL_MODEM_EVENTS(CIRCUIT := line1_port,
                              USER_DATA := user_data,
                              RESPONSE_PORT := modem_events_port);

  { Process all modem change events. }
```

Example 14–8 Cont'd on next page

Example 14–8 (Cont.): Monitoring Modem Events

```
  WHILE TRUE DO
    BEGIN

      { Wait for the driver to signal a mode status change occurrence. }

      WAIT_ANY(modem_events_port);

      { Get the modem status change information. }

      ELN$TTY_RECEIVE_MODEM_EVENTS(RESPONSE_PORT := modem_events_port,
                                   USER_DATA := user_data,
                                   MODEM_EVENT_PTR := modem_event_ptr);
      WITH modem_event_ptr^ do
        BEGIN

          { Do something with the modem state information. For  }
          { example, you might notify the user if the line was  }
          { disconnected.                                       }
    .
    .
    .
        END;
    END;
    .
    .
    .
  { Cancel notification request. }

  ELN$TTY_CANCEL_MODEM_EVENTS(CIRCUIT := line1_port);
    .
    .
    .
END.
END;
```

You can also monitor a serial line's modem characteristics, except the revision level and modem state, by calling ELN$TTY_GET_CHARACTERISTICS. If the modem is user-controlled, you can use the ELN$TTY_SET_CHARACTERISTICS procedure to set the DTR and RTS characteristics. For more information about retrieving modem characteristics, see Sections 14.4.5.2 and 14.4.8.1.

14.4.9 Performing Parallel I/O

You can use the parallel port on a DMF–32 device as a line printer port or to send and receive up to 16 bits of data on 16 parallel lines. The Pascal source file DR11C.PAS, included in the VAXELN Toolkit, contains declarations of the DMF–32 device registers suitable for using the device's parallel port for digital input and output.

You can use the source file DR11C.PAS as a template to write programs that perform parallel I/O. Use the type and variable declarations as delivered and modify the rest of the code to fit your application needs. In some cases, you need to add just a PROGRAM block that uses the declarations the module provides. In addition to the register declarations, the module provides templates for the following:

- An ISR and communication region
- An initialization procedure that creates DEVICE objects representing the device's *request A* and *request B* lines, as well as initializing the parallel port for digital I/O instead of for a line printer
- I/O procedures to read and write a 16-bit word of data from the device

14.5 Small Computer System Interface Driver

The VAXELN Toolkit provides a driver image that supports the American National Standards Institute, Small Computer System Interface (SCSI) devices on MicroVAX, VAXstation, and rtVAXstation 3100 series systems. The image includes a disk class driver (SCSIDISK) and a generic class driver (SCSIGNRC).

- The disk class driver supports RZ22, RZ23, RZ55, and RZ56 Winchester disks, RX23 SCSI diskettes, and RRD40 compact discs.
- The generic class driver provides an interface for all other types of SCSI devices, including scanners, optical devices, test equipment, and medical devices.

VAXELN application programs use a supplied message interface to communicate with the generic class driver.

A system can support up to two SCSI buses and each bus can support up to eight devices: a SCSI host adapter and up to seven device controllers connected to the SCSI bus. An integer in the range 0 to 7 uniquely identifies each of the SCSI devices. Each device controller supports one device unit; that is, VAXELN systems support only logical unit number (LUN) 0.

Only two devices connected to a SCSI bus can communicate on the bus at any given time. On VAXELN systems, the host adapter initiates communication to another device. The target device then performs a task. SCSI devices usually have a fixed role as an initiator or target, although some devices can perform both roles. On VAXELN systems, the host adapter is always the initiator and the device controllers are targets; target devices cannot handle selection operations.

The VAXELN SCSI driver image employs a class/port driver architecture for device communication. The architecture clearly defines class and port driver responsibilities. The class drivers are device-independent and provide standard interfaces to an underlying port driver (see Figure 14–2). The class drivers format commands, interpret status values, and manage user data. The port driver monitors and controls SCSI bus phase changes and sends and receives SCSI path control messages. Using this architecture, you can develop a class driver without regard to the underlying port software and hardware.

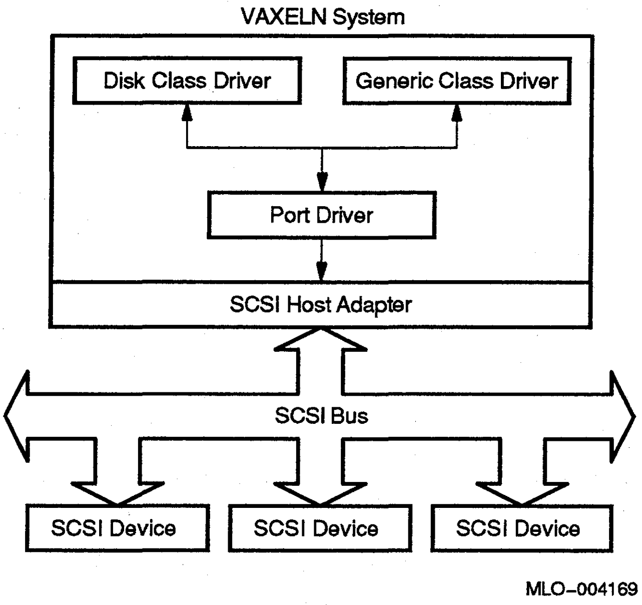
You can use the VAXELN SCSI driver image for third-party SCSI devices that attach to MicroVAX, VAXstation, and rtVAXstation 3100 series systems. The disk class driver supports disk and compact disc devices, while the generic class driver supports all other devices.

You can also combine a user-written SCSI class driver with the supplied VAXELN SCSI port driver to produce a vendor-specific VAXELN SCSI driver image. You can then build that image into a VAXELN system.

NOTE

The *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification allows flexibility for some device implementation details and omits other details. Thus, implementations of the SCSI standard may differ from manufacturer to manufacturer

Figure 14–2: SCSI Class/Port Driver Architecture



and from device to device. Although you can use third-party devices with the VAXELN SCSI disk class driver, the VAXELN Toolkit does not necessarily support such devices.

Digital does not guarantee that third-party devices that currently run with the supplied class driver will continue to run under subsequent releases of the VAXELN Toolkit.

At this writing, the Small Computer System Interface is under development. The draft *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification (Revision 10b) should be the official guide to what a third-party device implements.

To ensure that your third-party device will work properly in a VAXELN environment, Digital encourages the use of an established and supported VAXELN interface, such as those described in Sections 14.5.1 to 14.5.3.

Sections 14.5.1, 14.5.2, and Section 14.5.3 explain how to build third-party SCSI device support into VAXELN systems using the following:

- VAXELN SCSI disk class driver
- VAXELN SCSI generic class driver message interface
- A user-defined class driver

The decision as to which method to use for a particular SCSI device application is left to the application designer. The designer should consider the SCSI device's capabilities, user needs, and available programming resources.

For information about building the SCSI driver into a VAXELN system, see the *VAXELN Development Utilities Guide*.

For more information about Digital's implementation of the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification and how to use the implementation to develop SCSI peripheral devices that are currently available through Digital, see *Small Computer System Interface: An Overview* and *Small Computer System Interface: A Developer's Guide*.

14.5.1 Using the VAXELN SCSI Disk Class Driver

The device-independent design of the VAXELN SCSI disk class driver enables it to control most disk and compact disc device drives that conform to the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification. If your third-party device conforms to the specification, you should consider using the supplied disk class driver for your system.

If you use the supplied SCSI driver image for a third-party disk or compact disc device and your application does not need the generic SCSI device support, you should consider removing the generic device support from the SCSI driver. You can remove device support from the supplied driver by modifying the VAXELN SCSI driver start-up module. For information about modifying this module, see Section 14.5.3.1.

14.5.2 Using the VAXELN SCSI Generic Class Driver

The VAXELN SCSI generic class driver provides support for third-party SCSI devices that do not require file system services. Typical generic SCSI devices include devices such as scanners, optical devices, test equipment, and medical devices.

An application that uses a SCSI generic device communicates with the generic class driver, using a generic class driver message interface. The interface consists of the following runtime routines:

Routine	Description
ELN$SCSI_CONNECT_DEVICE	Connects the application to a SCSI device process.
ELN$SCSI_DISCONNECT_DEVICE	Disconnects the circuit between the application and a SCSI device process.
ELN$SCSI_FREE_CONFIG_DATA	Deletes a message object containing SCSI bus configuration data from the system.
ELN$SCSI_FREE_CONTROL_PORT	Deletes application's source control port from the system.
ELN$SCSI_GET_CONFIG_DATA	Returns SCSI bus configuration data from the generic class driver.
ELN$SCSI_GET_CONTROL_PORTS	Connects the application to the generic class driver and returns the source and destination control ports used to establish the connection.
ELN$SCSI_ISSUE_COMMAND	Delivers a SCSI command to a target SCSI device.
ELN$SCSI_MAP_MESSAGE_BUFFER	Creates a message for sending SCSI commands and data to a SCSI device.
ELN$SCSI_UNMAP_MESSAGE_BUFFER	Deletes a message used to send SCSI commands and data to a SCSI device.

Sections 14.5.2.1 to 14.5.2.4 explain how to use the interface routines to do the following:

- Connect to the generic class driver
- Get configuration data for devices attached to a SCSI bus from the generic class driver
- Connect to SCSI devices
- Issue SCSI commands

To use the message interface routines, you must include the appropriate modules from the VAXELN runtime libraries.

Language	Module
VAXELN Pascal	$SCSI_UTILITY
C	$scsi_utility
FORTRAN	ELN$:FORTRAN_DEFS.FOR

For descriptions of the message interface routines, see the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, or *VAXELN FORTRAN Runtime Library Reference Manual.*

Section 14.5.2.5 shows an example (see Example 14–9) of how you might use the message interface routines to program communication between a SCSI bus and a third-party SCSI generic device.

In addition to programming communication to the generic device driver, you may want to tailor the SCSI driver to your application. That is, if you use the supplied SCSI driver image for a third-party generic device and your application does not need the SCSI disk device support, you should consider removing the disk device support from the SCSI driver. You can remove device support from the supplied driver by modifying the VAXELN SCSI driver start-up module. For information about modifying this module, see Section 14.5.3.1.

14.5.2.1 Connecting to the Generic Class Driver

To use the SCSI generic class driver message interface, an application must first connect to the driver by calling the message interface routine ELN$SCSI_GET_CONTROL_PORTS. This routine creates two control ports — one for the calling job and one for the driver — and establishes a circuit connection between the ports. Once the circuit is established, the application can use it to issue requests for the following:

- Configuration data about the devices attached to the SCSI bus

- Connections to SCSI devices

A call to ELN$SCSI_GET_CONTROL_PORTS must specify the name of a SCSI bus controller and two port variables. The controller name that you specify must match the device name that you specify when configuring the SCSI bus at build time. For example, if you specify the device name DUA, you must specify DUA when you configure the bus with the System Builder.

Source and destination port variables receive the control port values. The source port variable receives the message port value for the calling job. The destination port variable receives the generic class driver port value for the specified device controller.

If your application needs to communicate with devices on two SCSI buses, you might design the application such that communication for devices on each bus is handled by a separate process. In this case, each process would call ELN$SCSI_GET_CONTROL_PORTS to establish a circuit connection for each bus.

When a connection to the generic class driver is no longer needed, the application should free the resources associated with source port by calling ELN$SCSI_FREE_CONTROL_PORT.

The following section of C code shows how you might establish a circuit connection for the bus controller named DUA and free the source port when it is no longer needed:

```
VARYING_STRING_CONSTANT(scsi_port_name, "DUA");
   .
   .
   .
{
  PORT                       source_control_port;
  PORT                       destination_control_port;
  int                        status;
   .
   .
   .
  status = eln$scsi_get_control_ports(&scsi_port_name,
                                      &source_control_port,
                                      &destination_control_port);
   .
   .
   .
  status = eln$scsi_free_control_port(&source_control_port);
   .
   .
   .
}
```

14.5.2.2 Requesting SCSI Bus Configuration Data

Once an application connects to the generic class driver, the application can use the circuit connection to request SCSI bus device configuration data from the driver. The VAXELN SCSI driver stores the configuration data for each SCSI bus in a table and sends that data to applications in a message upon request. The configuration data includes information that the application needs to connect to the devices on the bus.

To retrieve the configuration data, an application must call the ELN$SCSI_GET_CONFIG_DATA routine. Specify the routine with the source and destination ports returned by ELN$SCSI_GET_CONTROL_PORTS for a particular bus. You must also specify a variable that receives a pointer to the SCSI bus configuration table. The table that the application receives includes information about the message that was used to transfer the data and the data for each device attached to the bus.

The message information includes an error code, the message identifier, the size of the message in bytes, and an array that identifies the devices on the bus for which information was returned. The error code indicates whether the request was successful (ELN$_SUCCESS) or unsupported (ELN$_UNSUPPORTED).

Table 14–11 lists the characteristics returned for each device (the host adapter and device controllers).

Table 14–11: SCSI Device Characteristics

Characteristic	Description
Valid data	A flag that indicates whether a device exists for the SCSI bus identifier. If the flag is set to 1, a device is physically attached and the data in the table entry is valid. If the flag is set to 0, the data in the entry is ignored.

Table 14–11 (Cont.): SCSI Device Characteristics

Characteristic	Description
Device type	An integer that identifies the type of peripheral device that is attached to the SCSI bus. The value can be one of the following:

Value	Device Type
0	Direct-access device
1	Sequential-access device
2	Printer device
3	Processor device
4	Write-once, read-multiple device
5	CDROM device
6	Scanner device
7	Optical memory device
8	Medium changer device
9	Communications device
10 to 30	Reserved
31	Unknown or no device type

Characteristic	Description
	You can get the value for a device by using the SCSI INQUIRY command (see the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification).
Class attached	A flag that indicates whether a class driver is assigned to the device. If the flag is set to 1, the device is not available to other class drivers. If the flag is set to 0, the device is available to other class drivers.
Current connection	A flag that informs the generic class driver whether a class driver has made a connection to the device. If the flag is set to 1, the device is currently being used. If the flag is set to 0, the device is not currently being used.
Removable media	A flag that indicates whether the device is removable. If the flag is set to 1, the device is removable. If the flag is set to 0, the device is not removable.

Table 14–11 (Cont.): SCSI Device Characteristics

Characteristic	Description
Product identifier	A 16-byte ASCII text string that identifies the device type. You can get the product identifier for a device by using the SCSI INQUIRY command (see the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification).

To connect successfully to a SCSI device, an application must use the configuration data to determine whether the following conditions exist:

- A device exists for the SCSI bus identifier (valid data is set to 1)
- A class driver is assigned to the device (class attached is set to 1)
- The driver is not connected (current connection is set to 0)

Thus, the valid data and class attached attributes must be set and the current connection attribute must be cleared.

When the configuration data is no longer needed, the application should free the resources used for the configuration data message by calling ELN$SCSI_FREE_CONFIG_DATA.

The following section of C code shows how you might retrieve the configuration data for a SCSI bus, check the configuration data for a direct access device that is not currently connected, and free the configuration data message when it is no longer needed:

```
VARYING_STRING_CONSTANT(scsi_port_name, "DUA");
   .
   .
   .
{
  PORT                    source_control_port;
  PORT                    destination_control_port;
  struct scsi$config_msg  *config_msg_prt;
  int                     scsi_device
  int                     status;
   .
   .
   .
  status = eln$scsi_get_control_ports(&scsi_port_name,
                                      &source_control_port,
                                      &destination_control_port);
  status = eln$scsi_get_config_data(&source_control_port,
                                     &destination_control_port
                                     &config_msg_ptr);

  for (scsi_device = 0; scsi_device < SCSI$K_MAX_UNITS; scsi_device++)
    if ((config_msg_ptr ->
        (config_info.config_tbl[scsi_device].
           valid_data) &&
        (config_msg_ptr->config_info.config_tbl[scsi_device].
           device_type == 0) &&
        (config_msg_ptr->config_info.config_tbl[scsi_device].
           class_attached) &&
        (!config_msg_prt->config_info.config_tbl[scsi_device].
           current_connection))
    break;
   .
   .
   .
  status = eln$scsi_free_config_data(&config_msg_ptr);

  status = eln$scsi_free_control_port(&source_control_port);
   .
   .
   .
}
```

14.5.2.3 Connecting to SCSI Devices

If an application determines from the bus configuration data that the data for a device is valid, a class driver is assigned to the device, and the device is not already connected, the application can connect to it by calling ELN$SCSI_CONNECT_DEVICE. A call to this routine creates a driver process for handling communication for the device and connects that process to your application.

When an application connects to a SCSI device, the call to ELN$SCSI_CONNECT_DEVICE must specify the control ports returned by ELN$SCSI_GET_CONTROL_PORTS for a particular bus, a variable that receives a circuit port value, a process priority, and the SCSI ID for the device to which the application is connecting. The connection request is sent over the circuit connection between the control ports. When the driver receives the request, it creates a process for the specified SCSI device and assigns the specified priority to that process. The driver also creates a message port for the process and connects that port to the application's control port; the new port value is returned to the circuit port argument.

The process priority must be an integer in the range 0 to 15. The highest priority is 0; Digital recommends a priority of 10. If you specify a value that is not in the valid range, the driver uses the default value of 10. A value other than 10 can adversely affect system performance.

If your application needs to communicate with multiple SCSI devices, you might design the application such that communication for each device is handled by a separate application process. In this case, each process would call ELN$SCSI_CONNECT_DEVICE to establish a circuit connection for each device.

When the circuit between the application and the driver's device process is no longer needed, the application should disconnect it by calling ELN$SCSI_DISCONNECT_DEVICE. You must specify the circuit port returned by ELN$SCSI_CONNECT_DEVICE. The routine disconnects the circuit port from the application's circuit port, deletes the device process, returns the PORT and PROCESS objects to the system's kernel object pool, and returns the device to the available list.

The following section of C code connects to a device associated with SCSI ID 3 and disconnects the driver's device process when the connection is no longer needed:

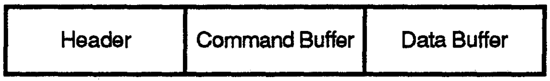
```
VARYING_STRING_CONSTANT(scsi_port_name, "DUA");
   .
   .
   .
{
  PORT                        source_control_port;
  PORT                        destination_control_port;
  PORT                        device_process_port;
  struct scsi$config_msg      *config_msg_prt;
  int                         scsi_device
  int                         status;
   .
   .
   .
  status = eln$scsi_get_control_ports(&scsi_port_name,
                                      &source_control_port,
                                      &destination_control_port);

  status = eln$scsi_get_config_data(&source_control_port,
                                    &destination_control_port
                                    &config_msg_ptr);
```

```
    for (scsi_device = 0; scsi_device < SCSI$K_MAX_UNITS; scsi_device++)
      if ((config_msg_ptr ->
          (config_info.config_tbl[scsi_device].
             valid_data) &&
          (config_msg_ptr->config_info.config_tbl[scsi_device].
             device_type == 0) &&
          (config_msg_ptr->config_info.config_tbl[scsi_device].
             class_attached) &&
          (!config_msg_prt->config_info.config_tbl[scsi_device].
             current_connection))
      break;

    if (scsi_device != SCSI$K_MAX_UNITS)
      {
        status = eln$scsi_connect_device(&source_control_port,
                                         &destination_control_port,
                                         &device_process_port,
                                         10,
                                         scsi_device);
        .
        .
        .
        eln$scsi_disconnect_device(&device_process_port);
      }
      .
      .
      .
    status = eln$scsi_free_config_data(&config_msg_ptr);

    status = eln$scsi_free_control_port(&source_control_port);
      .
      .
      .
  }
```

14.5.2.4 Issuing SCSI Commands

Once an application is connected to a SCSI device, the application can use the connected circuit to issue SCSI commands. Commands are sent to a device in a message that the application creates with a call to ELN$SCSI_MAP_MESSAGE_BUFFER. After creating the message, the application can specify the message identifier in subsequent calls to ELN$SCSI_ISSUE_COMMAND.

SCSI command messages include a header and a command buffer. Optionally, the message can include a buffer for read and write data. The following figure shows the SCSI command message format:



MLO–004171

An application must specify ELN$SCSI_MAP_MESSAGE_BUFFER with variables that are to receive the message object identifier and a pointer to the command buffer. The routine call must also specify the size of the command buffer in bytes. The buffer size cannot exceed 256 bytes.

A call to ELN$SCSI_MAP_MESSAGE_BUFFER can also specify a variable that is to receive a pointer to the data buffer, the size of the data buffer, and a pad size. If you specify the data buffer argument, you must also specify a size for the buffer. The size of the data buffer can range from 1 to 65,536 bytes; 0 bytes is the default.

The pad size argument is for SCSI device commands that require a transfer size that is larger than the size specified by the data buffer size argument. If the amount of data requested in a SCSI command exceeds the space allocated for the data buffer, the pad size accounts for the difference.

For example, the SCSI READ command transfers data in logical blocks — 512-byte units. Suppose an application uses the READ command to read the first two bytes of a disk block. The call to ELN$SCSI_MAP_MESSAGE_BUFFER will specify 2 for the data buffer size to accommodate the two bytes to be read. Since the READ command reads data a block at a time, the call must also specify a pad size of 510 to account for the extra 510 bytes.

Once the message is created, the application can use it to issue SCSI commands, such as INQUIRY, READ, and WRITE. To issue a command, the application must use the ELN$SCSI_ISSUE_COMMAND routine. A call to this routine must specify variables that are to receive the status byte returned by the target device (as defined by the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification) and the status value returned by a SCSI port driver. The status value that the port driver returns indicates whether the command was completed successfully or whether a controller or timeout error occurred.

When issuing a SCSI command, the application must also specify the following:

- A port returned by a call to ELN$SCSI_CONNECT_DEVICE
- The SCSI ID for the device to which the application is issuing the command
- Whether data is being sent or received
- Whether the target device can disconnect during command execution
- Whether the initiator and target devices support synchronous mode for data transfers
- Whether the port driver should attempt to repeat a command that fails due to a timeout, bus parity, or invalid phase transition error
- A phase timeout value
- A disconnect timeout value
- The identifier for the message object created by a call to ELN$SCSI_MAP_MESSAGE_BUFFER
- The message command buffer pointer returned by the call to ELN$SCSI_MAP_MESSAGE_BUFFER

Optionally, the routine call can specify the message data buffer pointer.

The values you can specify for the direction, disconnect, synchronous, and port retry arguments are defined as follows:

Values	**Descriptions**
Direction	
SCSI$K_READ	Target device enters a Data In phase to send data to the initiator.
SCSI$K_WRITE	Target device enters a Data Out phase to receive data from the initiator.

Values	Descriptions
Disconnect	
SCSI$K_DISCONNECT	Target device can disconnect.
SCSI$K_NODISCONNECT	Target device cannot disconnect. Target devices that remain connected to a bus for long periods of time can adversely affect system performance.
Synchronous	
SCSI$K_SYNCH	Initiator and target devices support synchronous mode for data transfers.
SCSI$K_NOSYNCH	The initiator or a target device does not support synchronous mode for data transfers. Currently, the port driver does not support synchronous mode data transfers. Therefore, you must specify SCSI$K_NOSYNCH.
Port Retry	
SCSI$K_RETRY	Port driver can retry a command that fails due to a timeout, bus parity, or invalid phase transition error up to three times.
SCSI$K_NORETRY	Port driver cannot retry a command that fails due to a timeout, bus parity, or invalid phase transition error.

If you do not specify a size for the message data buffer in the call to ELN$SCSI_MAP_MESSAGE_BUFFER, the driver ignores the direction argument.

The phase and disconnect timeout values an application specifies can range from 0 to 420 seconds. The phase timeout value specifies the amount of time a target device has to change to another SCSI bus phase or to complete a data transfer. The disconnect timeout value specifies the amount of time a target device has to reselect an initiator to proceed with a disconnected data transfer. If you specify 0 or an invalid value, the driver uses a timeout value of 20 seconds.

You can use ELN$SCSI_ISSUE_COMMAND to issue commands that are in the Common Command Set (CCS) for direct access devices. For information about these commands, see the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification.

When a SCSI command message is no longer needed, the application should delete the message by calling ELN$SCSI_UNMAP_MESSAGE_BUFFER. You must specify the message identifier returned by the call to ELN$SCSI_MAP_MESSAGE_BUFFER. The routine deletes the message object and returns the resources to the system's kernel pool.

The following section of C code calls a function that creates a SCSI command message and uses it to issue a SCSI INQUIRY command. When the function returns, the module uses the returned data and then deletes the command message.

```
struct inquiry_info {
  unsigned char device_type:5;
  unsigned char perif_qual:3;
  unsigned char resv01:7;
  unsigned char rmb:1;
  unsigned char ansi_ver:3;
  unsigned char ecma_ver:3;
  unsigned char iso_ver:2;
  unsigned char rd_format:4;
  unsigned char resv03:2;
  unsigned char trmiop:1;
  unsigned char aenc:1;
  unsigned char add_length;
  unsigned char resv05;
  unsigned char resv06;
  unsigned char stfre:1;
  unsigned char cmdque:1;
  unsigned char resv07:1;
  unsigned char linked:1;
  unsigned char sync:1;
  unsigned char wbus16:1;
  unsigned char wbus32:1;
  unsigned char reladr:1;
  unsigned char vendor_id[8];
  unsigned char product_id[16];
  unsigned char product_rev[4];
} ;
    .
    .
    .
unsigned char inquiry_cmd[SIX_BYTE_CMD_LENGTH]       = {18,0,0,0,36,0};
    .
    .
    .
{
  PORT                       source_control_port;
  PORT                       destination_control_port;
  PORT                       device_process_port;
  struct inquiry_info        *inquiry_data;
  MESSAGE                    inquiry_msg_obj;
  unsigned char              scsi_status_byte;
  int                        scsi_port_status;
  int                        scsi_device
  int                        status;
   .
   .
   .
  status = get_inquiry_info(&device_process_port,
                              scsi_device,
                              &scsi_status_byte,
                              &scsi_port_status,
                              &inquiry_data,
                              &inquiry_msg_obj);

  /*
   * Use the inquiry data.
   */

  status = eln$scsi_unmap_message_buffer(&inquiry_msg_obj);
   .
   .
   .
}

int get_inquiry_info(PORT          *device_process_port,
                     int           scsi_id,
                     unsigned char *status_byte,
                     int           *port_status,
                     char          *inq_prt,
                     MESSAGE       *inq_obj)

{
  int   i, status;
  char  *scsi_cmd_prt;

  status = eln$scsi_map_message_buffer(inq_obj,
                                         &scsi_cmd_ptr,
                                         sizeof(inquiry_cmd),
                                         inq_ptr,
                                         sizeof(struct inquiry_info),
                                         NULL);

  for(i = 0; i < sizeof(inquiry_cmd); i++)
    scsi_cmd_ptr[i] = inquiry_cmd[i];
  status = eln$scsi_issue_command(
              status_byte,        /* Receives SCSI bus status       */
              port_status,        /* Receives port function status  */
              circuit_port,       /* This job's half of circuit port */
              scsi_id,            /* SCSI bus ID of target device   */
              SCSI$K_READ,        /* Direction = read               */
              SCSI$K_DISCONNECT,  /* Allow disconnects              */
              SCSI$K_NOSYNC,      /* Synchronous disallowed         */
              SCSI$K_RETRY,       /* Port retries allowed           */
              0,                  /* Phase change timeout           */
              0,                  /* Disconnect timeout             */
              inq_obj,            /* Get message object             */
              &scsi_cmd_ptr,      /* Get return pointer to command  */
              inq_ptr);           /* Get return pointer to buffer   */
  return(status);
}
```

14.5.2.5 Programming a Generic Class Driver Message Interface Application

This section provides a sample program that demonstrates how an application might use the SCSI generic class driver message interface (see Example 14–9). The program uses the message interface routines to perform a nondestructive read and write test on a block of data on a direct access SCSI device. The program consists of a main routine and seven functions.

The main routine connects the application to the generic class driver and requests that the driver return bus configuration data for the SCSI bus attached to controller DUA. The program then searches the configuration table for a direct access device that has a class driver attached, but is not connected. If such a device exists, the application connects to it.

After the program connects to a device, it calls the functions *get_inquiry_info*, *go_spinup_drive*, and *go_wait_for_unit_ready*. These functions are defined as follows:

Function	Description
get_inquiry_info	Issues an INQUIRY command.
go_spinup_drive	Brings the disk drive on line. This function calls *issue_start_unit* and *get_request_sense_info*. These functions issue START UNIT and REQUEST SENSE commands, respectively.

Function	Description
go_wait_for_unit_ready	Calls *issue_test_unit_ready* and *get_request_sense_info* functions. These functions issue START UNIT and REQUEST SENSE commands, respectively.

When the device unit is ready, the main routine creates SCSI command messages for reading and writing data and performs a nondestructive read and write operation by calling the function *go_issue_read_write_cmd* five times. The first call to *go_issue_read_write_cmd* issues a READ command. The second call writes the complement of the data read to the write buffer. The third call reads the data that was written to the device; the data is returned in the write buffer. The fourth and fifth calls to *go_issue_read_write_cmd* write the complemented (original) data to and read it from the device.

After performing the read and write operations, the program returns resources to the system by deleting the SCSI command messages, disconnecting the circuit to the driver's device port, deleting the configuration data message, and deleting the application's control port.

Example 14–9: Programming a SCSI Generic Class Driver Message Interface Application

```
#module scsi_msg_class_prg

/*******************************************************************************
 *
 *  This program demonstrates the use of the calls to the SCSI
 *  generic class driver message interface.  It assumes that the
 *  device being used is a direct access device that has read/write
 *  capabilities. Each user interface call is utilized at least once
 *  within the program.  It is not the intent of this program
 *  to perform corrective actions on errors. In most cases, however,
 *  the program will retry error conditions continuously until the
 *  command succeeds or is stopped by the user. The program will attempt
 *  to perform a nondestructive read/write test on one block of data
 *  on the device.
 *
 *******************************************************************************
 */
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
#include $scsi_utility
#include $kernelmsg
#include $elnmsg
#include descrip

#define DIRECT_ACCESS_DEVICE    0
#define SIX_BYTE_CMD_LENGTH     6
#define CHECK_CONDITION         2

union lbn_type {
  unsigned int        num;
  struct {
    unsigned char     lsb:8;
    unsigned char     mid:8;
    unsigned char     msb:8;
    unsigned char     tmsb:8;
  } bits;
};

struct inquiry_info {
  unsigned char device_type:5;
  unsigned char perif_qual:3;
  unsigned char resv01:7;
  unsigned char rmb:1;
  unsigned char ansi_ver:3;
  unsigned char ecma_ver:3;
  unsigned char iso_ver:2;
  unsigned char rd_format:4;
  unsigned char resv03:2;
  unsigned char trmiop:1;
  unsigned char aenc:1;
  unsigned char add_length;
  unsigned char resv05;
  unsigned char resv06;
  unsigned char stfre:1;
  unsigned char cmdque:1;
  unsigned char resv07:1;
  unsigned char linked:1;
  unsigned char sync:1;
  unsigned char wbus16:1;
  unsigned char wbus32:1;
  unsigned char reladr:1;
  unsigned char vendor_id[8];
  unsigned char product_id[16];
  unsigned char product_rev[4];
} ;
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
struct request_sense_info {
  unsigned char error_code;
  unsigned char segment_number;
  unsigned char sense_code;
  unsigned char resv01;
  unsigned char resv02;
  unsigned char resv03;
  unsigned char resv04;
  unsigned char additional_sense_length;
  unsigned char resv05;
  unsigned char resv06;
  unsigned char resv07;
  unsigned char resv08;
  unsigned char additional_sense_code;
  unsigned char additional_sense_code_qualifier;
  unsigned char field_replaceable_unit_code;
  unsigned char resv09;
  unsigned char resv10;
  unsigned char resv11;
  unsigned char resv12;
} ;

VARYING_STRING_CONSTANT(scsi_port_name,"DUA");

$DESCRIPTOR(time_1sec,"0 00:00:01");
LARGE_INTEGER   second_timeout;

unsigned char inquiry_cmd[SIX_BYTE_CMD_LENGTH]          = { 18,0,0,0,36,0 };
unsigned char start_cmd[SIX_BYTE_CMD_LENGTH]            = { 27,1,0,0,1,0  };
unsigned char request_sense_cmd[SIX_BYTE_CMD_LENGTH]    = {  3,0,0,0,19,0 };
unsigned char test_unit_ready_cmd[SIX_BYTE_CMD_LENGTH] = {  0,0,0,0,0,0  };
unsigned char read_cmd[SIX_BYTE_CMD_LENGTH]             = {  8,0,0,0,0,0  };
unsigned char write_cmd[SIX_BYTE_CMD_LENGTH]            = { 10,0,0,0,0,0  };

int get_inquiry_info();
int go_spinup_drive();
int go_wait_for_unit_ready();
int get_request_sense_info();
int issue_start_unit();
int issue_test_unit_ready();
int go_issue_read_write_cmd();
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
scsi_msg_class_test()
{
  PORT                          src_dg_port;
  PORT                          dest_dg_port;
  PORT                          circuit_port;
  struct scsi$config_msg        *config_msg_ptr;
  struct inquiry_info           *inquiry_data;
  MESSAGE                       inquiry_msg_obj;
  unsigned char                 scsi_status_byte;
  int                           scsi_port_status;

  int                           status,i;
  int                           xfer_size;
  int                           pad_size;
  int                           num_of_blocks;

  char                          *rd_cmd;
  char                          *rd_buff;
  MESSAGE                       rd_msg_obj;

  char                          *wrt_cmd;
  char                          *wrt_buff;
  MESSAGE                       wrt_msg_obj;

  int                           test_device;
  int                           block = 100;

  /*
   *  Convert the one-second time string to a LARGE_INTEGER.
   */

  second_timeout = eln$time_value(&time_1sec);

  /*
   *  Get the PORT values associated with this job's message port and the SCSI
   *  generic class driver's message port.  The ports are used for passing
   *  datagrams between this process and the generic class driver.
   */

  status = eln$scsi_get_control_ports(&scsi_port_name,
                                      &src_dg_port,
                                      &dest_dg_port);

  /*
   *  Get the SCSI bus configuration data.  The data is placed in a
   *  configuration aggregate and a pointer to that aggregate is returned to
   *  config_msg_ptr.
   */
```

Example 14–9 Cont'd on next page

```
status = eln$scsi_get_config_data(&src_dg_port,
                                  &dest_dg_port,
                                  &config_msg_ptr);

/*
 *  Search the configuration data for a direct access device that
 *  is not currently connected.
 */

for (test_device = 0; test_device < SCSI$K_MAX_UNITS; test_device++)
  if ((config_msg_ptr->config_info.config_tbl[test_device].valid_data) &&
      (config_msg_ptr->config_info.config_tbl[test_device].device_type ==
                                                 DIRECT_ACCESS_DEVICE) &&
      (config_msg_ptr->config_info.config_tbl[test_device].class_attached) &&
      (!config_msg_ptr->config_info.config_tbl[test_device].
                                                    current_connection) )
  break;

/*
 *  Check whether a SCSI device has been found. If not, exit because
 *  no devices are available for testing.
 */

if (test_device != SCSI$K_MAX_UNITS) {

  /*
   *  Connect the application to the SCSI device found in the configuration
   *  data.  Set the generic class driver process priority to 10.
   */

  status = eln$scsi_connect_device(&src_dg_port,
                                   &dest_dg_port,
                                   &circuit_port,
                                   10,
                                   test_device);

  status = get_inquiry_info(&circuit_port,
                            test_device,
                            &scsi_status_byte,
                            &scsi_port_status,
                            &inquiry_data,
                            &inquiry_msg_obj);

  /*
   *  Use the inquiry data.  When the data is no longer needed,
   *  unmap the SCSI command message buffer to free system resources
   *  and memory.
   */
```

Example 14–9 Cont'd on next page

```
status = eln$scsi_unmap_message_buffer(&inquiry_msg_obj);

/*
 *  Issue a START UNIT command to the specified device.
 */

status = go_spinup_drive(&circuit_port, test_device);

/*
 *  Issue a TEST UNIT READY command to wait for the drive to spin up.
 */

status = go_wait_for_unit_ready(&circuit_port, test_device);

xfer_size = 512;
num_of_blocks = ((xfer_size+511) >> 9); /* Divide by 512. */
pad_size = (num_of_blocks * 512) - xfer_size;

/*
 *  The following code assumes that the read buffer, write buffer,
 *  and SCSI READ and WRITE commands are all of the same size.
 */

/*
 *  Map a buffer for the read request.
 */

status = eln$scsi_map_message_buffer(&rd_msg_obj,
                                     &rd_cmd,
                                     sizeof(read_cmd),
                                     &rd_buff,
                                     xfer_size,
                                     pad_size );

/*
 *  Map a buffer for the write request.
 */

status = eln$scsi_map_message_buffer(&wrt_msg_obj,
                                     &wrt_cmd,
                                     sizeof(write_cmd),
                                     &wrt_buff,
                                     xfer_size,
                                     pad_size );

/*
 *  Initialize the read buffer to all ones.
 */
```

Example 14–9 Cont'd on next page

```
for (i=0; i < xfer_size; i++)
  rd_buff[i] = 0xFF;

/*
 *  Issue a READ command.
 */

status = go_issue_read_write_cmd(&circuit_port,
                                 test_device,
                                 &rd_cmd,
                                 &rd_buff,
                                 &rd_msg_obj,
                                 read_cmd,
                                 sizeof(read_cmd),
                                 num_of_blocks,
                                 block,
                                 SCSI$K_READ);

/*
 *  Write the complement of the data just read into the write buffer.
 */

for (i=0; i < xfer_size; i++)
  wrt_buff[i] = -rd_buff[i];

/*
 *  Issue a WRITE command, using the write buffer.
 */

status = go_issue_read_write_cmd(&circuit_port,
                                 test_device,
                                 &wrt_cmd,
                                 &wrt_buff,
                                 &wrt_msg_obj,
                                 write_cmd,
                                 sizeof(write_cmd),
                                 num_of_blocks,
                                 block,
                                 SCSI$K_WRITE );

/*
 *  Initialize the write buffer.  It will be used for the next read.
 */

for (i=0; i < xfer_size; i++)
  wrt_buff[i] = 0xFF;
```

Example 14–9 Cont'd on next page

```
/*
 *  Read the data just written to the device.  The write buffer
 *  will be used for the return data this time.
 */

status = go_issue_read_write_cmd(&circuit_port,
                                 test_device,
                                 &wrt_cmd,
                                 &wrt_buff,
                                 &wrt_msg_obj,
                                 read_cmd,
                                 sizeof(read_cmd),
                                 num_of_blocks,
                                 block,
                                 SCSI$K_READ );

/*
 *  Write the complement of the data just read.
 */

for (i=0; i < xfer_size; i++)
  wrt_buff[i] = -wrt_buff[i];

/*
 *  Write the complemented data to the device.  This should
 *  be the original data.
 */

status = go_issue_read_write_cmd(&circuit_port,
                                 test_device,
                                 &wrt_cmd,
                                 &wrt_buff,
                                 &wrt_msg_obj,
                                 write_cmd,
                                 sizeof(write_cmd),
                                 num_of_blocks,
                                 block,
                                 SCSI$K_WRITE );

/*
 *  Initialize the write buffer.  It will be used for the next read.
 */

for (i=0; i < xfer_size; i++)
  wrt_buff[i] = 0xFF;
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
    /*
     *  Read the complemented data.  The data should be the original data
     *  read.
     */

    status = go_issue_read_write_cmd(&circuit_port,
                                     test_device,
                                     &wrt_cmd,
                                     &wrt_buff,
                                     &wrt_msg_obj,
                                     read_cmd,
                                     sizeof(read_cmd),
                                     num_of_blocks,
                                     block,
                                     SCSI$K_READ );

    /*
     *  Return the resources back to the system.
     */

    status = eln$scsi_unmap_message_buffer(&rd_msg_obj);
    status = eln$scsi_unmap_message_buffer(&wrt_msg_obj);

    /*
     *  Disconnect the circuit to the generic class driver and return the
     *  associated devices to the available list.
     */

    status = eln$scsi_disconnect_device(&circuit_port);

  } /* End of if (test_device != SCSI$K_MAX_UNITS) */

  /*
   *  Return memory resources and the MESSAGE object back to the pool
   *  associated with the configuration data.
   */

  status = eln$scsi_free_config_data(&config_msg_ptr);

  /*
   *  Delete the PORT assigned to this job by the call to
   *  eln$scsi_get_control_ports.
   */

  status = eln$scsi_free_control_port(&src_dg_port);
}
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
int go_issue_read_write_cmd(PORT            *circuit_port,
                            int             scsi_id,
                            char            **cmd,
                            char            **buffer,
                            MESSAGE         *msg_obj,
                            char            *cmd_ptr,
                            int             cmd_size,
                            int             block_count,
                            int             starting_block,
                            int             direction)
{
  int                           scsi_port_status;
  int                           scsi_port_status1;
  int                           status;
  int                           status1;
  int                           i;
  struct request_sense_data     *request_sense_data;
  MESSAGE                       request_sense_obj;
  unsigned char                 scsi_status_byte;
  unsigned char                 scsi_status_byte1;
  union lbn_type                lbn;

  /*
   *  Copy the READ or WRITE command into the SCSI command message
   *  buffer.
   */

  for (i=0; i < cmd_size; i++)
    (*cmd)[i] = cmd_ptr[i];

  /*
   *  Insert the logical block number and the number of blocks to transfer
   *  into the command.
   */

  lbn.num = starting_block;
  (*cmd)[1] = lbn.bits.msb;
  (*cmd)[2] = lbn.bits.mid;
  (*cmd)[3] = lbn.bits.lsb;
  (*cmd)[4] = block_count;

  do {
    /*
     *  Issue the SCSI READ or WRITE command. The host requests the
     *  target to return read data or sends write data to the target.
     */
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
    status = eln$scsi_issue_command(
                  &scsi_status_byte,      /* Receives SCSI bus status       */
                  &scsi_port_status,      /* Receives port function status  */
                  circuit_port,           /* This job's half of circuit port*/
                  scsi_id,                /* SCSI bus ID of target device   */
                  direction,              /* Direction = read or write      */
                  SCSI$K_DISCONNECT,      /* Allow disconnects              */
                  SCSI$K_NOSYNC,          /* Synchronous disallowed         */
                  SCSI$K_RETRY,           /* Port retries allowed           */
                  0,                      /* Phase change timeout           */
                  0,                      /* Disconnect timeout             */
                  msg_obj,                /* Get message object             */
                  cmd,                    /* Get return pointer to command  */
                  buffer );               /* Get return pointer to buffer   */

    if ( (scsi_port_status == ELN$_SUCCESS) &&
         (scsi_status_byte == CHECK_CONDITION) ) {

      /*
       *  On error, issue a SCSI REQUEST SENSE command to the specified device
       *  to determine the error condition. Refer to the ANSI SCSI specification
       *  for information about this command and its response.
       */

      status1 = get_request_sense_info(circuit_port,
                                       scsi_id,
                                       &scsi_status_byte1,
                                       &scsi_port_status1,
                                       &request_sense_data,
                                       &request_sense_obj );

      /*
       *  Return the request sense data to the system.  Otherwise,
       *  we may use up system resources while waiting for the command
       *  to succeed.
       */

      status1 = eln$scsi_unmap_message_buffer ( &request_sense_obj );
    }

  } while ( (status != KER$_SUCCESS) || (scsi_status_byte) ||
            (scsi_port_status != ELN$_SUCCESS) );

  return(status);

}
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
int get_inquiry_info(PORT          *circuit_port,
                     int           scsi_id,
                     unsigned char *status_byte,
                     int           *port_status,
                     char          **inq_ptr,
                     MESSAGE       *inq_obj)
{
  int                         i,status;
  char                        *scsi_cmd_ptr;

  /*
   *  Map buffers for the SCSI INQUIRY command and for storing
   *  data.
   */

  status = eln$scsi_map_message_buffer(inq_obj,
                                        &scsi_cmd_ptr,
                                        sizeof(inquiry_cmd),
                                        inq_ptr,
                                        sizeof(struct inquiry_info),
                                        NULL);
  /*
   *  Use the pointer to the SCSI command buffer to insert the SCSI
   *  command into the message packet.
   */

  for (i = 0; i < sizeof(inquiry_cmd); i++ )
    scsi_cmd_ptr[i] = inquiry_cmd[i];

  /*
   *  Issue the SCSI INQUIRY command.  The host requests that the target
   *  return inquiry data.
   */
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
  status = eln$scsi_issue_command(
                status_byte,            /* Receives SCSI bus status      */
                port_status,            /* Receives port function status  */
                circuit_port,           /* This job's half of circuit port*/
                scsi_id,                /* SCSI bus ID of target device   */
                SCSI$K_READ,            /* Direction = read               */
                SCSI$K_DISCONNECT,      /* Allow disconnects              */
                SCSI$K_NOSYNC,          /* Synchronous disallowed         */
                SCSI$K_RETRY,           /* Port retries allowed           */
                0,                      /* Phase change timeout           */
                0,                      /* Disconnect timeout             */
                inq_obj,                /* Get message object             */
                &scsi_cmd_ptr,          /* Get return pointer to command  */
                inq_ptr);               /* Get return pointer to buffer   */
  return(status);
}

int go_spinup_drive(PORT *circuit_port, int scsi_id)
{
  int                           scsi_port_status;
  int                           scsi_port_status1;
  int                           status;
  int                           status1;
  unsigned char                 scsi_status_byte;
  unsigned char                 scsi_status_byte1;
  struct request_sense_info     *request_sense_data;
  MESSAGE                       request_sense_obj;

  do {

    /*
     *  Issue the SCSI START UNIT command.
     */

    status = issue_start_unit(circuit_port,
                              scsi_id,
                              &scsi_status_byte,
                              &scsi_port_status);

    if ( (scsi_port_status == ELN$_SUCCESS) &&
         (scsi_status_byte == CHECK_CONDITION) ) {

      /*
       *  On error, issue the SCSI REQUEST SENSE command to determine the
       *  error condition. Refer to the ANSI SCSI specification for
       *  information about the command and its response.
       */
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
      status1 = get_request_sense_info(circuit_port,
                                       scsi_id,
                                       &scsi_status_byte1,
                                       &scsi_port_status1,
                                       &request_sense_data,
                                       &request_sense_obj );

       *  Return the request sense data to the system.  Otherwise,
       *  we may use up system resources while waiting for the command
       *  to succeed.
       */

      status1 = eln$scsi_unmap_message_buffer(&request_sense_obj);
    }

  } while ( (status != KER$_SUCCESS) || (scsi_status_byte) ||
            (scsi_port_status != ELN$_SUCCESS) );

  return(status);
}


int go_wait_for_unit_ready(PORT *circuit_port, int scsi_id)
{
  int                           scsi_port_status;
  int                           scsi_port_status1;
  int                           status;

  int                           status1;
  unsigned char                 scsi_status_byte;
  unsigned char                 scsi_status_byte1;

  struct request_sense_info     *request_sense_data;
  MESSAGE                       request_sense_obj;

  do {

    /*
     *  Issue the SCSI TEST UNIT READY command.
     */

    status = issue_test_unit_ready(circuit_port,
                                   scsi_id,
                                   &scsi_status_byte,
                                   &scsi_port_status);

    if ( (scsi_port_status == ELN$_SUCCESS) &&
         (scsi_status_byte == CHECK_CONDITION) ) {
```

Example 14–9 Cont'd on next page

```
      /*
       *  On error, issue the SCSI REQUEST SENSE command to determine
       *  the error condition.  Refer to the ANSI SCSI specification
       *  for information about this command and its response.
       */

      status1 = get_request_sense_info(circuit_port,
                                       scsi_id,
                                       &scsi_status_byte1,
                                       &scsi_port_status1,
                                       &request_sense_data,
                                       &request_sense_obj);

      /*
       *  Return the request sense data to the system.  Otherwise,
       *  we may use up system resources while waiting for the command
       *  to succeed.
       */

      status1 = eln$scsi_unmap_message_buffer(&request_sense_obj);

      /*
       *  To give the device time to spin up, wait a second before issuing
       *  the TEST UNIT READY command again.  Some devices may take up to a
       *  minute to spin up.
       */

      ker$wait_any(NULL,
                   NULL,
                   &second_timeout);
    }

  } while ( (status != KER$_SUCCESS) || (scsi_status_byte) ||
            (scsi_port_status != ELN$_SUCCESS) );

  return(status);
}

int get_request_sense_info(PORT             *circuit_port,
                           int              scsi_id,
                           unsigned char    *status_byte,
                           int              *port_status,
                           char             **req_ptr,
                           MESSAGE          *req_obj)
{
  int                      i,status;
  char                     *scsi_cmd_ptr;
```

Example 14–9 Cont'd on next page

```
  /*
   *  Map a buffer for the SCSI REQUEST SENSE command and a buffer for
   *  storing the data.
   */

  status = eln$scsi_map_message_buffer(req_obj,
                                       &scsi_cmd_ptr,
                                       sizeof(request_sense_cmd),
                                       req_ptr,
                                       sizeof(struct request_sense_info),
                                       NULL);

  /*
   *  Use the pointer to the SCSI command buffer to insert the SCSI
   *  command into the message packet.
   */

  for (i = 0; i < sizeof(request_sense_cmd); i++ )
    scsi_cmd_ptr[i] = request_sense_cmd[i];

  /*
   *  Issue the SCSI REQUEST SENSE command.  The host requests that the
   *  target return request sense data.
   */

  status = eln$scsi_issue_command(
                 status_byte,           /* Receives SCSI bus status       */
                 port_status,           /* Receives port function status  */
                 circuit_port,          /* This job's half of circuit port*/
                 scsi_id,               /* Device bus ID of target        */
                 SCSI$K_READ,           /* Direction = read               */
                 SCSI$K_DISCONNECT,     /* Allow disconnects              */
                 SCSI$K_NOSYNC,         /* Synchronous disallowed         */
                 SCSI$K_RETRY,          /* Port retries allowed           */
                 0,                     /* Phase change timeout           */
                 0,                     /* Disconnect timeout             */
                 req_obj,               /* Get message object             */
                 &scsi_cmd_ptr,         /* Get return pointer to command  */
                 req_ptr );             /* Get return pointer to buffer   */

  return(status);
}
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
int issue_start_unit(PORT                *circuit_port,
                     int                 scsi_id,
                     unsigned char       *status_byte,
                     int                 *port_status)
{
  int                           i,status;
  char                          *scsi_cmd_ptr;
  MESSAGE                       msg_obj;

  /*
   *  Map a buffer for the SCSI START_UNIT command.
   */

  status = eln$scsi_map_message_buffer(&msg_obj,
                                       &scsi_cmd_ptr,
                                       sizeof(start_cmd),
                                       NULL,
                                       NULL,
                                       NULL);

  /*
   *  Use the pointer to the SCSI command buffer to insert the SCSI
   *  command into the message packet.
   */

  for (i = 0; i < sizeof(start_cmd); i++ )
    scsi_cmd_ptr[i] = start_cmd[i];

  /*
   *  Issue the START_UNIT command. The SCSI status byte, port status, and
   *  message status are returned from the device.
   */

  status = eln$scsi_issue_command(
              status_byte,            /* Receives SCSI bus status       */
              port_status,            /* Receives port function status  */
              circuit_port,           /* This job's half of circuit port*/
              scsi_id,                /* Device bus ID of target        */
              SCSI$K_READ,            /* Direction = read               */
              SCSI$K_DISCONNECT,      /* Allow disconnects              */
              SCSI$K_NOSYNC,          /* Synchronous disallowed         */
              SCSI$K_RETRY,           /* Port retries allowed           */
              0,                      /* Phase change timeout           */
              0,                      /* Disconnect timeout             */
              &msg_obj,               /* Get message object             */
              &scsi_cmd_ptr,          /* Get return pointer to command  */
              NULL );                 /* No buffer needed for command   */
```

Example 14–9 Cont'd on next page

```
  if (status != KER$_SUCCESS)
    return(status);

  /*
   *  Return the resource back to the system.  Otherwise, we may use
   *  up system resources.
   */

  status = eln$scsi_unmap_message_buffer(&msg_obj);

  return(status);
}

int issue_test_unit_ready(PORT               *circuit_port,
                          int                scsi_id,
                          unsigned char      *status_byte,
                          int                *port_status)
{
  int                        i,status;
  char                       *scsi_cmd_ptr;
  MESSAGE                    msg_obj;

  /*
   *  Map a buffer for the SCSI TEST UNIT READY command.
   */

  status = eln$scsi_map_message_buffer(&msg_obj,
                                       &scsi_cmd_ptr,
                                       sizeof(start_cmd),
                                       NULL,
                                       NULL,
                                       NULL);

  /*
   *  Use the pointer to the SCSI command buffer to insert the SCSI
   *  command into the message packet.
   */

  for (i = 0; i < sizeof(test_unit_ready_cmd); i++ )
    scsi_cmd_ptr[i] = test_unit_ready_cmd[i];

  /*
   *  Issue the SCSI START_UNIT command.  The SCSI status byte, port
   *  status, and message status are returned from the device.
   */
```

Example 14–9 Cont'd on next page

Example 14–9 (Cont.): Programming a SCSI Generic Class Driver Message Interface Application

```
  status = eln$scsi_issue_command(
              status_byte,           /* Receives SCSI bus status       */
              port_status,           /* Receives port function status  */
              circuit_port,          /* This job's half of circuit port*/
              scsi_id,               /* Device bus ID of target        */
              SCSI$K_READ,           /* Direction = read               */
              SCSI$K_DISCONNECT,     /* Allow disconnects              */
              SCSI$K_NOSYNC,         /* Synchronous disallowed         */
              SCSI$K_RETRY,          /* Port retries allowed           */
              0,                     /* Phase change timeout           */
              0,                     /* Disconnect timeout             */
              &msg_obj,              /* Get message object             */
              &scsi_cmd_ptr,         /* Get return pointer to command  */
              NULL );                /* No buffer needed for command   */

  if (status != KER$_SUCCESS)
    return(status);

  /*
   *  Return the resource for this command to the system.  Otherwise, we
   *  may use up system resources.
   */

  status = eln$scsi_unmap_message_buffer(&msg_obj);

  return(status);
}
```

14.5.3 Developing User-Defined SCSI Class Drivers

If the supplied VAXELN SCSI class drivers do not provide the device support that you need, you can create an application-specific SCSI class driver that communicates with the VAXELN SCSI port driver. Before developing such a driver, you should be familiar with the SCSI driver components and understand how they communicate.

Table 14–12 describes the components of the VAXELN SCSI driver.

Table 14–12: VAXELN SCSI Driver Components

Component	Description	
Start-up module	SCDRIVER.C	Associates class drivers with supported device types and starts the class drivers.
Sniffer module	SCSISNIF.C	Searches a bus for available devices, sets up a bus configuration table based on the information it finds, and informs the class drivers about devices they are to service.
Class drivers	SCSIDISK.C SCSIGNRC.C	Format commands, interpret status values, and manage user data.
Port driver	SCSI5380.C	Monitors and controls SCSI bus phase changes and sends and receives SCSI path control messages.
Data structure and constant definitions	$SCSI_UTILITY.H $SCSIPORT.H	Define the data structures and constants used by the SCSI driver modules. These definition files are in the definition module **vaxelnc**.

The start-up module specifies the device types that each class driver is to service. For example, the supplied start-up module associates the disk class driver with direct-access and CDROM devices and the generic class driver with all other devices. The sniffer module uses this information when it determines which devices are to service available SCSI devices.

The sniffer module checks whether a device is attached to a SCSI bus at each of the eight SCSI device IDs and builds a configuration table based on the information that it finds. The configuration table consists of eight entries, one for each SCSI device ID. Table 14–13 lists the types of data that each table entry provides.

Table 14–13: SCSI Bus Configuration Data

Data	Description
Valid data	A flag that indicates whether a device exists for the SCSI bus identifier. If the flag is set to 1, a device is physically attached and the data in the table entry is valid. If the flag is set to 0, the data in the entry is ignored.
Device type	An integer that identifies the type of peripheral device that is attached to the SCSI bus. The value can be one of the following:

Value	Device Type
0	Direct-access device
1	Sequential-access device
2	Printer device
3	Processor device
4	Write-once, read-multiple device
5	CDROM device
6	Scanner device
7	Optical memory device
8	Medium changer device
9	Communications device
10 to 30	Reserved
31	Unknown or no device type

You can get the value for a device by using the SCSI Inquiry command (see the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification).

Data	Description
Class attached	A flag that indicates whether a class driver is assigned to the device. If the flag is set to 1, the device is not available to other class drivers. If the flag is set to 0, the device is not available to other class drivers.
Current connection	A flag that indicates whether a class driver has made a connection to the device. If the flag is set to 1, the device is currently being used. If the flag is set to 0, the device is not currently being used.

Table 14–13 (Cont.): SCSI Bus Configuration Data

Data	Description
Removable media	A flag that indicates whether the device is removable. If the flag is set to 1, the device is removable. If the flag is set to 0, the device is not removable.
Product identifier	A 16-byte ASCII text string that identifies the device type. You can get the product identifier for a device by using the SCSI INQUIRY command (see the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification).

The sniffer module uses the valid data and device type information to create a device marker for each class driver. The device marker is an 8-element array that identifies which devices on the bus the class driver is to service. If a configuration table entry contains valid data, the sniffer module compares the device type with the device types defined for each class driver's in the start-up module. If a match exists for a class driver, the sniffer module places a value of 1 in the appropriate field of that class driver's device marker.

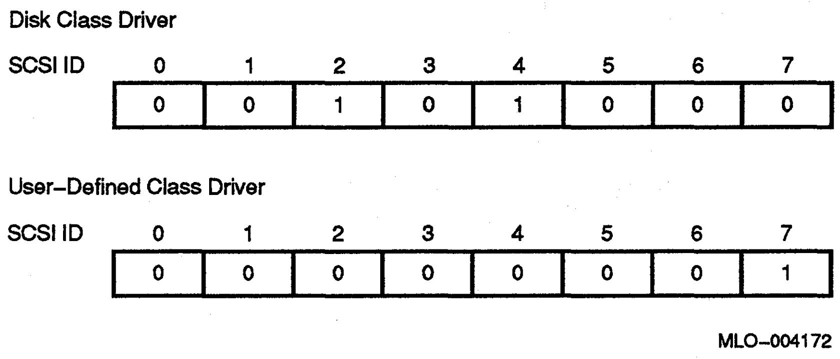
Consider the following scenario:

- The start-up module associates the supplied disk class driver with direct-access devices (0) and CDROM devices (5) and associates a user-defined class driver with a scanner device (6).
- The sniffer module finds valid data in the configuration table entries for SCSI devices 2, 4, and 7.
- SCSI device 2 is a direct-access device (device type 0), SCSI device 4 is a CDROM device (device type 5), and SCSI device 7 is a scanner device (device type 6).

Based on this information, the sniffer module will create the device markers shown in Figure 14–3 for the disk and user-supplied class drivers.

The sniffer module then passes the device marker and configuration table information to the class drivers.

Figure 14–3: SCSI Device Markers



To develop a user-written class driver for inclusion in a VAXELN system, you must do the following:

- Modify the SCSI driver start-up module
- Program the class driver
- Compile and link the SCSI driver modules

Sections 14.5.3.1 to 14.5.3.3 explain how to modify the start-up module, program a driver, and compile and link the driver modules, respectively.

14.5.3.1 Modifying the SCSI Driver Start-Up Module

To add user-written class driver support to the VAXELN SCSI driver, you must edit the SCSI driver's start-up module. This module associates your class driver with the device types it supports.

Example 14–10 shows the start-up module that the toolkit supplies. Callouts identify the lines in the module that you need to edit. The list accompanying the example explains the changes that you need to make.

❶ **Declares the class driver source.** The supplied start-up module declares the driver source for the disk class driver, *scsidisk*, and

Example 14–10: Modifying the SCSI Driver Start-Up Module

```
#module scdriver /* SCSI start-up module */

#include $vaxelnc

extern void scsi_class$start();
extern void scsidisk();                                                  ❶
extern void scsignrc();

struct scsi_descriptor_type {
    int (*class_start)();
    int dev_type[8];
} ;

struct scsi_descriptor_type disk_class = { scsidisk, 0, 5, -1};        ❷
struct scsi_descriptor_type generic_class = { scsignrc, -1 };

void scsi$start()
{
    scsi_class$start(&disk_class, &generic_class);                     ❸
}
```

the generic class driver, *scsignrc*. If your application does not need the supplied class drivers, you can delete the corresponding declarations.

To add support for your user-defined class driver, add the appropriate declaration. For example:

```
extern void scsiuser();
```

❷ **Associates the class driver with supported device types.** The descriptors 0 and 5 in the first structure definition identify direct access and CDROM devices, respectively. The −1 flag signifies the end of the list of supported device types for the class driver. When the −1 flag is the first element of the descriptor list, as is the case in the second structure definition, the driver is generic and supports all other valid SCSI devices whose descriptors are not listed in the structure definition.

If your VAXELN application does not need the supplied class drivers, you can delete the corresponding structure definitions.

To add support for your user-defined class driver, add the appropriate structure definition. For example:

```
struct scsi_descriptor_type user_class = { scsiuser, 6, -1 }
```

This structure definition adds the class driver SCSIUSER, which supports scanner devices (device type 6).

❸ **Starts the class drivers.** The supplied start-up module starts the supplied disk and generic drivers. If your VAXELN application does not need the supplied class drivers, delete the appropriate pointers. To add support for your user-defined class driver, add a pointer to your class driver. For example:

```
scsi_class$start(&disk_class, &user_class, &generic_class);
```

NOTE

The generic class driver provides support for all SCSI devices in the configuration table that do not have a class driver attached. Therefore, if you include the generic SCSI class driver, you must list it as the last driver in the call to **scsi_class$start**.

For information about recompiling and relinking the start-up module, see Section 14.5.3.3.

14.5.3.2 Programming SCSI Class Drivers

SCSI class drivers communicate with the VAXELN SCSI port driver by using a port driver interface that includes the following routines:

Routine	**Description**
PORT$ALLOCATE_DEVICE	Reserves an I/O request packet for a device.
PORT$FREE_DEVICE	Returns an I/O request packet to the free list.
PORT$INITIALIZE_CONTROLLER	Initializes a SCSI bus controller.
PORT$ISSUE_COMMAND	Issues a SCSI command.
PORT$MAP_BUFFER	Maps consecutive SCSI DMA RAM bit map data bytes for an I/O request packet.
PORT$UNMAP_BUFFER	Unmaps consecutive SCSI DMA RAM bit map data bytes used by an I/O request packet.

The sniffer module declares a routine address structure that the class drivers use to call these routines. The structure is declared as follows:

```
struct routine_addresses {
  char *ctx_a_context;
  int (*ctx_a_init) ();
  int (*ctx_a_issue) ();
  int (*ctx_a_alloc) ();
  int (*ctx_a_free) ();
  int (*ctx_a_map) ();
  int (*ctx_a_unmap) ();
  int (*ctx_a_exit) ();
};

struct contxt routine_addresses;
```

The structure contains a pointer to the port driver's data structures and the addresses of the entry points for the interface routines. After starting the port driver, the sniffer module fills in the correct addresses. The structure elements are defined as follows:

Element	Description
ctx_a_context	A pointer to the port driver's data structures. The first argument in calls to the port driver interface routines must specify this context pointer.
ctx_a_init	The address of the entry point for the PORT$INITIALIZE_CONTROLLER routine. The sniffer module calls this routine to initialize the bus controller. Class drivers should not use this routine.
ctx_a_issue	The address of the entry point for the PORT$ISSUE_COMMAND routine.
ctx_a_alloc	The address of the entry point for the PORT$ALLOCATE_DEVICE routine.
ctx_a_free	The address of the entry point for the PORT$FREE_DEVICE routine.
ctx_a_map	The address of the entry point for the PORT$MAP_BUFFER routine.
ctx_a_unmap	The address of the entry point for the PORT$UNMAP_BUFFER routine.
ctx_a_exit	The address of the entry point for the PORT$EXIT_HANDLER routine.

A class driver calls (invokes) a port driver interface routine by specifying the appropriate routine address, the context pointer, and the routine's arguments. The format you use for calling the routines in C follows:

status = (*routine_addresses.ctx_a_*port-routine*)
(routine_addresses.ctx_a_context,
routine-argument, . . .);

The following example shows how a class driver written in C might call the PORT$ISSUE_COMMAND routine:

```
status = (*routine_addresses.ctx_a_issue)
         (routine_addresses.ctx_a_context,
          virtual_device,
          DISCONNECT,
          0,
          0,
          0);
```

A class driver written in Pascal must use calls to the INVOKE function to invoke the port driver interface routines. Prior to invoking an interface routine, the driver must declare a function type for the routine. The call to INVOKE then specifies a pointer to the routine's entry mask, the name of the function type, and the routine's arguments. For more information about using the port driver interface routines from Pascal, see Appendix D.

Prior to calling the interface routines, a FORTRAN class driver must declare the variables *routine_addresses* and *lock_device* as external data, using the EXTERNAL statement as follows:

```
EXTERNAL routine_addresses
EXTERNAL lock_device
```

These statements ensure that the symbols are resolved such that they are the addresses for the SCSI port interface callback routines as declared in the VAX C global definition (globaldef) storage class.

One way a FORTRAN driver can gain access to the interface routines, given their addresses, is to do the following:

- Pass the external variable *routine_addresses* to a subroutine that declares the variable as a RECORD /PORT_ROUTINES/. This enables the driver to access the necessary fields of the *routine_addresses*.
- As appropriate, pass a routine address (for example, *routine_addresses.alloc*) by value to another subroutine that redeclares the address to be EXTERNAL. The driver can then call the routine by using the name of the dummy argument.

For more information about using the port driver interface routines from FORTRAN, see Appendix D.

To use the port driver interface routines, a driver must include the modules $SCSIPORT, $MUTEX, and $KERNELMSG from the VAXELN runtime libraries.

Before calling the port driver interface routines, a class driver must do the following:

- Define device locks
- Set up the appropriate entry point
- Check for devices to service
- Set the current connection flag

Once the driver completes these tasks, it can use the port driver interface routines to:

- Allocate and free device I/O request packets
- Map and unmap buffers for I/O requests
- Issue commands
- Stop the port driver

Sections 14.5.3.2.1 to 14.5.3.2.4 explain how to define the device locks, set up the driver's entry point, check for devices to service, and set the current connection flag. Sections 14.5.3.2.5 to 14.5.3.2.8 explain how to use the interface routines to program driver communication. For descriptions of the interface routines, see Appendix D.

For examples of user-written class drivers in Pascal, C, and FORTRAN, see the modules SAMPLE_SCSIDRIVER.PAS, SAMPLE_SCSIDRIVER.C, and SAMPLE_SCSIDRIVER.FOR in the VAXELN ELN$ directory. You might also consider using the supplied disk and generic class drivers as models while preparing an application-specific class driver.

14.5.3.2.1 Defining Device Locks

The SCSI driver's sniffer module declares eight device locks for controlling the class drivers' ability to gain access to the SCSI devices. The locks are declared as follows:

```
MUTEX lock_device[max_units];
```

Thus, *lock_device[0]* is the lock for SCSI device 0, *lock_device[1]* is the lock for SCSI device 1, and so forth.

A class driver must declare the device locks as external data. The following line of code declares the locks in a class driver written in C:

```
extern MUTEX lock_device[max_units];
```

A class driver must use the device locks to gain exclusive access to a device for issuing commands. Before issuing a command to a device, the driver must lock the device. After issuing the command, the driver must make the device available to other drivers by unlocking it. For more information about issuing commands, see Section 14.5.3.2.7.

14.5.3.2.2 Setting Up an Entry Point

The SCSI driver's sniffer module passes a device marker array and the bus configuration data to each of the class drivers responsible for servicing devices on a bus. Thus, the entry point for class drivers must include two arguments: a pointer to the device marker array and a pointer to the bus configuration table. The marker array identifies the devices on the bus that the class driver is to service. The configuration table provides information about the devices on the bus. An example of an entry point for a user-written class driver follows:

```
void scsi_special(marker_array, scsi$config_table_prt)
  struct marker_arry_type *marker_array;
  struct scsi$config_tbl_data *scsi$config_table_prt;
```

In the sniffer module, the device markers are stored as an array of array structures, with each structure defining the device marker for a particular class driver. The markers are declared as follows:

```
struct marker_array_type
  {
    unsigned char match[max_units];
  };

struct marker_array_type marker_array[max_units];
```

The *max_units* defines the maximum number of device markers that the sniffer module can create. Currently, the maximum is eight, allowing a different class driver to service each of the eight SCSI devices on a bus.

The sniffer module defines the configuration table and table entries as follows:

```
struct scsi$config_tbl_data scsi$config_table

struct scsi$config_tbl_type {
  unsigned char         valid_data;
  unsigned char         devcie_type;
  unsigned char         class_attached;
  unsigned char         current_connection;
  unsigned char         removeable_media;
  unsigned char         product_id[16];
} ;

struct scsi$config_tbl_data {
  struct scsi$config_tbl_type config_tbl[SCSI$K_MAX_UNITS];
};
```

See Table 14–13 for descriptions of the types of data stored in the configuration table.

14.5.3.2.3 Checking for Devices to Service

A class driver must check for the devices that it is to service. A class driver is responsible for servicing a device if the valid data flag in the configuration table is set and the corresponding element in the marker array contains the value 1. The following lines of C code show how a class driver might check for devices that it needs to service:

```
if ((scsi$config_table_prt->config_tbl[unit].valid_data == 1) &&
    (marker_array->match[unit] == 1))
```

14.5.3.2.4 Setting the Current Connection Flag

After a class driver checks for devices that it is to service and is ready to service a particular device, the driver should set the current connection flag for that device to 1. When this flag is set to 1, the device cannot be used by another class driver.

For example, if *unit* represents SCSI device 2, the following line of C code sets the current connection flag for SCSI device 2:

```
scsi$config_table_ptr->config_tbl[unit].current_connection = 1;
```

NOTE

The current connection flag is the only data in the configuration table that a class driver should modify.

When a class driver no longer needs to service a device, the driver should clear the current connection flag. This allows another class driver to connect to the device.

14.5.3.2.5 Allocating I/O Request Packets for Devices

A class driver communicates with a SCSI device using one of 16 available I/O request packets. The request packet transfers command data to the port driver and returns command status information to the class driver. To allocate a request packet, the driver must call the PORT$ALLOCATE_DEVICE routine. This routine allocates a request packet for the calling driver and returns the packet's ID.

When allocating a request packet, a driver must specify the pointer to the port driver's data structures (*routine_addresses.ctx_a_context*), a SCSI device ID, and a command buffer byte count. The routine call must also specify variables that receive pointers to the packet's SCSI command buffer and SCSI status buffer.

The SCSI device ID identifies the device on the SCSI bus that is to handle the I/O request.

The command buffer byte count specifies the number of bytes to be allocated for the packet's SCSI command buffer. The command buffer can store up to 256 bytes of command data. PORT$ALLOCATE_DEVICE returns the address of the command buffer to the specified buffer argument. The driver must use the returned address to place a SCSI command in the request packet.

PORT$ALLOCATE_DEVICE returns the address of the packet's SCSI status buffer to the specified status buffer argument. The 1-byte status buffer receives a status code from the target device after the class driver issues a SCSI command. Using the returned status buffer address, the class driver can check the status code and respond appropriately.

When a class driver no longer needs an I/O request packet, the driver should deallocate the packet by calling the PORT$FREE_DEVICE routine. This routine returns a packet to the list of free request packets. If another process is waiting for a request packet, PORT$FREE_DEVICE will signal that process.

The call to PORT$FREE_DEVICE must specify the pointer to the port driver's data structure and the request packet ID returned by PORT$ALLOCATE_DEVICE.

The following section of C code shows how a class driver might allocate and free an I/O request packet:

```
globalref struct contxt routine_addresses;

int             scsi_dev, cmd_buf_length, packet_id, status;
unsigned char   *cmd_buf_ptr;
unsigned char   *stat_buf_ptr;
    .
    .
    .
{
    .
    .
    .
  packet_id = (*routine_addresses.ctx_a_alloc)
                (routine_addresses.ctx_a_context,
                 scsi_dev,
                 cmd_buf_length,
                 &cmd_buf_ptr,
                 &stat_buf_ptr);
    .
    .
    .
  status = (*routine_addresses.ctx_a_free)
             (routine_addresses.ctx_a_context,
              packet_id);
    .
    .
    .
}
```

14.5.3.2.6 Mapping Data Buffers for I/O Requests

To issue a SCSI command that reads or writes data, a class driver must map a data buffer for the I/O request packet. A driver maps a data buffer by calling the PORT$MAP_BUFFER routine. This routine searches the 128-Kbyte SCSI DMA RAM bit map for a specified amount of contiguous data bytes, updates the I/O request packet with the appropriate mapping information, and marks the bit map pages as unavailable.

A call to PORT$MAP_BUFFER must specify the pointer to the port driver's data structures (*routine_addresses.ctx_a_context*), a packet request ID, a pointer to the buffer to be mapped, the length of the data buffer, a pad size, and the direction of the transfer.

The packet request ID must be a request ID returned by a call to PORT$ALLOCATE_DEVICE.

The buffer pointer identifies the address at which the data buffer is to be mapped. The buffer can store up to 65,536 bytes of read or write data.

The pad size argument is for SCSI device commands that require a transfer size that is larger than the size specified by the data buffer size argument. If the amount of data requested in a SCSI command exceeds the space allocated for the data buffer, the pad size accounts for the difference.

For example, the SCSI READ command transfers data in logical blocks — 512-byte units. Suppose a driver uses the READ command to read the first two bytes of a disk block. The call to PORT$MAP_BUFFER will specify 2 for the data buffer size to accommodate the two bytes to be read. Since the READ command reads data a block at a time, the call must also specify a pad size of 510 to account for the extra 510 bytes.

The direction argument specifies whether the data transfer is a read or write operation. A value of SCSI$K_WRITE indicates a write operation; a value of SCSI$K_READ indicates a read operation.

When a class driver no longer needs an I/O request packet data buffer, the driver should unmap the buffer by calling the PORT$UNMAP_BUFFER routine. This routine returns the memory used for a data buffer back to the 128-Kbyte DMA RAM bit map and marks the returned pages as available. If another process is waiting for DMA RAM memory, PORT$UNMAP_BUFFER signals that process.

The call to PORT$UNMAP_BUFFER must specify the pointer to the port driver's data structures, a request packet ID, the address of the data buffer to be unmapped, the size of the buffer being unmapped, and the buffer's pad size.

The following section of C code shows how a class driver might map and unmap an I/O request packet data buffer:

```
extern struct contxt routine_addresses;

int            packet_id, data_buf_size, data_buf_pad_size;
unsigned char  *data_buf_ptr;
   .
   .
   .
{
  status = (*routine_addresses.ctx_a_map)
              (routine_addresses.ctx_a_context,
               packet_id,
               data_buf_ptr,
               data_buf_size,
               data_buf_pad_size);
   .
   .
   .
  status = (*routine_addresses.ctx_a_unmap)
             (routine_addresses.ctx_a_context,
              packet_id,
              data_buf_ptr,
              data_buf_size,
              data_buf_pad_size);
   .
   .
   .
}
```

14.5.3.2.7 Issuing SCSI Commands

Once a class driver has set up an I/O request packet, the driver can use it to issue SCSI commands, such as INQUIRY, READ, and WRITE. To issue a command, the application must use the PORT$ISSUE_COMMAND routine. This routine arbitrates and selects a device on the SCSI bus, issues the SCSI command that is in the specified request packet, and performs the operations necessary to complete the operation.

A call to the PORT$ISSUE_COMMAND routine must supply a pointer to the port driver's data structures, a request packet ID, and values that specify the following:

- Whether the target device can disconnect during command execution
- Whether the port driver should attempt to repeat a command that fails
- A phase timeout value
- A disconnect timeout value

You specify constant values for the disconnect and port retry arguments. The value for the disconnect argument can be SCSI$K_DISCONNECT or SCSI$K_NODISCONNECT. SCSI$K_DISCONNECT indicates that a target device can disconnect; SCSI$K_NODISCONNECT indicates that the target cannot disconnect. Target devices that remain connected to a bus for long periods of time can adversely affect system performance.

The value for the port retry argument can be SCSI$K_RETRY or SCSI$K_NORETRY. If the value is SCSI$K_RETRY, the port driver can retry a command that fails due to a timeout, bus parity, or invalid phase transition error up to three times. If the value is SCSI$K_NORETRY, the port driver cannot retry commands.

The phase and disconnect timeout values a driver specifies can range from 0 to 420 seconds. The phase timeout value specifies the amount of time a target device has to change to another SCSI bus phase or to complete a data transfer. The disconnect timeout value specifies the amount of time a target device has to reselect an initiator to proceed with a disconnected data transfer. If you specify 0 or an invalid value, the driver uses a timeout value of 20 seconds.

A driver can use PORT$ISSUE_COMMAND to issue commands that are in the Common Command Set (CCS). For information about these commands, see the *American National Standard for Information Systems—Small Computer System Interface-2 (SCSI-2)* specification.

The following section of C code shows how a class driver might issue a SCSI command:

```
extern struct contxt routine_addresses;

int             packet_id, disconnect, disable_retry, phase_timeout
                disconnect_timeout;
    .
    .
    .
{
    .
    .
    .
  status = (*routine_addresses.ctx_a_issue)
               (routine_addresses.ctx_a_context,
                disconnect,
                disable_retry,
                phase_timeout,
                disconnect_timeout);
    .
    .
    .
}
```

14.5.3.2.8 Initializing a SCSI Device Controller

A class driver might want to initialize a SCSI bus controller when a SCSI bus is hung. To initialize a controller, a driver must call the PORT$INITIALIZE_CONTROLLER routine. This routine asserts the SCSI RST signal on the SCSI bus. This signal causes all devices on the SCSI bus to release all asserted signals and places the bus in a BUS FREE state.

A call to PORT$INITIALIZE_CONTROLLER must specify the pointer to the port driver's data structures and the SCSI device ID for a working SCSI target device. For example:

```
status = (*routine_addresses.ctx_a_init)
            (routine_addresses.ctx_a_context,
             scsi_dev);
```

NOTE

The sniffer module calls PORT$INITIALIZE_CONTROLLER once after starting the port driver. A class driver should not call this routine unless the bus is hung.

14.5.3.3 Compiling and Linking the SCSI Driver Modules

After you modify the SCSI driver start-up module and program your class driver, you must compile the modules and then link them into a new VAXELN SCSI driver image.

Compile the start-up module (SCDRIVER.C) and a user-written C class driver as follows:

```
$  CC SCDRIVER + ELN$:VAXELNC/LIBRARY
$  CC SCSIUSER + ELN$:VAXELNC/LIBRARY
```

After compiling the modules, you must link them with the VAXELN SCSI driver components to produce a new VAXELN SCSI driver image. For example:

```
$  LINK SCDRIVER + SCSISNIF + SCSIDISK + SCSIGNRC + -
_$  + SCSIUSER + SCSI5380 + ELN$:CRTLSHARE/LIB + -
_$  RTLSHARE/LIB + RTL/LIB
```

This LINK command links a user class driver with the start-up module, the sniffer module, the supplied disk and generic class drivers, and the port driver. If you modified the start-up module such that it does not include the supplied class drivers, omit those driver modules when linking the driver image as follows:

```
$  LINK SCDRIVER + SCSISNIF + SCSIUSER + SCSI5380 + -
_$  ELN$:CRTLSHARE/LIB + RTLSHARE/LIB + RTL/LIB
```

After you compile and link the driver module, you can build the image into your VAXELN system. For information about building the SCSI driver into VAXELN systems, see the *VAXELN Development Utilities Guide*.

14.6 Realtime Device Drivers

The VAXELN Toolkit includes device drivers for the realtime devices listed in Table 14–14.

Table 14–14: Realtime Devices

Devices	Description
ADQ32	Analog-to-digital converter. The ADQ32 transfers data in DMA mode.

Table 14–14 (Cont.): Realtime Devices

Devices	Description
ADV11–C AXV11–C	Analog-to-digital converter. The AXV11–C is an ADV11–C with two additional digital-to-analog output channels.
ADV11–D	Analog-to-digital converter. The ADV11–D transfers data in programmed and DMA modes.
DLVJ1	Asynchronous serial-line controller. The DLVJ1 (formerly DLV11–J) is a Q-bus interface that contains four asynchronous serial-line channels. It is intended for realtime applications that collect data and control realtime devices by using asynchronous serial lines.
DRB32–E DRB32–M DRB32–W	Parallel-line interface devices. The DRB32 is a 32-bit, half-duplex DMA parallel port for the VAXBI bus. The DRB32–W option is for users who have equipment currently designed to interface with DR11–W devices.
DRQ3B	Parallel-line interface device. The DRQ3B is a 16-bit parallel port for the Q-bus that can run in full-duplex or half-duplex mode.
DRV11–J	Parallel-line interface device. The DRV11–J is a Q-bus interface that provides communication, in 16-bit word lengths, between a MicroVAX system and up to four user devices by using four I/O ports.
DRV11–W	Parallel-line interface device. The DRV11–W is a 16-bit half-duplex DMA parallel port for the Q-bus that supports 18- and 22-bit addressing.
IEQ11–A IEU11–A	IEC/IEEE–488 instrument bus interfaces. The IEQ11–A and IEU11–A interface a Q-bus system to two independent IEC/IEEE instrument buses.
KWV11–C	Programmable, realtime clock. You can use the KWV11–C to initiate action after a specified time interval (by using an interrupt or an external signal) or to time an event.

The design of these drivers prohibits access to a given device from more than one job. However, you can gain access from different processes within the same job, provided the caller ensures that processes do not access the same device simultaneously.

14.6.1 ADQ32 DMA Analog-to-Digital Converter

The VAXELN Toolkit supplies a programming interface for applications that use ADQ32 modules. The ADQ32 module is a high-speed DMA analog input device for Q-bus systems. Up to 32 single-ended or 16 differential channels of input data are converted to 12-bit digital data. Both single-ended and differential input sampling can be used in a single application.

An application can sample multiple channels in any order and can use the programmable gain amplifier at any gain for any sample. The application can sample channel 0 at unity gain and sample channel 1 at a gain of 8. You specify the gain to be used for each sample, independent of each channel.

The interface lets the device's DMA mode logic use block mode data transfers. If you prefer, you can use the ADQ32 device in extended block mode, which provides even more use of Q-bus systems.

The ADQ32 supports a variety of clock modes. The nature of an application determines the clock mode that you should use. Based on the clock mode used, you can also specify the following:

- Base frequency for the sample clock
- Number of ticks to wait before a sample is taken
- Base frequency for the sweep clock
- Number of ticks to wait before a sweep is taken
- Number of conversions to be performed for each sweep
- Base frequency for the delay clock
- Number of ticks to delay before sampling is started

You can access an ADQ32 module from only one job, which must be running in kernel mode. This job can be an ADQ32 server if desired, which allows other jobs to communicate with the device. More than one process in the same job can access the device.

The ADQ32 interface consists of the following routines:

Routine	Description
ELN$ADQ_INITIALIZE	Prepares an ADQ32 device for input and creates the necessary data structures.
ELN$ADQ_QUEUE_READ	Places a DMA read request for an ADQ32 on a request queue.
ELN$ADQ_START	Tells the ADQ32 to start processing data.
ELN$ADQ_TRANSFER_DONE	Removes an ADQ32 read request from the done queue and returns the status of that request.

An application can call the ADQ32 interface routines only from programs running in kernel mode. To use the routines you must include the appropriate modules from the VAXELN runtime libraries. For Pascal programs, you must include the module $ADQ32_UTILITY. If you are programming in C, you must include the modules $vaxelnc and $adq32_utility. For FORTRAN programs, you must include the definition file ELN$FORTRAN_DEFS.FOR.

The supplied modules can be linked as delivered with your calling programs to perform analog-to-digital conversion. The modules also define constants and types used by the routines and status codes returned by the routines. The driver source can serve as a model for drivers for other realtime devices.

Descriptions of the ADQ32 interface procedures are provided for Pascal, C, and FORTRAN programming in *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*, respectively.

For more information about the ADQ32, see the *ADQ32 A/D Converter Module User's Guide*.

14.6.2 ADV11–C/AXV11–C Analog-to-Digital Converter

The Pascal module $AXV_UTILITY, supplied with your development system, defines the procedures provided to interface with the ADV11–C analog-to-digital converter and the AXV11–C. The AXV11–C provides all of the functionality of the ADV11–C and two digital-to-analog outputs as well.

By using a hardware jumper, you can configure an ADV11–C device to have 8 or 16 input channels. With 8 channels, analog voltage is measured across 2 input channels; with 16 channels, voltage is measured with respect to ground. The device has a built-in multiplexer, which permits the sampling and conversion of one channel at a time to a 12-bit binary integer. You can also write a value to the device to be used as a gain in the conversion. (The *LSI–11 Analog System User's Guide* contains more information on the hardware.)

An analog-to-digital conversion can be initiated by program control (setting a bit in the control/status register) by an external signal, or by overflow from the KWV11–C clock option (see Section 14.6.10).

You can access an AXV11–C from only one job, which must be running in kernel mode. This job can be an *AXV11–C server* if desired, which allows other jobs to communicate with the device. More than one process in the same job is permitted to access the device; however, the caller must ensure that no simultaneous accesses to the same device occur.

The procedures provided in the $AXV_UTILITY module can be linked as delivered with your calling programs to perform analog-to-digital conversion. This module also defines status codes returned by the procedures and types needed by the routines. The driver can serve as a model for drivers for other realtime devices. Because the KWV11–C clock can be used in conjunction with an AXV11–C device, some types used in $AXV_UTILITY are defined in the module $KWV_UTILITY.

The $AXV_UTILITY module provides the following procedures:

Routine	**Description**
ELN$AXV_INITIALIZE	Causes an ADV11–C or AXV11–C device to be readied for input, output, or both, and causes all needed data structures to be created. This procedure must be called at least once for each device; it may be called more than once for the same device to change the value of a parameter — for example, to enable the device to gather a larger number of values.

Routine	Description
ELN$AXV_READ	Causes analog data to be sampled from the specified channels, converted to binary form by the device, and stored in a data array. One read is performed for each specified channel. The process is repeated until all data has been collected. This procedure may be called for either an ADV11–C or AXV11–C device.
ELN$AXV_WRITE	Causes a binary number to be converted to an analog voltage on one of the digital-to-analog output channels. This procedure may be called only for an AXV11–C device.

The procedures just described return optional status values. To ensure good realtime response, the procedures provide limited error checking; they report only errors detected by the device. No input parameters are verified, and kernel service calls made in the course of execution raise exceptions upon failure.

Call formats and detailed argument descriptions for the AXV11–C support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual.*

14.6.3 ADV11–D DMA Analog-to-Digital Converter

The Pascal module $ADV_DMA_UTILITY, supplied with your development system, defines the procedures provided to interface with the ADV11–D DMA analog-to-digital converter. The ADV11–D is an analog-to-digital converter that can transfer data in programmed mode or DMA mode. Up to 16 channels of input data are converted to 12-bit digital data. In DMA mode, one command can transfer up to 32,768 words. (For more information about the ADV11–D device, see the *Q-bus DMA Analog System User's Guide.*)

An analog-to-digital conversion can be initiated by program control, by an external signal, or by overflow from the KWV11–C clock option (see Section 14.6.10). To use the clock to trigger input, you should jump the clock-overflow tab to either pin 1 or pin 3 of the J2 connector on the ADV11–D device.

You can access an ADV11–D from only one job, which must be running in kernel mode. This job can be an *ADV11–D server* if desired, which allows other jobs to communicate with the device. More than one process in the same job is permitted to access the device; however, the caller must ensure that no simultaneous accesses to the same device occur.

The procedures provided in the $ADV_DMA_UTILITY module can be linked as delivered with your calling programs to perform analog-to-digital conversion. This module also defines status codes returned by the procedures and types needed by the routines. The driver source can serve as a model for drivers for other realtime devices.

The $ADV_DMA_UTILITY module provides the following procedures:

Routine	**Description**
ELN$ADV_INITIALIZE	Prepares an ADV11–D device for input and creates the necessary data structures.
ELN$ADV_QUEUE_READ	Places a programmed or DMA read request on an ADV11–D request queue.
ELN$ADV_TRANSFER_DONE	Removes the entry of a completed request from the ADV11–D done queue and returns the status of that request.

Call formats and detailed argument descriptions for the ADV11–D support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

14.6.4 DLVJ1 Asynchronous Serial-line Controller

The Pascal module $DLV_UTILITY, supplied with your development system, defines the procedures provided to interface with the DLVJ1 (formerly DLV11–J) asynchronous serial-line controller. The DLVJ1 is a Q-bus interface that contains four asynchronous serial-line channels. The channels can be configured independently for EIA RS–422, RS–423, or RS–232C signal compatibility. Provisions are also made for configuring the channels for 20 milliampere (mA) current loop operation.

Four independent serial-line interfaces exist with consecutive bus device address and vector assignments that can be user-configured by using wire-wrap jumpers on the module. Each serial line can also be configured independently for the following:

- Baud rates 150, 300, 600, 1200, 2400, 4800, 9600, 19200, or 38400 bits per second
- Number of data bits (7 or 8)
- Number of stop bits (1 or 2)
- Parity (none, even, or odd)

All of these configuration parameters are also set by using wire-wrap jumpers on the controller module. (The *DLV11–J User's Guide* contains more information on the hardware.)

The $DLV_UTILITY procedures are intended to provide the most efficient method of controlling the DLVJ1. The procedures are intended for realtime applications that collect data and control realtime devices using asynchronous serial lines. This is in contrast to the support provided for CXY08, CXA16/CXB16, DHQ11, DHT32, DHV11, DMB32, DZV11, DZQ11, which is intended to provide a more functional interface for reading and writing using standard Pascal, C, and FORTRAN I/O routines to terminals connected over the serial lines.

The procedures provided in the $DLV_UTILITY module can be linked with your calling program, which must be running in kernel mode. This module also defines status codes returned by the procedures and types needed by the routines. The driver source, contained in DLVUTIL.PAS and DLVBODY.PAS, can also serve as a model for other drivers for realtime devices. The $DLV_UTILITY module also exports definitions of the DLVJ1's device registers if it is desirable to directly read and write the registers. (See DLVUTIL.PAS for the Pascal definitions or extract the $DLV_UTILITY module from the VAXELNC.TLB library for the C definitions.)

The $DLV_UTILITY module provides the following procedures:

Routine	Description
ELN$DLV_INITIALIZE	Prepares a DLV device line for input and output and creates all needed data structures. This procedure must be called once for each DLV serial line used. Since each line is initialized and handled separately from other lines, each line should have its own device description specified in the target system's System Builder menus.
ELN$DLV_READ_BLOCK	Causes characters to be read from the serial line until the specified number of characters is read. This procedure should be called to read from the serial line if the *string_mode* argument was FALSE in the call to ELN$DLV_INITIALIZE.
ELN$DLV_READ_STRING	Causes characters to be read from the serial line until a carriage return character is encountered. This procedure should be called to read from the serial line if the *string_mode* argument was TRUE in the call to ELN$DLV_INITIALIZE.
ELN$DLV_WRITE_STRING	Causes the specified character string to be written to the serial line. The characters are not interpreted by this procedure; therefore, any variable-length string can be written.

Call formats and detailed argument descriptions for the DLVJ1 support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

14.6.5 DRB32 DMA Parallel-Line Interface

The Pascal module $DRB_UTILITY, supplied with your development system, defines the procedures provided to interface with the DRB32–E, DRB32–M, and DRB32–W DMA parallel-line controllers. The DRB32 is a VAXBI bus interface that provides communication through a half-duplex DMA parallel port at data widths of 8, 16, or 32 bits. In addition, the DRB32:

- Uses page tables so that buffers do not need to be physically contiguous.

- Uses two sets of control registers for hardware-supported double buffering. When the device finishes a transfer by using one set of registers and page tables, the device automatically starts a transfer from the second set of registers.
- Has 8-bit input and output control/status registers that correspond to control lines on the port and have a fixed meaning. You can set the DRB32 to interrupt when an input control line changes.
- Can check the parity on its data lines.

The DRB32–W option is for users of equipment currently designed to interface with the DR11–W device.

In closely coupled symmetric multiprocessing configurations, KA800 processors can use DRB32 devices to communicate with user devices. KA800 processors can directly control the DRB32 parallel port for high interrupt response time.

The $DRB_UTILITY procedures are intended to provide the most efficient method of controlling the DRB32. The procedures are intended for realtime applications that collect data and control realtime devices using parallel lines. This is in contrast to the support provided for devices that are not used in a realtime environment and are intended to provide a more functional interface for reading and writing using standard Pascal and C I/O routines.

The procedures provided in the $DRB_UTILITY module can be linked with your calling program, which must be running in kernel mode. This module also defines status codes returned by the procedures and types needed by the routines. The driver source, contained in DRB32UTIL.PAS and DRB32BODY.PAS, can also serve as a model for other drivers for realtime devices. The $DRB_UTILITY module also exports definitions of the DRB32's device registers if it is desirable to directly read and write the registers. (See DRB32UTIL.PAS for the Pascal definitions or extract the $DRB_UTILITY module from the VAXELNC.TLB library for the C definitions.)

These procedures assume that the user device connected to the DRB32 asserts the SYNCH OUT, SYNCH IN, CONTROL SYNCH OUT, and CONTROL SYNCH IN lines when the device is to inform the DRB32 that data is available for the application program to read or that the application program wrote data to the device. See the *DRB32 Hardware Manual* for more information.

The $DRB_UTILITY module provides the following procedures:

Routine	Description
ELN$DRB_FINISHED_TRANSFER	Dequeues a completed request from the device driver and returns its status and a pointer to its data buffer. If no completed request is available, the procedure can wait or return, at your option.
ELN$DRB_INITIALIZE	Initializes a DRB32 device for input and output, creates all needed data structures, starts the queues that handle requests, and aborts current or queued commands. This procedure must be called once for each DRB32 controller used. The procedure call specifies the data width. Two arguments are provided for use with a DRB32–W device. One argument identifies whether the device being initialized is a DRB32–W device. The other argument specifies whether a DRB32–W device is to operate in link mode, which typically means two DRB32–W devices are connected for data transfer. The default mode specifies that the DRB32–W is connected to a DR11–W device.
ELN$DRB_QUEUE_READ	Queues a read request to the driver, starts the request if the queue is empty, and returns. The request causes data to be read into a buffer you specify. If you are using a DRB32–W device, you may have to use the ELN$DRB_WRITE_CTRL procedure to set or clear appropriate function bits in the IOCTL register (FUNCT1, FUNCT2, FUNCT3) before calling ELN$DRB_QUEUE_READ, or to properly establish the direction of transfer.

Routine	Description
ELN$DRB_QUEUE_WRITE	Queues a write request to the driver, starts the request if the queue is empty, and returns. The request causes data to be written from a buffer you specify. If you are using a DRB32–W device, you may have to use the ELN$DRB_WRITE_CTRL procedure to set or clear appropriate function bits in the IOCTL register (FUNCT1, FUNCT2, FUNCT3) before calling ELN$DRB_QUEUE_WRITE to properly establish the direction of transfer.
ELN$DRB_WRITE_CTRL	Writes an 8-bit pattern to a DRB32's 8-bit control register. An argument is provided for use with DRB32–W devices. This argument specifies which bits of the IOCTL register are to be returned: the upper bits 8 to 15 (output) or the lower bits 0 to 7 (input).
ELN$DRB_READ_CTRL	Returns an 8-bit pattern from a DRB32's 8-bit control register.

NOTE

These routines assume that the user device is connected to the DRB32 device and asserts the SYNCH OUT, SYNCH IN, CONTROL SYNCH OUT, and CONTROL SYNCH IN lines to inform the device that data is available for the application program to read or that the application program wrote data to the device. See the *DRB32 Hardware Manual* for more information.

To link two DRB32–W devices, the receiving end must post a read request (ELN$DRB_QUEUE_READ) before the sending end posts a write request (ELN$DRB_QUEUE_WRITE). Also, the receiving end must not post a subsequent read until the sending end has completed sending its data. For more information on links, refer to the *DR11–W Direct Memory Access Interface User's Guide*.

Call formats and detailed argument descriptions for the DRB32 support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual.*

14.6.6 DRQ3B DMA Parallel-Line Interface

The Pascal module $DRQ3B_UTILITY, supplied with your development system, defines the procedures provided to interface with the DRQ3B DMA parallel-line controller. The DRQ3B module performs DMA data transfers to or from system memory through 16-bit parallel data ports. The module provides two distinct ports for connection to external devices: an input port that supports device-to-memory or memory-to-memory transfers and an output port that supports memory-to-device or memory-to-memory transfers. Each channel is unique, allowing a full-duplex mode.

The DRQ3B device performs DMA operations in nonblock mode (single-cycle or burst-mode) or block mode. In nonblock mode, each data word to be transferred is accompanied by an address location when placed on the Q-bus. In block mode, only the first address asserted in each block (up to 16 words) of data is asserted on the bus to indicate the starting address.

For nonblock mode, you can specify two types of DMA operations: single-cycle and burst-mode transfers. Single-cycle operations transfer one address and one data word per bus cycle, then release the bus. Burst-mode operations transfer one address word for each data word. However, up to four address/data word combinations are transferred before the bus is released.

For block mode transfers, the address location of the first data word is placed on the bus, followed by up to 16 data words, before the DRQ3B device gives up the bus.

For more information about the DRQ3B device, see the *DRQ3B Parallel DMA I/O Module User's Guide.*

The $DRQ3B_UTILITY procedures are intended to provide the most efficient method of controlling the DRQ3B. The procedures are intended for realtime applications that collect data and control realtime devices using parallel lines. This is in contrast to the support provided for devices that are not used in a realtime environment and are intended to provide a more functional interface for reading and writing using standard Pascal and C I/O routines.

You can access a DRQ3B from only one job, which must be running in kernel mode. More than one process in the same job is permitted to access the device; however, the caller must ensure that no simultaneous accesses to the same device occur.

The procedures provided in the $DRQ3B_UTILITY module can be linked as delivered with your calling programs to perform DRQ3B I/O. This module also defines status codes returned by the procedures and types needed by the routines. The driver source can serve as a model for drivers for other realtime devices.

The $DRQ3B_UTILITY module provides the following procedures:

Routine	**Description**
ELN$DRQ3B_INITIALIZE	Initializes a DRQ3B device, creates necessary data structures, starts the internal request queues, and aborts current or queued commands.
ELN$DRQ3B_QUEUE_READ	Queues a read request to the DRQ3B driver and returns.
ELN$DRQ3B_QUEUE_WRITE	Queues a write request to the DRQ3B driver and returns.
ELN$DRQ3B_READ_FUNCTION	Returns the DRQ3B general-purpose function bits.
ELN$DRQ3B_TRANSFER_DONE_ READ	Dequeues a completed DRQ3B read request and returns its status.
ELN$DRQ3B_TRANSFER_DONE_ WRITE	Dequeues a completed DRQ3B write request and returns its status.
ELN$DRQ3B_WRITE_FUNCTION	Writes to the DRQ3B general-purpose latched function bits.

Call formats and detailed argument descriptions for the DRQ3B support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

14.6.7 DRV11–J Parallel-Line Interface

The Pascal module $DRV_UTILITY, supplied with your development system, defines the procedures provided to interface with the DRV11–J parallel-line interface device. The DRV11–J is a Q-bus interface that provides communication between a MicroVAX system and up to four user devices in 16-bit word lengths through four I/O ports.

Four control lines are associated with each of the four ports to ensure orderly information transfers. Word transfers are executed by programmed I/O bus operations using either polling or interrupt-driven routines. Write data is output by the DRV11–J to the I/O bus through three-state data latches, and read data is input through unlatched bus drivers.

The $DRV_UTILITY procedures are intended to provide the most efficient method of controlling the DRV11–J. The procedures are intended for realtime applications that collect data and control realtime devices using parallel lines. This is in contrast to the support provided for devices that are not used in a realtime environment and are intended to provide a more functional interface for reading and writing using standard Pascal and C I/O routines.

The procedures provided in the $DRV_UTILITY module can be linked with your calling program, which must be running in kernel mode. This module also defines status codes returned by the procedures and types needed by the routines. The driver source, contained in the DRVUTIL.PAS and DRVBODY.PAS modules can also serve as a model for other drivers for realtime devices. The $DRV_UTILITY module also exports definitions of the DRV11–J's device registers if it is desirable to directly read and write the registers. (See the DRVUTIL.PAS module for the Pascal definitions or extract the $DRV_UTILITY module from the VAXELNC.TLB library for the C definitions.)

The procedures perform all I/O operations, using a dynamically allocated, 2-dimensional buffer array. The first array index specifies the parallel port number (0 to 3), and the second array index specifies a data word. The procedures internally utilize a separate DEVICE object for each parallel port. Therefore, a user program can have interrupt-driven I/O in progress on each port simultaneously. For example, an application program can have a process writing data to ports 0 and 1 and another process reading data from ports 2 and 3. Due to the way the DRV11–J functions, though, only one port can have concurrent I/O if polling is used instead of interrupts.

The procedures assume that the user device connected to the DRV11–J asserts the USER REPLY lines when the user device is to inform the DRV11–J either that data is available for reading by the application program or that data has been accepted (written by the application program).

The $DRV_UTILITY module provides the following procedures:

Routine	Description
ELN$DRV_INITIALIZE	Prepares a DRV device controller for input and output and creates all needed data structures. This procedure must be called once for each DRV controller used.
ELN$DRV_READ	Causes data words to be read from the specified parallel port. The resulting data is stored in the buffer pointed to by the buffer parameter returned by ELN$DRV_INITIALIZE.
ELN$DRV_WRITE	Causes data words to be written to the specified parallel port. Before you call this procedure, the data words should be stored in the buffer pointed to by the buffer parameter returned by ELN$DRV_INITIALIZE.

NOTE

These procedures assume that the user device connected to the DRV11–J asserts the USER REPLY lines to inform the DRV11–J device either that data is available for reading by the application program or that data has been accepted (written by the application program).

Call formats and detailed argument descriptions for the DRV11–J support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual, VAXELN C Runtime Library Reference Manual, VAXELN FORTRAN Runtime Library Reference Manual.*

14.6.8 DRV11–W DMA Parallel-Line Interface

The Pascal module $DRV_DMA_UTILITY, supplied with your development system, defines the procedures provided to interface with the DRV11–W DMA parallel-line controller. The DRV11–W is a Q-bus interface that provides communication through a 16-bit, half-duplex DMA parallel port. The device supports 18- and 22-bit addressing but does not support page tables (data must be contiguous) or double buffering.

The $DRV_DMA_UTILITY procedures are intended to provide the most efficient method of controlling the DRV11–W. The procedures are intended for realtime applications that collect data and control realtime devices using parallel lines. This type of support is in contrast to the support provided for devices that are not used in a realtime environment and are intended to provide a more functional interface for reading and writing using standard Pascal and C I/O routines.

The procedures provided in the $DRV_DMA_UTILITY module can be linked with your calling program, which must be running in kernel mode. This module also defines status codes returned by the procedures and types needed by the routines. The driver source, contained in the DRV11WAUTIL.PAS and DRV11WABODY.PAS modules, can also serve as a model for other drivers for realtime devices. The $DRV_DMA_UTILITY module also exports definitions of the DRV11–W's device registers if it is desirable to directly read and write the registers. (See the DRV11WAUTIL.PAS module for the Pascal definitions or extract the $DRV11W_UTILITY module from the VAXELNC.TLB library for the C definitions.)

These procedures assume that the user device connected to the DRV11–W asserts the USER REPLY lines when the user device is to inform the DRV11–W that data is available for the program to read or that data written by the program was accepted.

The $DRV_DMA_UTILITY module provides the following procedures:

Routine	Description
ELN$DRV_DMA_INITIALIZE	Initializes a DRV11–W device controller for input and output, creates all needed data structures, and starts the queues that handle requests. This procedure must be called once for each DRV11–W controller used.
ELN$DRV_DMA_QUEUE_READ	Queues a read request to the driver, starts the request if the queue is empty, and returns. The request causes data to be read into the buffer you specify; you are responsible for creating your data area using messages to ensure physically contiguous data.
ELN$DRV_DMA_QUEUE_WRITE	Queues a write request to the driver, starts the request if the queue is empty, and returns. The request causes data to be written from the buffer you specify; again, you must create your data area using messages to ensure physically contiguous data.
ELN$DRV_DMA_TRANSFER_ DONE	Dequeues a completed request from the device driver and returns its status and a pointer to its data buffer. If no completed request is available, the procedure can wait or return, at your option.
ELN$DRV_DMA_WRITE_ FUNCTION	Modifies the function bits of the DRV11–W control status register (CSR). This procedure writes a 3-bit pattern to the 3-bit CSR function field.
ELN$DRV_DMA_READ_STATUS	Returns the status bits (3-bit field) from the DRV11–W CSR.

NOTE

These procedures assume that the user device connected to the DRV11–W asserts the USER REPLY lines to inform the DRV11–W device either that data is available for reading by the application program or that data has been accepted (written by the application program).

Call formats and detailed argument descriptions for the DRV11–W support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

14.6.9 IEQ11–A and IEU11–A Dual IEC/IEEE Instrument Bus Interfaces

The Pascal module $GPIB_SUB supplied with your development system defines the procedures provided to interface with the IEQ11–A and IEU11–A devices. The IEQ11–A is a DMA controller that interfaces a Q-bus system to two independent IEC/IEEE instrument buses. Similarly, the IEU11–A is a DMA controller that interfaces a UNIBUS system (not a VAXBI system with the DWBUA BI-to-UNIBUS adapter) to two independent IEC/IEEE instrument buses. (Alternatively, the IEQ11–A and IEQ11U–A devices can provide two ports to the same instrument bus.) Each instrument bus can have up to 15 devices, including the IEQ11–A or IEU11–A, in a sequential configuration. Each bus allows instruments on the same bus to communicate with each other. Each device on the bus has a unique address to which it responds. Information is transmitted in byte, serial-bit, or parallel format and may consist of either commands or data.

The IEC/IEEE instrument bus is a General Purpose Interface Bus (GPIB). ANSI/IEEE 488–1978, *IEEE Standard Digital Interface for Programmable Instrumentation*, specifies the characteristics of the bus and the functions it performs.

The bus consists of 24 lines. Of these, 8 lines are ground wires, and 16 carry information. Of the 16 information lines, 3 are used for handshaking control, and 5 for bus management; 8 carry data between devices on the bus.

You will generally not be concerned with the control lines (NRFD, DAV, and NDAC), since the hardware takes care of the handshaking.

The five bus management lines are:

Line	Mnemonic
Attention	ATN
Service request	SRQ
Interface clear	IFC

Line	Mnemonic
End or identify	EOI
Remote enable	REN

The eight data lines are used to transfer a byte of data at a time across the bus.

At any time, only one device on the bus acts as *bus controller*. The bus controller issues the commands needed to perform data transfers. Each device on the bus has the potential to perform the following functions:

- Act as bus controller
- Act as *talker* in a bus transfer
- Act as *listener* in a bus transfer
- Issue a service request to the bus controller
- Respond to polls by the bus controller

The IEQ11–A and IEU11–A provide two independent ports to the IEC/IEEE bus. These ports can interface to two different buses or provide two ports into the same bus. The ports are treated as separate controllers.

The functioning of these ports is controlled by eight hardware registers for each port. The registers are:

Register	Mnemonic	Address
IEEE Status Read: Address Status/Bus Status Write: Int Mask 0/Int Mask 1	ISR	76XXX0
IEEE Interrupt Read: Int Status 0/Int Status 1 Write: -/Address	IIR	76XXX2
IEEE Command Read: Cmd Pass Thru/- Write: Serial Poll/Auxiliary Cmd	ICR	76XXX4
IEEE Data Read: -/Data In Write: Parallel Poll/Data Out	IDR	76XXX6

Register	Mnemonic	Address
Control/Status	CSR	76XX10
Bus Address	BAR	76XX12
Byte Count	BCR	76XX14
Match Character	MCR	76XX16

The corresponding registers for the two ports have identical addresses. The setting of a multiplexer bit in the CSR, based on a user-specified controller ID or unit number, determines which port's register is referenced. Aside from register sharing, however, the two instrument bus ports are functionally independent.

As indicated by the Read and Write designations in the preceding table, the four IEEE register addresses reference different registers, depending on whether a reference is a read or a write.

For more information about the IEQ11–A or IEU11–A device, see the *IEU11–A / IEQ11–A User's Guide* and the *IEX11–A IEC / IEEE Bus Interface*.

To use the procedures provided in the $GPIB_SUB module, you must link your programs with RTLOBJECT.OLB, or in the case of C programs, with CTRLOBJECT.OLB. In addition to providing the procedures, the $GPIB_SUB module defines status codes returned by the procedures and the data types that the procedures use. The driver source can serve as a model for drivers for other realtime devices.

The $GPIB_SUB module provides the following procedures:

Routine	Description
ELN$GP_AUXILIARY_COMMAND	Issues a specified auxiliary command to an IEQ11–A or IEU11–A unit's auxiliary command register.
ELN$GP_CLEAR_EVENT	Clears all events set previously by GP_SET_EVENT for an IEQ11–A or IEU11–A unit.
ELN$GP_CONFIGURE	Configures an IEC/IEEE instrument bus.

Routine	Description
ELN$GP_DEFINE_PATH	Defines the data paths between devices that can be talkers and listeners on an IEC/IEEE instrument bus.
ELN$GP_GET_CONTROL	Takes control of an IEC/IEEE bus if the specified IEQ11–A or IEU11–A unit is the controller-in-charge.
ELN$GP_GOTO_STANDBY	Issues the auxiliary command GTS (go to standby mode) to an IEQ11–A or IEU11–A unit if the unit is the controller-in-charge.
ELN$GP_INITIALIZE	Establishes communication with the IEQ11–A or IEU11–A instrument-bus interface device.
ELN$GP_LOAD_PARALLEL_POLL	Loads an IEQ11–A unit's parallel-poll hardware register with a specified value.
ELN$GP_PARALLEL_POLL	Requests a parallel poll of devices on an IEC/IEEE bus and returns a parallel-poll value.
ELN$GP_PARALLEL_POLL_CONFIG	Configures a parallel poll for the specified devices on an IEC/IEEE bus.
ELN$GP_PASS_CONTROL	Passes control from an IEQ11–A or IEU11–A unit to another device on the IEC/IEEE bus.
ELN$GP_RECEIVE_CONTROL	Lets an IEQ11–A or IEU11–A unit receive control from another device on the IEC/IEEE bus.
ELN$GP_REQUEST_SERVICE	Issues a service request (SRQ) on behalf of a specified IEQ11–A or IEU11–A unit.
ELN$GP_SEND_COMMAND	Sends the specified number of interface commands or data bytes to the IEQ11–A or IEU11–A data output register.
ELN$GP_SENSE_MODE	Returns the IEC/IEEE bus status and the specified IEQ11–A or IEU11–A unit's controller status.

Routine	Description
ELN$GP_SERIAL_POLL	Performs a serial poll of the specified devices on an IEC/IEEE instrument bus while the service request (SRQ) bit is asserted, to determine which devices requested service.
ELN$GP_SET_EVENT	Specifies events to watch for on an IEC/IEEE bus.
ELN$GP_TRANSFER	Transfers data between devices on an IEC/IEEE instrument bus according to the data paths specified in a call to GP_DEFINE_PATH.
ELN$GP_UNIT_INIT	Initializes a specified IEQ11–A or IEU11–A port (unit).

Call formats and detailed argument descriptions for the IEQ11–A and IEU11–A support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

14.6.10 KWV11–C Realtime Clock

The Pascal module $KWV_UTILITY, supplied with your development system, defines the procedures provided to interface with the KWV11–C realtime clock. The KWV11_C is a programmable, realtime clock that can be used to initiate action after a specified time interval (through an interrupt or an external signal) or to time an event. In the first mode, it can be used with an ADV11–C, AXV11–C, or ADV11–D device to initiate the collection of data.

The device's clock counter has a resolution of 16 bits. The clock counter can be driven from any of five internal crystal-controlled frequencies, from a line frequency input, or from Schmitt Trigger #1, which is fired by an external input. Another Schmitt Trigger, #2, can be used to start the counter. (A Schmitt Trigger is a logic device that responds to voltage levels rather than to voltage transitions. The *LSI–11 Analog System User's Guide* contains more information on the hardware.)

The driver interface provided for the KWV11–C is of the same style as that provided for the ADV11–C and ADV11–D, described previously. The VAXELN Toolkit supplies the KWV11–C driver to allow you to use all of the functionality of the ADV11–C and ADV11–D.

The design of this driver precludes accessing a given KWV11–C device from more than one job, and that job must be running in kernel mode. More than one process in the same job is permitted to access the device; however, the caller must ensure that no simultaneous accesses to the same device occur.

The procedures provided in the $KWV_UTILITY module can be linked as delivered with your calling programs to interface with the KWV11–C clock. This module also defines status codes returned by the procedures and types needed by the routines. The $KWV_UTILITY module provides the following procedures:

Routine	Description
ELN$KWV_INITIALIZE	Causes a KWV11–C device to be readied for input and causes all needed data structures to be created. This procedure must be called at least once for each KWV11–C; it may be called more than once for the same device to change the value of a parameter — for example, to enable the device to gather a larger number of values.
ELN$KWV_READ	Causes time values to be read from the device and stored in a data array; these values represent timings of external events. This procedure may also be used to gather the elapsed time that began with a call to ELN$KWV_WRITE.
ELN$KWV_WRITE	Causes the device to be set up such that, when the given number of ticks has occurred, the clock overflow signal is generated. Overflow signals may be repeatedly generated, depending on how the device was initialized. This procedure can also be used to start the clock if the intent is to later stop and read it with ELN$KWV_READ.

The procedures just described return optional status values. To ensure good realtime response, the procedures provide limited error checking; they report only errors detected by the device. No input parameters are verified, and kernel service calls made in the course of execution raise exceptions upon failure.

Call formats and detailed argument descriptions for the KWV11–C support routines are provided in the *VAXELN Pascal Runtime Library Reference Manual*, *VAXELN C Runtime Library Reference Manual*, and *VAXELN FORTRAN Runtime Library Reference Manual*.

Appendix A
Status Values/Exception Names

The VAXELN Kernel procedures and some utility procedures accept an optional status argument that receives the procedure's completion status. If you specify the status argument in a procedure call, you can check the status value after the call to determine whether the operation was successful. If you omit the status argument and a fatal error occurs, an exception condition results.

Exceptions have the same names as the corresponding status values. For example, KER$_NO_SUCH_PROGRAM can be either a status value or exception name, depending on whether you specify the status argument. You can use these names in exception handlers.

For information about checking status arguments and establishing exception handlers, see Chapter 7. Table A–1 lists the status values/exception names that VAXELN programs raise. For more details about a particular status value/exception name, see the corresponding message symbol in the *VAXELN Messages Manual.*

Table A–1: Status Values/Exception Names

Name	Description
C Runtime Library	
C$_EACCES	Permission denied
C$_EADDRINUSE	Address already in use
C$_EADDRNOTAVAIL	Cannot assign requested address
C$_EAFNOSUPPORT	Address family not supported

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
C Runtime Library	
C$_EAGAIN	No more processes
C$_EALREADY	Operation already in progress
C$_EBADF	Bad file number
C$_E2BIG	Argument list too long
C$_EBUSY	Mount device busy
C$_ECHILD	No children
C$_ECONNABORTED	Software caused connection to abort
C$_ECONNREFUSED	Connection refused
C$_ECONNRESET	Connection reset by peer
C$_EDESTADDRREQ	Destination address required
C$_EDOM	Math argument error
C$_EEXIST	File exists
C$_EFAULT	Bad address
C$_EFBIG	File too large
C$_EHOSTDOWN	Host is down
C$_EHOSTUNREACH	No route to host
C$_EINPROGRESS	Operation in progress
C$_EINTR	Interrupted system call
C$_EINVAL	Invalid argument
C$_EIO	I/O error
C$_EISCONN	Socket is already connected
C$_EISDIR	Is a directory
C$_ELOOP	Too many levels of symbolic links
C$_EMSGSIZE	Message too long
C$_EMFILE	Too many open files
C$_EMLINK	Too many links
C$_ENAMETOOLONG	File name too long

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
C Runtime Library	
C$_ENETDOWN	Network is down
C$_ENETRESET	Network dropped connection on reset
C$_ENETUNREACH	Network is unreachable
C$_ENFILE	File table overflow
C$_ENOBUFS	No buffer space available
C$_ENODEV	No such device
C$_ENOENT	No such file or directory
C$_ENOEXEC	Exec format error
C$_ENOMEM	Not enough core
C$_ENOPROTOOPT	Protocol not available
C$_ENOSPC	No space left on device
C$_ENOTBLK	Block device required
C$_ENOTCONN	Socket is not connected
C$_ENOTDIR	Not a directory
C$_ENOTSOCK	Not a socket; socket operation requires a socket
C$_ENOTTY	Not a typewriter
C$_ENXIO	No such device or address
C$_EOPNOTSUPP	Operation not supported on socket
C$_EPERM	Not owner; need appropriate privileges
C$_EPFNOSUPPORT	Protocol family not supported
C$_EPIPE	Broken pipe
C$_EPROTONOSUPPORT	Protocol not supported
C$_EPROTOTYPE	Protocol wrong type for socket
C$_ERANGE	Result too large

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
C Runtime Library	
C$_EROFS	Read-only file system
C$_ESHUTDOWN	Cannot send after socket shutdown
C$_ESOCKTNOSUPPORT	Socket type not supported
C$_ESPIPE	Invalid seek
C$_ESRCH	No such process
C$_ETIMEDOUT	Connection timed out
C$_ETOOMANYREFS	Too many references; cannot splice
C$_ETXTBSY	Text file busy
C$_EVMSERR	VMS error code for non-translatable errors
C$_EWOULDBLOCK	I/O operation would block channel
C$_EXDEV	Cross-device link
DECwindows XUI Toolkit Runtime Library	
DWT$_DWTABORT	X Toolkit fatal error
VAXELN Runtime Library	
ELN$_ABORTED	Connection attempt failed
ELN$_ACC	Files–11 ACP access failed
ELN$_ACS	Error in access control string
ELN$_ACT	File activity precludes operation
ELN$_ADAWI	First argument in call to ADD_INTERLOCKED is out of range
ELN$_ALLRDYRUN	Device is already running
ELN$_ALQ	Invalid allocation quantity
ELN$_AMBENUMSTR	Ambiguous specification for enumerated type
ELN$_ANI	Not ANSI "D" format
ELN$_ARGUMENT	Nonexistent argument in call to ARGUMENT
ELN$_ARRAYBOUND	Corresponding array bounds are not equal

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_ASSERT	Failed assertion
ELN$_AUTH_DUPLICATE_USER	Duplicate user
ELN$_AUTH_INVALID_UIC	Invalid UIC value
ELN$_AUTH_NO_AUTHORIZATION	No authorization for user
ELN$_AUTH_NO_PRIVILEGE	No privilege for request
ELN$_AUTH_NO_SUCH_USER	No such user
ELN$_AUTH_UNKNOWN_REQUEST	Unknown request
ELN$_AXV_DEVICE_ERROR	Device error; clock too fast for requests
ELN$_BADIMGFMT	Bad image format
ELN$_BADSTATE	Bad state exists
ELN$_BADVALUE	Bad parameter value
ELN$_BES	Bad escape sequence
ELN$_BLKCHK_CRC_ERR	Block check or CRC error
ELN$_BOF	Beginning-of-file detected
ELN$_BOOTERROR	Insufficient physical memory, insufficient contiguous physical memory, processor identification mismatch, unexpected interrupt or exception, or unexpected machine check
ELN$_BUGDAP	Internal network error condition detected
ELN$_CASELAB	No case label exists corresponding to the selector value
ELN$_CHARASGN	Assignment of a string not of length 1 to a character
ELN$_CHR	Operand to CHR is out of the range 0 to 255
ELN$_CONFLICTINGVAL	Conflicting argument values specified
ELN$_CRC	Network DAP level CRC check failed

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_CUR	No current record; operation not preceded by $GET or $FIND
ELN$_DATA_OVERRUN	Data overrun
ELN$_DEL	RFA-accessed record was deleted
ELN$_DEV	Error in device name or inappropriate device type for operation
ELN$_DEVACTIVE	Device already active
ELN$_DEVNOTREADY	Device not ready
ELN$_DEVOFFLINE	Device is not on line
ELN$_DIR	Error in directory name
ELN$_DIR_FNM	Directory listing; error in reading volume-set name, directory name, or file name
ELN$_DIR_FUL	Directory full
ELN$_DISK_ALLOCFAIL	Index file allocation failure
ELN$_DISK_BADRANGE	Bad block address not on volume
ELN$_DISK_BLKZERO	Block zero is bad; volume not bootable
ELN$_DISK_CLUSTER	Unsuitable cluster factor
ELN$_DISK_DEVMOUNT	Device is already mounted
ELN$_DISK_DIAGPACK	Disk is a diagnostic pack
ELN$_DISK_FACTBAD	Cannot read factory bad block data
ELN$_DISK_INVCHRVOL	Invalid character in volume label
ELN$_DISK_LARGECNT	Disk too large to be supported
ELN$_DISK_MAXBAD	Bad block table overflow
ELN$_DISK_NOBADDATA	Bad block data not found on volume
ELN$_DISK_NOTFILEDEV	Device is not file structured
ELN$_DME	Dynamic memory exhausted
ELN$_DNF	Directory not found
ELN$_DNR	Device not ready or not mounted

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_DPE	Device positioning error
ELN$_END_OF_TAPE	End-of-tape detected
ELN$_END_OF_VOLUME	End-of-volume detected
ELN$_ENTRYEXISTS	Entry already exists
ELN$_EOF	End-of-file detected
ELN$_EOFNOTDEF	EOF taken when undefined
ELN$_EOLN	EOLN taken when file at end-of-file
ELN$_ERRDURLOA	Error occurred during load operation
ELN$_FAC	Record operation not permitted by specified file access (FAC)
ELN$_FATAL_HWE	Fatal hardware error
ELN$_FEX	File already exists, not superseded
ELN$_FILE_ALROPEN	File already open
ELN$_FILE_ALTHOMBLK	Alternate home block used
ELN$_FILE_ALTIDXFHD	Alternate index file header used
ELN$_FILE_BADIDXFHD	No valid index file header found
ELN$_FILE_BITMAPERR	I/O error on storage bitmap; volume locked
ELN$_FILE_DEVINUSE	Another processor is using device
ELN$_FILE_DEVNOTMNT	No volume mounted on device
ELN$_FILE_FILESTRUCT	Unsupported file structure level or ODS feature
ELN$_FILE_HDR_CHKSUM	File header checksum failure
ELN$_FILE_HDR_FULL	File header full
ELN$_FILE_IDXMAPERR	I/O error on index file bitmap; volume locked
ELN$_FILE_INCVOLLABEL	Incorrect volume label, volume mounted anyway
ELN$_FILE_MAPHDRBAD	Storage map header is bad; volume locked
ELN$_FILE_MLTVOLABEL	A volume with this name has already been mounted

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_FILE_NOHOMEBLOCK	No valid home block found on volume
ELN$_FILE_VOLALRMNT	ELN$MOUNT_VOLUME
ELN$_FILE_VOLIMPDSM	Volume was improperly dismounted; rebuild on VMS system
ELN$_FINDFIRST	Start index out-of-range in call to FIND_FIRST_BIT_CLEAR or FIND_FIRST_BIT_SET
ELN$_FLK	File currently locked by another user
ELN$_FND	Files–11 ACP file or directory lookup failed
ELN$_FNF	File not found
ELN$_FNM	Error in file name
ELN$_FOP	Invalid file options
ELN$_FSZ	Invalid fixed control header size
ELN$_FTM	Network file transfer mode precludes operation (SQO) set
ELN$_FUL	Device full; insufficient space for allocation
ELN$_IDR	Invalid directory rename operation
ELN$_IDXF_FULL	Index file full
ELN$_IFA	Invalid file attributes detected; file header corrupted
ELN$_INTCONVERT	Expression out-of-range for conversion to type BOOLEAN or an enumerated type
ELN$_INVALADDR	Invalid or missing address value
ELN$_INVALBUFSIZ	Invalid buffer size
ELN$_INVALCHARSIZ	Invalid character size
ELN$_INVALDSKSIZ	Bad parameter input for VM disk size
ELN$_INVALFUNC	Invalid function
ELN$_INVALLINE	Invalid line name
ELN$_INVALNAM	Invalid node, port, or service name

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_INVALNODE	Invalid node name or address
ELN$_INVALPARITY	Invalid parity type
ELN$_INVALREC	Invalid record definition
ELN$_INVALSPEED	Invalid terminal speed
ELN$_INVALSUBFUNC	Invalid subfunction
ELN$_INVALTYP	Invalid port type
ELN$_INVDBLSTR	Invalid specification for a number of type DOUBLE
ELN$_INVENUMSTR	Invalid enumerated type syntax
ELN$_INVENUMVAL	Invalid enumerated type value
ELN$_INVREALSTR	Invalid specification for a number of type REAL
ELN$_INVTIMSTR	Invalid time specification
ELN$_INVTIMVAL	Invalid time value
ELN$_IOP	Invalid operation for file organization or device
ELN$_IRC	Invalid record encountered; with sequential files only
ELN$_KEY	Invalid record number key or key value
ELN$_KWV_DATA_OVERRUN	Data overrun; external events occurring too fast
ELN$_LATACTIVE	LAT protocol already active
ELN$_LATNOTACTIVE	LAT protocol is not active
ELN$_LBL	Tape label is not ANSI format
ELN$_LNE	Logical name translation count exceeded
ELN$_LOCKED	Entry is locked
ELN$_MAXLOADS	Maximum number of concurrent loads reached
ELN$_MAXSERVICE	Maximum number of services reached

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_MISLINNAM	Missing line name
ELN$_MISLOAFIL	Missing load file
ELN$_MISNODID	Missing node name or address
ELN$_MISPHYADR	Missing physical address
ELN$_MKD	Files–11 ACP could not mark file for deletion
ELN$_MOVVEC	Vector moved from shareable image
ELN$_MRS	Invalid maximum record size
ELN$_NEF	Not positioned to EOF on $PUT; sequential organization only
ELN$_NEGSIZE	The size of a dynamic aggregate is negative
ELN$_NEGSTRLEN	Negative string length specified
ELN$_NET	Network operation failed at remote node
ELN$_NI_ADDRNOTSET	Physical address could not be set
ELN$_NI_BSHORT	Buffer too short
ELN$_NI_CARRIERLOSS	Carrier loss during transmission
ELN$_NI_EXCESSCOLL	Excessive collisions; transmission stopped
ELN$_NI_ILLEGALCMD	Illegal command opcode
ELN$_NI_INVALIDBUFF	Invalid buffer specified
ELN$_NI_INVALIDCMD	Invalid command parameters
ELN$_NI_INVALIDPTDB	Invalid PTDB
ELN$_NI_INVALIDSAP	Invalid SAP value; even group SAPJ or odd individual SAP
ELN$_NI_INVLLCCLASS	Invalid LLC class specified for this user
ELN$_NI_LENGTH	Invalid length
ELN$_NI_LONG	Frame too long
ELN$_NI_NOTENABLED	User not enabled in a connection request
ELN$_NI_NOTUNIQUE	PTT, SAP, or PROTID not unique
ELN$_NI_PROMENABLED	Promiscuous mode already enabled

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_NI_RCVFAIL802TR	Receive failed; IEEE 802 packet truncated
ELN$_NI_RCVFAILNRSN	Receive failed; no reason given
ELN$_NI_SHORT	Frame too short
ELN$_NI_TOOMANYADR	Too many addresses defined
ELN$_NI_TOOMANYFQ	Too many FQs defined
ELN$_NI_TOOMANYPTDB	Too many PTDBs defined
ELN$_NI_TRANSFAILED	Transmission failed
ELN$_NI_UNKNOWNPTDB	Specified PTDB is unknown
ELN$_NI_XMTFAILLCOL	Transmit failed; late collision
ELN$_NI_XMTFAILNRSN	Transmit failed; no reason given
ELN$_NI_XMTFAILTIME	Transmit failed; transmit timeout
ELN$_NMF	No more files found
ELN$_NOBLOCKSPEC	Device driver indicated zero blocks on device
ELN$_NOD	Error in node name
ELN$_NOHANDLER	Exit handler not in system
ELN$_NOMODEM	No modem support
ELN$_NOMOREINFO	No information in data base
ELN$_NORESOURC	No resources available
ELN$_NORMAL	Operation successful
ELN$_NOSERVERS	No terminal servers known to service node
ELN$_NOSUCHENTRY	No matching entry found
ELN$_NOSUCHLINK	No such link
ELN$_NOSUCHOPTION	Hardware option not present
ELN$_NOSUCHPORT	No such port
ELN$_NOSUCHSERV	No such service
ELN$_NOTENUMSTR	String is not of the enumerated type
ELN$_OBSVEC	Obsolete termclass vectored routine

Table A-1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_ORG	Invalid file organization value
ELN$_PAOC	A packed array of type CHAR is too large to be used as a string
ELN$_PAOCASGN	Assignment of string of wrong length to packed array of type CHAR
ELN$_PES	Partial escape sequence
ELN$_PORTEXISTS	Port already exists
ELN$_PRED	Operand to PRED is too small
ELN$_PROBESIZE	Size of argument to PROBE_READ or PROBE_WRITE is greater than 65535 bytes
ELN$_PROTOCOL_FORMAT	DAP protocol error detected; message field contains invalid format
ELN$_PROTOCOL_INVALID	DAP protocol error detected; message field is invalid
ELN$_PRV	Insufficient privilege or file protection violation
ELN$_QUO	Error in quoted string
ELN$_RAC	Invalid record access mode
ELN$_RAT	Invalid record attributes
ELN$_RECEIVE	Size of message received is different from the size of the associated type of the data pointer
ELN$_RENAME_2	Rename; 2 different device names specified
ELN$_REQMAX	Maximum number of requests already queued
ELN$_REQUEST_OUTSTANDING	Modem event signaling request already exists
ELN$_RFA	Invalid record's file address (RFA)
ELN$_RFM	Invalid record format
ELN$_RLK	Target record currently locked by another stream
ELN$_RMV	Files–11 ACP remove function failed

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_RNF	Record not found
ELN$_RNL	Record not locked
ELN$_ROP	Invalid record options
ELN$_RSZ	Invalid record size
ELN$_SEND_RECEIVE	Send or receive failure
ELN$_SERVEXISTS	Service already exists
ELN$_SETASGN	Members present in the set source are out of range specified by target
ELN$_SETCONSTR	An expression in a set constructor is out of range
ELN$_SHR	Invalid file sharing (SHR) options
ELN$_SNE	File sharing not enabled
ELN$_STRLEN	A string length exceeds 32767
ELN$_SUBRASGN	The source value is out of the range of the target subrange
ELN$_SUBSCR	Array index value is out of range
ELN$_SUBSTR	Operand in a call to SUBSTR is out of range
ELN$_SUC	Operation successful
ELN$_SUCC	Operation in call to SUCC is too large
ELN$_SUCCESS	Operation completed successfully
ELN$_SUCCESS_ERROR	Error in DAP success message
ELN$_SUP	Network operation not supported
ELN$_SYN	File specification syntax error
ELN$_TAPE_DEVERROR	Device error occurred
ELN$_TAPE_DEVINUSE	Another process is using the device
ELN$_TAPE_DEVMOUNT	Device is already mounted
ELN$_TAPE_DIFLBLMNT	A volume with a different label was mounted
ELN$_TAPE_VOLNAMMSK	Specified volume's name is masked by another volume

Table A-1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Runtime Library	
ELN$_TERM_RECV	Terminator received
ELN$_TIMEOUT	Timeout occurred
ELN$_TMO	Timeout occurred
ELN$_TNS	Terminator not seen
ELN$_TRANSLATE	No translation exists for a character in the source specified with TRANSLATE
ELN$_TUTL_BLKSIZ	Invalid block size specified
ELN$_TUTL_INVCHRVOL	Invalid character in volume label
ELN$_TYP	Error in file type
ELN$_TYPECAST	Target type is larger than the variable being cast
ELN$_TYPEEXTENT	Corresponding type extents are not equal
ELN$_UNSUPPORT	Network operation not supported
ELN$_UNSUPPORTED	Driver received unsupported request
ELN$_UPI	UPI not set when sharing and BIO or BRO set
ELN$_VER	Error in version number
ELN$_VOL	No such volume
ELN$_WCC	Invalid wildcard context (WCC) value.
ELN$_WLD	Invalid wildcard operation
ELN$_ZEROSIZE	Size of the target specified with ZERO is greater than 65,535 bytes
File Service	
FLS$K_ALLFHDNOTMAP	Allocated file header not mapped
FLS$K_BADBLOCK	Bad block encountered; handling not implemented
FLS$K_BADFILEID	File number out of range for this volume
FLS$K_BADFILENAME	Bad file name for enter operation

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
File Service	
FLS$K_BADFILEVER	Bad version number for enter operation
FLS$K_BADPARAM	Bad input parameter
FLS$K_BADSBMBLK	Bad storage bitmap block specified in bitmap search
FLS$K_ERRDURDMT	Error during dismount; outstanding file open
FLS$K_ERREXTIDX	Error extending index file
FLS$K_ERRRDIDX	Error reading index file header
FLS$K_ERRWRTIDX	Error writing index file header
FLS$K_FILBLKNOTMAP	Attempt to read from or write to block not mapped in file
FLS$K_FILESTRUCT	Unsupported file structure level or unsupported ODS2 feature
FLS$K_ILLEGALEXT	Illegal extent specified in bitmap deallocation
FLS$K_ILLPTRCNT	Illegal pointer count specified during retrieval pointer creation
FLS$K_MAPCNTZERO	Attempt to create map pointer with zero block count
FLS$K_MOUNTED	Actual volume name is *name*
FORTRAN Runtime Library	
FOR$_ADJARRDIM	Adjustable array dimension error
FOR$_ATTACCNON	Attempt to access nonexistent record
FOR$_BACERR	BACKSPACE error
FOR$_CLOERR	CLOSE error
FOR$_DUPFILSPE	Duplicate file specifications
FOR$_ENDDURREA	End-of-file during read
FOR$_ENDFILERR	ENDFILE error
FOR$_ERRDURREA	Error during read
FOR$_ERRDURWRI	Error during write

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
FORTRAN Runtime Library	
FOR$_FILNAMSPE	File name specification error
FOR$_FILNOTFOU	File not found
FOR$_FINERR	FIND error
FOR$_FLOUNDEXC	Floating underflow exception
FOR$_FORVARMIS	Format and variable type mismatch
FOR$_INCFILORG	Inconsistent file organization
FOR$_INCOPECLO	Inconsistent OPEN and CLOSE parameters
FOR$_INCRECLEN	Inconsistent record length
FOR$_INCRECTYP	Inconsistent record type
FOR$_INFFORLOO	Infinite format loop
FOR$_INPCONERR	Input conversion error
FOR$_INPRECTOO	Input record too long
FOR$_INPSTAREQ	Input statement requires too much data
FOR$_INSVIRMEM	Insufficient virtual memory
FOR$_INVARGFOR	Invalid argument
FOR$_INVLOGUNI	Invalid logical unit number
FOR$_INVREFVAR	Invalid reference to variable in NAMELIST input
FOR$_KEYVALERR	Keyword value error in OPEN statement
FOR$_LISIO_SYN	List-directed I/O syntax error
FOR$_NO_CURREC	No currrent record
FOR$_NO_SUCDEV	No such device
FOR$_NOTFORSPE	Not a FORTRAN-specific error
FOR$_OPEFAI	Open failure
FOR$_OUTCONERR	Output conversion error
FOR$_OUTSTAOVE	Output statement overflows record
FOR$_RECIO_OPEN	Recursive I/O operation
FOR$_RECNUMOUT	Record number outside range

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
FORTRAN Runtime Library	
FOR$_REWERR	REWIND error
FOR$_REWRITERR	REWRITE error
FOR$_SEGRECFOR	Segmented record format error
FOR$_SPERECLOC	Specified record locked
FOR$_SYNERRFOR	Syntax error in format
FOR$_SYNERRNAM	Syntax error in NAMELIST input
FOR$_TOOMANREC	Too many records in I/O statement
FOR$_TOOMANVAL	Too many values for NAMELIST variable
FOR$_UNIALROPE	Unit already open
FOR$_UNLERR	UNLOCK error
FOR$_VFEVALERR	Variable format expression value error
FOR$_WRIREAFIL	Write to READONLY file
VAXELN Kernel	
KER$_AREA_EXISTS	Previous job created area
KER$_BAD_ACCESS_ CONTROL	Remote system rejected user name or password
KER$_BAD_COUNT	Bad parameter count
KER$_BAD_CREATE	Bad job or process creation
KER$_BAD_IMAGE_FORMAT	Unsupported program image format
KER$_BAD_LENGTH	Bad string parameter length
KER$_BAD_MESSAGE_SIZE	Bad message size
KER$_BAD_MODE	Bad access mode
KER$_BAD_STACK	Bad stack
KER$_BAD_STATE	Bad object state
KER$_BAD_TYPE	Bad object type
KER$_BAD_VALUE	Bad parameter value

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Kernel	
KER$_BAD_WRITE	Not enough free memory process's kernel or user stack
KER$_CONNECT_PENDING	Connect circuit pending
KER$_CONNECT_TIMEOUT	Connect circuit timeout
KER$_COUNT_OVERFLOW	Count overflow
KER$_COUNT_UNDERFLOW	Count underflow
KER$_DEVICE_CONNECTED	Device already connected
KER$_DISCONNECT	Circuit disconnected by partner
KER$_DUPLICATE	Duplicate name
KER$_EXPEDITED	Expedited message
KER$_KERNEL_STACK	Kernel stack not valid
KER$_MACHINECHK	Machine check
KER$_NO_ACCESS	No access to parameter
KER$_NO_DESTINATION	No destination port
KER$_NO_INITIALIZATION	No job initialization specified
KER$_NO_MAP_REGISTER	No I/O mapping register available
KER$_NO_MEMORY	No physical memory available
KER$_NO_MESSAGE	No message available
KER$_NO_OBJECT	No object table entry available
KER$_NO_PAGE_TABLE	No process page table available
KER$_NO_PATH_REGISTER	No data path register available
KER$_NO_POOL	No pool available
KER$_NO_PORT	No port available
KER$_NO_STATUS	No exit status value specified
KER$_NO_SUCH_DEVICE	No such device
KER$_NO_SUCH_IMAGE	No such image
KER$_NO_SUCH_NAME	No such name
KER$_NO_SUCH_PORT	No such port

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
VAXELN Kernel	
KER$_NO_SUCH_PROGRAM	No such program
KER$_NO_SUCH_SERVICE	No such service
KER$_NO_SYSTEM_PAGE	No system page table entries available
KER$_NO_VIRTUAL	No virtual address space available
KER$_POWER_SIGNAL	System power recovery is in progress
KER$_PROCESS_ATTENTION	Interprocess signal
KER$_QUIT_SIGNAL	Quit signal
KER$_SUCCESS	Operation completed successfully
KER$_TIME_NOT_SET	Time has not been set
KER$_UNREACHABLE	Remote system currently unreachable
General Runtime Library	
LIB$_AMBKEY	Ambiguous keyword
LIB$_AMBSYMDEF	Ambiguous symbol definition
LIB$_ATTCONSTO	Attempt to continue from stop
LIB$_ATTREQREF	Attach request refused
LIB$_BADBLOADR	Bad block address
LIB$_BADBLOSIZ	Bad block size
LIB$_BADCCC	Invalid compilation code
LIB$_BADSTA	Bad stack
LIB$_BADTAGVAL	Bad boundary tag value
LIB$_DECOVF	Decimal overflow
LIB$_DESSTROVF	Destination string overflow
LIB$_EF_ALRFRE	Event flag already free
LIB$_EF_ALRRES	Event flag already reserved
LIB$_EF_RESSYS	Event flag reserved to system
LIB$_EOMERROR	Compilation errors in module

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
General Runtime Library	
LIB$_EOMFATAL	Fatal compilation errors in module
LIB$_EOMWARN	Compilation warnings in module
LIB$_ERRROUCAL	Error in routine call
LIB$_FATERRLIB	Fatal error in library
LIB$_FLTOVF	Floating overflow
LIB$_FLTUND	Floating underflow
LIB$_GSDTYP	Invalid GSD record type in module
LIB$_ILLFMLCNT	Minimum argument count exceeds maximum for procedure in module
LIB$_ILLMODNAM	Invalid module name length for module
LIB$_ILLPSCLEN	Psect has invalid length in module
LIB$_ILLRECLEN	Invalid record length in module
LIB$_ILLRECLN2	Invalid record length
LIB$_ILLRECTYP	Invalid record type in module
LIB$_ILLRECTY2	Invalid record type
LIB$_ILLSYMLEN	Symbol has invalid length in module
LIB$_INPSTRTRU	Input string truncated
LIB$_INSEF	Insufficient event flags
LIB$_INSLUN	Insufficient logical unit numbers
LIB$_INSVIRMEM	Insufficient virtual memory
LIB$_INTLOGERR	Internal logic error
LIB$_INTOVF	Integer overflow
LIB$_INVARG	Invalid arguments
LIB$_INVARGORD	Invalid argument order
LIB$_INVCHA	Invalid character
LIB$_INVCLADSC	Invalid class descriptor
LIB$_INVCLADTY	Invalid class data type combination in descriptor

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
General Runtime Library	
LIB$_INVCVT	Invalid conversion
LIB$_INVDTYDSC	Invalid data type in descriptor
LIB$_INVFILSPE	Invalid file specification
LIB$_INVNBDS	Invalid numeric byte data string
LIB$_INVOPEZON	Invalid operation for zone
LIB$_INVSCRPOS	Invalid screen position
LIB$_INVSTRDES	Invalid string descriptor
LIB$_INVSYMNAM	Invalid symbol name
LIB$_INVTYPE	Invalid LIB$TPARSE state table entry
LIB$_IVTIME	Invalid time passed in or computed
LIB$_KEYALRINS	Key already inserted in tree
LIB$_KEYNOTFOU	Key not found in tree
LIB$_LUNALRFRE	Logical unit number already free
LIB$_LUNRESSYS	Logical unit number reserved to system
LIB$_NEGTIM	Negative time was computed
LIB$_NOEOM	Module does not contain end-of-module record
LIB$_NORMAL	Operation completed successfully
LIB$_NOSUCHSYM	No such symbol
LIB$_NOTFOU	Not found
LIB$_ONEDELTIM	At least one delta time is required
LIB$_ONEENTQUE	One entry in queue
LIB$_OUTSTRTRU	Output string truncated
LIB$_PAGLIMEXC	Page limit exceeded for zone
LIB$_PUSSTAOVE	Pushdown stack overflow
LIB$_QUEWASEMP	Queue was empty
LIB$_RECTOOSML	Data overflows object record in module
LIB$_ROPRAND	Reserved operand

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
General Runtime Library	
LIB$_SCRBUFOVF	Screen buffer overflow
LIB$_SECINTFAI	Secondary interlock failure in queue
LIB$_SEQUENCE	Invalid record sequence in module
LIB$_SEQUENCE2	Invalid record sequence
LIB$_SIGNO_ARG	Signal with no arguments
LIB$_STRIS_INT	String is interlocked
LIB$_STRLVL	Invalid object language structure level in module
LIB$_STRTRU	String truncated
LIB$_SYNTAXERR	String syntax error detected by LIB$TPARSE
LIB$_UNRKEY	Unrecognized keyword
LIB$_USEFLORES	Use of floating reserved operand
LIB$_WRONUMARG	Wrong number of arguments
Math Runtime Library	
MTH$_FLOOVEMAT	Floating-point overflow
MTH$_FLOUNDMAT	Floating-point underflow
MTH$_INVARGMAT	Invalid argument
MTH$_LOGZERNEG	Logarithm of zero or negative value
MTH$_SQUROONEG	Square root of negative value
MTH$_UNDEXP	Undefined exponentiation
MTH$_WRONUMARG	Wrong number of arguments
Language Independent Runtime Library	
OTS$_FATINTERR	Fatal internal error
OTS$_INPCONERR	Input conversion error
OTS$_INSVIRMEM	Insufficient virtual memory
OTS$_INTDATCOR	Internal data corrupted

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
Language Independent Runtime Library	
OTS$_INVSTRDES	Invalid string descriptor
OTS$_IO_CONCLO	I/O continued to closed file
OTS$_OUTCONERR	Output conversion error
OTS$_STRIS_INT	String is interlocked
OTS$_USEFLORES	Use of floating reserved operand
OTS$_WRONUMARG	Wrong number of arguments
Pascal Runtime Library	
PAS$_ACCMETINC	ACCESS_METHOD specified is incompatible with file
PAS$_AMBVALENU	Ambiguous value for enumerated type
PAS$_BUGCHECK	Internal consistency failure
PAS$_ERRDURCLO	Error during CLOSE
PAS$_ERRDURDIS	Error during DISPOSE
PAS$_ERRDURFIN	Error during FIND
PAS$_ERRDURGET	Error during GET
PAS$_ERRDURNEW	Error during NEW
PAS$_ERRDUROPE	Error during OPEN
PAS$_ERRDURPRO	Error during prompting
PAS$_ERRDURPUT	Error during PUT
PAS$_ERRDURRES	Error during RESET
PAS$_ERRDURREW	Error during REWRITE
PAS$_ERRDURWRI	Error during WRITELN
PAS$_FAIGETLOC	Failed to get locked component
PAS$_FILALRACT	File already active
PAS$_FILALRCLO	File already closed
PAS$_FILALROPE	File already open

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
Pascal Runtime Library	
PAS$_FILNAMREQ	File name required for this history or disposition
PAS$_FILNOTDIR	File not opened for direct access
PAS$_FILNOTFOU	File not found
PAS$_FILNOTGEN	File not in generation mode
PAS$_FILNOTINS	File not in inspection mode
PAS$_FILNOTOPE	File not open
PAS$_FILNOTTEX	File not a text file
PAS$_GENNOTALL	Generation mode not allowed for a READONLY file
PAS$_GETAFTEOF	GET attempted after end-of-file
PAS$_GOTO	Non-local GOTO requested
PAS$_GOTOFAILED	Non-local GOTO failed
PAS$_HALT	Program execution terminated
PAS$_INSVIRMEM	Insufficient virtual memory
PAS$_INVARGPAS	Invalid argument
PAS$_INVFILSYN	Invalid filename syntax
PAS$_INVFILVAR	Invalid file variable
PAS$_INVRECLEN	Invalid record length
PAS$_INVSYNENU	Invalid syntax for an enumerated value
PAS$_INVSYNINT	Invalid syntax for an integer value
PAS$_INVSYNREA	Invalid syntax for a real value
PAS$_INVSYNUNS	Invalid syntax for an unsigned value
PAS$_LINTOOLON	Line too long
PAS$_LINVALEXC	LINELIMIT value exceeded
PAS$_NEGDIGARG	Negative digits argument to BIN, HEX, or OCT not allowed
PAS$_NEGWIDDIG	Negative width or digits specification not allowed

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
Pascal Runtime Library	
PAS$_NOTVALTYP	Item not a value of specified type
PAS$_RECLENINC	RECORD_LENGTH specified is inconsistent with this file
PAS$_RECTYPINC	RECORD_TYPE specified is inconsistent with this file
PAS$_RESNOTALL	RESET not allowed on an unopened internal file
PAS$_REWNOTALL	REWRITE not allowed for a shared file
PAS$_TEXREQSEQ	Text files require sequential organization and access
PAS$_WRIINVENU	WRITE of an invalid enumerated value
Runtime System	
SS$_ACCVIO	Access violation
SS$_BREAK	Breakpoint fault
SS$_CMODUSER	Change mode to user trap
SS$_COMPAT	Compatibility mode fault
SS$_DECOVF	Arithmetic trap, decimal overflow
SS$_FLTDIV	Arithmetic trap, floating-point/decimal divide by zero
SS$_FLTDIV_F	Arithmetic trap, floating-point divide by zero
SS$_FLTOVF	Arithmetic trap, floating-point overflow
SS$_FLTOVF_F	Arithmetic trap, floating-point overflow
SS$_FLTUND	Arithmetic trap, floating-point underflow
SS$_FLTUND_F	Arithmetic trap, floating-point underflow
SS$_INSFRAME	Insufficient call frames to unwind
SS$_INTDIV	Arithmetic trap, integer divide by zero
SS$_INTOVF	Arithmetic trap, integer overflow
SS$_IVTIME	Invalid time

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
Runtime System	
SS$_NORMAL	Normal successful completion
SS$_NOSIGNAL	No signal currently active
SS$_OPCCUS	Opcode reserved to customer fault
SS$_OPCDEC	Opcode reserved to Digital fault
SS$_RADRMOD	Reserved addressing fault
SS$_ROPRAND	Reserved operand fault
SS$_SUBRNG	Arithmetic trap, subscript out of range
SS$_TBIT	T-bit pending trap
SS$_UNWIND	Unwind currently in progress
SS$_UNWINDING	Unwind already in progress
String Runtime Library	
STR$_DIVBY_ZER	Division by zero
STR$_FATINTERR	Fatal internal error
STR$_ILLSTRCLA	Invalid string class
STR$_ILLSTRPOS	Invalid string position
STR$_ILLSTRSPE	Invalid string specification
STR$_INSVIRMEM	Insufficient virtual memory
STR$_MATCH	Match found against input string
STR$_NEGSTRLEN	Negative string length
STR$_NOMATCH	No match found against input string
STR$_STRIS_INT	String interlocked
STR$_STRTOOLON	String too long
STR$_TRU	Truncation
STR$_WRONUMARG	Wrong number of arguments

Table A–1 (Cont.): Status Values/Exception Names

Name	Description
DECwindows Xlib Runtime Library	
X$_ERROREVENT	Error event received from server
X$_INSFMEM	Insufficient dynamic memory
X$_IOERROR	Xlib I/O error
X$_LIBABORT	Xlib fatal error
X$_OBSOLETE	Obsolete Xlib entry point referenced

Appendix B

Machine-Check Stack Frames

The VAXELN software supports optional error logging in a VAXELN target application. If you select error logging at system build time, an error log file is produced that you can analyze by using the VMS Error Log Utility. The reports generated by the VMS utility are primarily intended to assist Digital Customer Services personnel. (See the *VAXELN Development Utilities Guide* for more information on VAXELN error logging.)

Among the errors that can be logged when error logging is built into a VAXELN system are processor *machine checks*. A machine check is an exception that is reported when the processor or an external adapter detects an error. When a machine-check occurs, the processor pushes a *machine-check stack frame* onto an interrupt stack that consists of a count longword, an implementation-dependent number of error report longwords, a program counter (PC), and a process status longword (PSL). The count longword reports the number of bytes of error report pushed onto the stack. For example, if four longwords of error report are pushed onto the stack, the count longword will contain 16.

The initial processing of a machine check or interrupt is processor-specific. However, the VAXELN machine-check handler for all processor types determines the seriousness of a machine check, whether the machine check is fatal to a job or to the system, and how to respond based on the following:

- The nature of the machine check
- The access mode in which the machine check occurred

If the job or system can recover from a machine check, the machine-check handler places the machine-check stack frame in the error log file. If error logging is not enabled, you can locate and inspect the stack frame manually.

When a machine check places a system in a state from which it cannot recover, the machine-check handler checks the access mode in which the machine check occurred. If it occurred in user mode or kernel mode — at interrupt priority level (IPL) 0 — the handler reports a machine-check exception through the exception dispatching mechanism. (Unless the process has taken special action, process execution terminates.) If the machine check occurred in kernel mode at an elevated IPL, a fatal system bugcheck may occur, causing an orderly shut-down of the system.

When machine-check stack frames are not logged, you can look for the stack frame in the interrupt stack. This appendix describes how to locate and interpret machine-check stack frames manually. This material is presented to assist Digital Customer Services personnel.

NOTE

In order for you to use the procedures described in this appendix, the VAXELN system on which the machine check occurred must have a system console.

B.1 Obtaining a Machine-Check Stack Frame

When a machine check occurs in kernel mode at an elevated IPL, a VAXELN system attempts to display the entire current stack on the system console terminal (if the system has no console) in a raw format (address and contents only). If the stack is successfully displayed, you can locate the start of the machine-check stack frame, which follows the count longword on the stack. The value of the count longword depends on the target processor type; these values are shown in the uppermost portions of Figures B–1 to B–8. For example, on a VAX-11/730 processor, the machine-check stack frame follows a length parameter of *0000000C(hex)* on the stack.

If the failure is very serious, the attempt to display the current stack might not succeed. In that case, you must use console commands to manually examine the VAX computer's processor status longword (PSL), program counter (PC), stack pointer (SP), and in-memory stack. For example, on a VAX–11/750 processor, you would use the following commands:

```
>>>  E P        ; Get PSL
>>>  E/G F      ; Get PC
>>>  E/G E      ; Get SP
>>>  E/L (SP)   ; Get first (bottom) stack longword
>>>  E/L        ; Get next stack longword, then repeat E/L
```

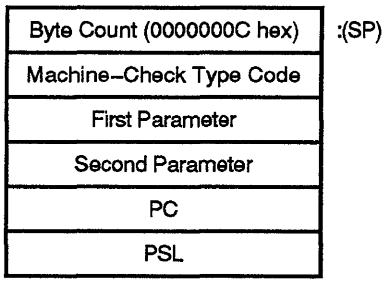
For examine (E) commands subsequent to the last one shown, the address being examined will increment automatically, allowing you to progress toward the top of the stack. The object is to locate the start of the machine-check stack frame, which on a VAX–11/750 processor follows a length parameter of *00000028*(hexadecimal) on the stack, ignoring the intervening locations, which contain parameters pushed by VAXELN bugcheck code and exceptions pushed on the stack after the machine check occurred.

Once you locate the start of the machine-check stack frame, you examine the stack frame and interpret it according to the frame layout for the particular processor. The remaining sections in this appendix give the machine-check stack frame formats for each supported target VAX processor.

B.2 Machine-Check Stack Frame for MicroVAX I Processors

Figure B–1 shows the information that is left on the stack when a machine check occurs on a MicroVAX I processor.

Figure B–1: Machine-Check Stack Frame for MicroVAX I Processors



MLO–004295

Table B–1 lists the possible values for the machine-check type code field.

Table B–1: Machine-Check Type Codes for MicroVAX I Processors

Code	Meaning
0	Memory-controller bug check[1]
1	Unrecoverable memory-read error[1]
2	Nonexistent memory[1]
3	Illegal I/O-space operation[1]
4	Unrecoverable PTE-read error[1]
5	Unrecoverable PTE-write error[1]
6	Control-store parity error[2]
7	Micromachine bug check[2]
8	Q22-bus vector read error[2]
9	Write parameter error[3]

[1]Bits<29,21:0> of the first parameter contain the corresponding bits of the physical address of the last memory reference, and the second parameter contains the address presented to the memory controller.

[2]Both parameters are 0.

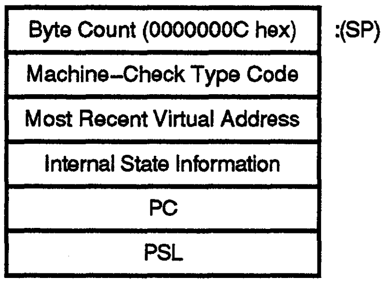
[3]The first parameter contains the virtual address that was being written, and the second parameter is 0.

B.3 Machine-Check Stack Frame for MicroVAX II and 2000, VAXstation II and 2000, and KA800 Processors

Figure B–2 shows the information that is left on the stack when a machine check occurs on one of the following processors:

- MicroVAX II
- VAXstation II
- MicroVAX 2000
- VAXstation 2000
- KA800

Figure B–2: Machine-Check Stack Frame for MicroVAX II and 2000, VAXstation II and 2000, and KA800 Processors



Byte Count (0000000C hex)
:(SP)
Machine–Check Type Code
Most Recent Virtual Address
Internal State Information
PC
PSL

MLO–004296

Table B–2 lists the possible values for the machine-check type code field.

Table B–2: Machine-Check Type Codes for MicroVAX II and 2000, VAXstation II and 2000, and KA800 Processors

Code	Meaning
1	Impossible microcode state (FSD)
2	Impossible microcode state (SSD)
3	Undefined FPU error code 0
4	Undefined FPU error code 7
5	Undefined memory management status (TB miss)
6	Undefined memory management status (M = 0)
7	Process PTE in P0 space
8	Process PTE in P1 space
9	Undefined interrupt ID code
80	Read bus error, address parameter is virtual
81	Read bus error, address parameter is physical
82	Write bus error, address parameter is virtual
83	Write bus error, address parameter is physical

B.4 Machine-Check Stack Frame for rtVAX 300, MicroVAX 3*nnn* Series, VAXstation 3100, 3200, and 3500, and VAX 6000–2*nn* and 6000–3*nn* Series Processors

Figure B–3 shows the information that is left on the stack when a machine check occurs on one of the following processors:

- rtVAX 300
- MicroVAX 3*nnn*
- VAXstation 3100
- VAXstation 3200

- VAXstation 3500
- VAX 6000–2*nn*
- VAX 6000–3*nn*

Figure B–3: Machine-Check Stack Frame for rtVAX 300, MicroVAX 3*nnn* Series, VAXstation 3100, 3200, and 3500, and VAX 6000–2*nn* and 6000–3*nn* Series Processors

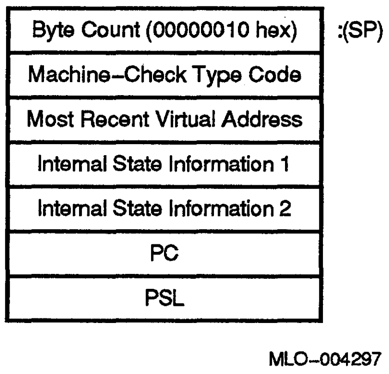

Byte Count (00000010 hex)
:(SP)
Machine–Check Type Code
Most Recent Virtual Address
Internal State Information 1
Internal State Information 2
PC
PSL
MLO–004297

Table B–3 lists the possible values for the machine-check type code field.

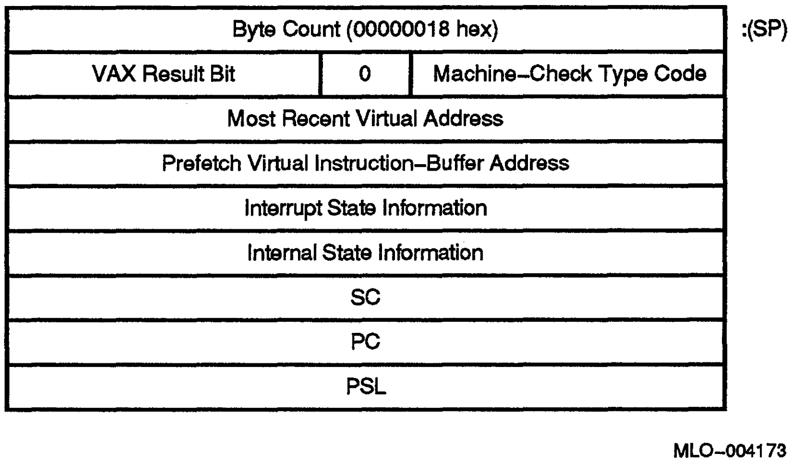
Table B–3: Machine-Check Type Codes for rtVAX 300, MicroVAX 3*nnn* Series, VAXstation 3100, 3200, and 3500, and VAX 6000–2*nn* and 6000–3*nn* Series Processors

Code	Meaning
1	FPU detected protocol error
2	FPU detected reserved instruction
3	FPU error of unknown origin
4	FPU error of unknown origin
5	Process PTE address in P0 space (TB miss)
6	Process PTE address in P1 space (TB miss)
7	Process PTE address in P0 space (M = 0)
8	Process PTE address in P1 space (M = 0)
9	Undefined IPL
A	Impossible MOVC state detected
80	Read error
81	Read error
82	Write error
83	Write error

B.5 Machine-Check Stack Frame for VAX 6000–4*nn* Series Processors

Figure B–4 shows the information that is left on the stack when a machine check occurs on a VAX 6000–400 series processor.

Figure B–4: Machine-Check Stack Frame for VAX 6000–4*nn* Series Processors



Byte Count (00000018 hex)
:(SP)
VAX Result Bit
0
Machine–Check Type Code
Most Recent Virtual Address
Prefetch Virtual Instruction–Buffer Address
Interrupt State Information
Internal State Information
SC
PC
PSL
MLO–004173

Table B–4 possible values for the machine-check type code field.

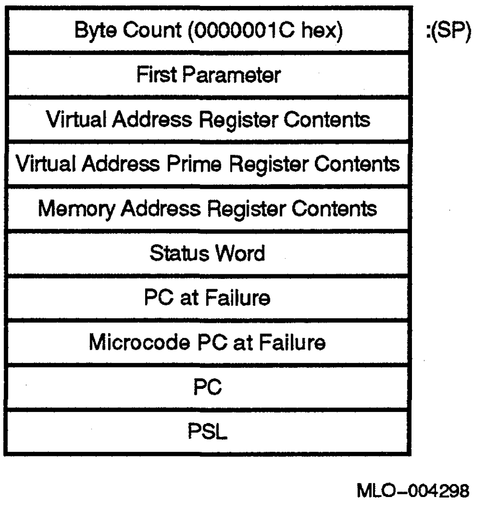
Table B–4: Machine-Check Type Codes for VAX 6000–4*nn* Series Processors

Code	Meaning
1	Protocol error during F-chip operand/result transfer
2	F-chip detected invalid opcode
3	F-chip detected operand parity error
4	F-chip returned unknown status
5	F-chip result parity error
8	TB miss status generated in ACV/TNV microflow
9	TB hit status generated in ACV/TNV microflow
10	Undefined INT.ID value during interrupt service
11	Undefined state bit combination in MOVCx
12	Undefined trap code produced by the IBOX
13	Undefined control store address reached
16	P-cache tag or data parity error during read
17	DAL bus or data parity error during read
18	DAL bus error on write or clear write buffer
19	Undefined bus error microtrap
20	Vector unit error

B.6 Machine-Check Stack Frame for VAX 8200 and 8250 Processors

Figure B–5 shows the information that is left on the stack when a machine check occurs on a VAX 8200 or 8250 processor.

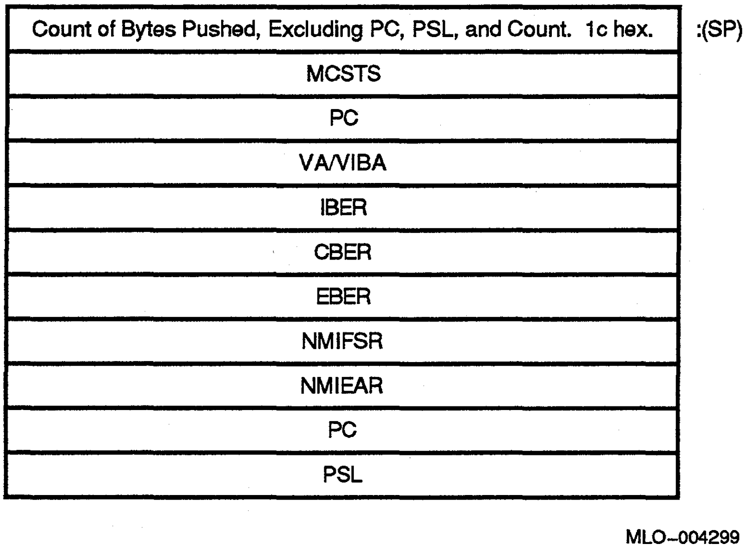
Figure B–5: Machine-Check Stack Frame for VAX 8200 and 8250 Processors


Byte Count (0000001C hex)
:(SP)
First Parameter
Virtual Address Register Contents
Virtual Address Prime Register Contents
Memory Address Register Contents
Status Word
PC at Failure
Microcode PC at Failure
PC
PSL
MLO–004298

B.7 Machine-Check Stack Frame for VAX 8500, 8550, 8700, 8800, and 8810 Processors

Figure B–6 shows the information that is left on the stack when a machine check occurs on a VAX 8500, 8550, 8700, 8800, or 8810 processor.

Figure B–6: Machine-Check Stack Frame for VAX 8500, 8550, 8700, 8800, and 8810 Processors



Count of Bytes Pushed, Excluding PC, PSL, and Count. 1c hex.
:(SP)
MCSTS
PC
VA/VIBA
IBER
CBER
EBER
NMIFSR
NMIEAR
PC
PSL
MLO–004299

Table B–5 lists the offset value and contents for each field in the stack frame.

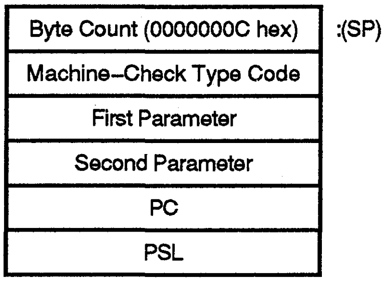
Table B–5: Machine-Check Stack Frame Contents for VAX 8500, 8550, 8700, 8800, and 8810 Processors

Mnemonic	Offset	Contents
COUNT	00	Count of bytes pushed, excluding PC, PSL, and count
MCSTS	04	Machine-check status
PC	08	Current PC
VA/VIBA	0C	Virtual address/virtual instruction-buffer address
IBER	10	IBOX error
CBER	14	CBOX error
EBER	18	EBOX error
NMIFSR	1C	NMI fault summary
NMIEAR	20	NMI error address
PC	24	PC of faulted opcode
PSL	28	Processor status longword

B.8 Machine-Check Stack Frame for VAX–11/730 Processors

Figure B–7 shows the information that is left on the stack when a machine check occurs on a VAX–11/730 processor.

Figure B–7: Machine-Check Stack Frame for VAX–11/730 Processors



MLO–004295

Table B–6 lists the possible values for the machine-check error type code field.

Table B–6: Machine-Check Error Type Codes for VAX–11/730 Processors

Code	Meaning
0	Microcode should not be here. If the first parameter is 0, no other information is available. If the first parameter is 2, the problem was inability to write back a PTE<M> bit. If the parameter is 3, the problem was a bad 8085 interrupt. The second parameter is always 0.
1	Translation-buffer parity error. The first parameter is the bad value from the TB. PFN is in bits<23:0>. PTE<V>, the protection code, and PTE<M> are in bits<31:26>. TB valid bit is in bit<25>. The second parameter is the virtual address referenced.

Table B–6 (Cont.): Machine-Check Error Type Codes for VAX–11/730 Processors

Code	Meaning
3	Impossible value in memory CSR. The first parameter is the virtual address referenced. The second parameter is the bad value of the CSR.
4	Fast interrupt without support. A fast interrupt was requested and no microcode was loaded to handle it. Both parameters are 0.
5	FPA parity error. The FPA control store had a parity error. The first parameter has parity error summary in bit<0>, group 0 parity in bit<1>, group 1 parity in bit<2>, and is unpredictable in bits<31:3>. The second parameter is 0.
6	Error on SPTE read. The first parameter is the physical address of the SPTE. The second parameter contains the error syndrome bits.
7	Uncorrectable ECC error. The first parameter is the physical address of the reference. The second parameter contains the error syndrome bits.
8	Nonexistent memory. The first parameter is the physical address referenced. The second parameter is 0.
9	Unaligned or nonlongword reference to I/O space. The first parameter is the physical address referenced. The second parameter is 0.
A	Illegal I/O-space address. The first parameter is the physical address referenced. The second parameter is 0.
B	Illegal UNIBUS reference. The first parameter is the physical address referenced. The second parameter is 0.

B.9 Machine-Check Stack Frame for VAX–11/750 Processors

Figure B–8 shows the information that is left on the stack when a machine check occurs on a VAX–11/750 processor.

Figure B–8: Machine-Check Stack Frame for VAX–11/750 Processors

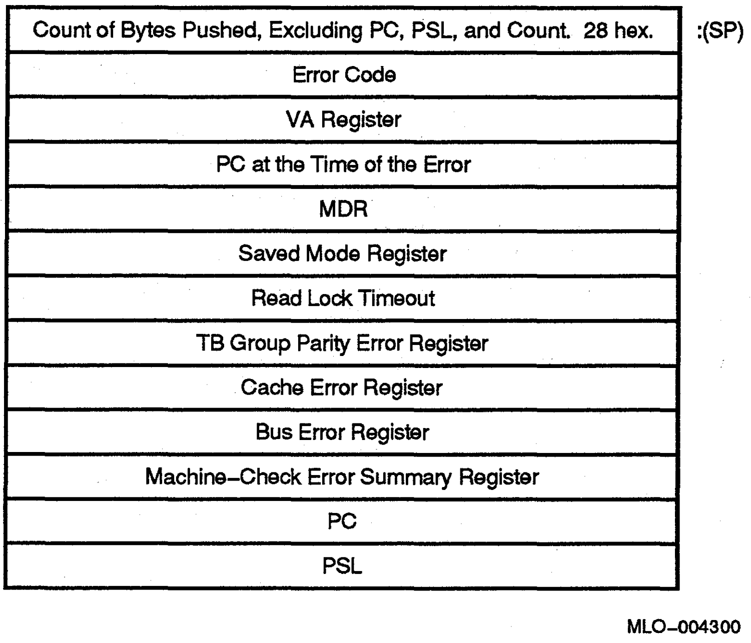

Count of Bytes Pushed, Excluding PC, PSL, and Count. 28 hex.
:(SP)
Error Code
VA Register
PC at the Time of the Error
MDR
Saved Mode Register
Read Lock Timeout
TB Group Parity Error Register
Cache Error Register
Bus Error Register
Machine–Check Error Summary Register
PC
PSL
MLO–004300

Table B–7 lists the possible values for the machine-check error type code field.

Table B–7: Machine-Check Error Codes for VAX–11/750 Processors

Code	Meaning
1	Control-store parity error
2	Translation-buffer parity error, bus error, or cache parity error
6	"Microcode should not be here" error
7	"Unused IRD ROM slot" error

Appendix C

VMS Emulation Routines

The VAXELN Toolkit supports subsets of the VMS system services and Runtime Library routines to simplify the task of porting VMS programs to the VAXELN environment. *System services* are procedures that the VMS operating system uses to control resources that are available to processes, provide for communication among processes, and perform basic operating system functions, such as coordination of input/output operations. The Runtime Library routines are commonly used to perform a wide variety of operations. You can call the supported system services and Runtime Library routines from your VAXELN Pascal, VAX C, or FORTRAN programs using the standard VAX procedure-calling conventions.

This appendix provides a summary of the supported emulation routines (see Section C.1), explains how to call the routines (see Section C.2), and describes the following:

- VMS system service emulation routines, Section C.3
- LIB$ emulation routines, Section C.4
- STR$ emulation routines, Section C.5

C.1 VMS Emulation Routine Summary

This section summarizes the VAXELN Toolkit's VMS emulation routine support. Table C–1 summarizes the supported system services. Table C–2 provides a summary of the supported Runtime Library routines.

Table C–1: VMS System Service Emulation Routines

Routine	Function
SYS$ASCTIM	Convert binary time to ASCII string
SYS$GETTIM	Get the current system time
SYS$UNWIND	Unwind the procedure call stack

For brief descriptions of the supported system services, see Section C.3. For detailed descriptions, see the *VMS System Services Reference Manual*.

The VMS Runtime Library routines are grouped as facilities according to the tasks they perform. The VAXELN Toolkit supports a subset of LIB$ facility routines and an STR$ facility routine. LIB$ facility routines provide access to VMS components such as system services and VAX machine instructions and perform such functions as the following:

- Get records from devices
- Manipulate strings
- Convert data types for I/O
- Allocate resources
- Get system information
- Signal exceptions
- Enable detection of hardware exceptions

The STR$ facility provides string manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings.

Table C–2: VMS Runtime Library Emulation Routines

Routine	Function
LIB$ Facility Routines	
LIB$ADD_TIMES	Add two quadword times
LIB$ADDX	Add two multiple-precision binary numbers

Table C–2 (Cont.): VMS Runtime Library Emulation Routines

Routine	Function
LIB$ Facility Routines	
LIB$ANALYZE_SDESC	Analyze a string descriptor
LIB$CREATE_USER_VM_ZONE	Create a user-defined storage zone
LIB$CREATE_VM_ZONE	Create a new storage zone
LIB$CVT_DTB	Convert ASCII decimal number to binary
LIB$CVT_HTB	Convert ASCII hexadecimal number to binary
LIB$CVT_OTB	Convert ASCII octal number to binary
LIB$DELETE_VM_ZONE	Delete virtual memory zone
LIB$EDIV	Perform an extended-precision divide
LIB$EMUL	Perform an extended-precision multiply
LIB$FLT_UNDER	Enable or disable floating-point underflow detection
LIB$FREE_VM	Free virtual memory from the program region
LIB$FREE_VM_PAGE	Free virtual memory page
LIB$GET_INPUT	Get a line from SYS$INPUT
LIB$GET_VM[1]	Allocate virtual memory
LIB$GET_VM_PAGE[1]	Get a virtual memory page
LIB$INT_OVER	Enable or disable integer overflow detection
LIB$LEN	Return the length of a string as a longword
LIB$MATCH_COND	Match condition values
LIB$MULTF_DELTA_TIME	Multiply delta time by scalar
LIB$MULT_DELTA_TIME	Multiply delta time by F_floating scalar
LIB$PUT_OUTPUT	Put a line in SYS$OUTPUT
LIB$RESET_VM_ZONE	Reset virtual memory zone

[1]Differs from VMS routine.

Table C–2 (Cont.): VMS Runtime Library Emulation Routines

Routine	Function
LIB$ Facility Routines	
LIB$SCOPY_DXDX	Copy source string by descriptor to destination
LIB$SCOPY_R_DX	Copy source string by reference to destination
LIB$SIGNAL	Signal exception condition
LIB$SIG_TO_RET	Convert signaled message to a return status
LIB$STOP	Stop execution and signal the condition
LIB$SUBX	Perform multiple-precision binary subtraction
LIB$SUB_TIMES	Subtract two quadword times
STR$ Facility Routines	
STR$ANALYZE_SDESC	Analyze a string descriptor

For brief descriptions of the supported Runtime Library routines, see Sections C.4 and C.5. For detailed descriptions, see *VMS RTL Library (LIB$) Manual* and *VMS RTL String Manipulation (STR$) Manual*.

C.2 Calling VMS Emulation Routines

The VMS systems services and Runtime Library routines are external routines that accept arguments. The VAXELN Pascal, VAX C, and VAX FORTRAN languages each provide a mechanism for calling external procedures and for passing arguments to those procedures. The specific mechanisms and the associated terminology for the different languages vary. For example, FORTRAN programs invoke external routines with CALL statements or function references.

The call formats for the supported system services and Runtime Library routines are summarized in Sections C.3 to C.5. The *VMS System Services Reference Manual*, *VMS RTL Library (LIB$) Manual*, and *VMS RTL String Manipulation (STR$) Manual* provide detailed routine descriptions that indicate how arguments are to be passed and describe routine-specific data structures.

You must pass arguments to a routine in the order shown in the documented call formats. Each argument has four characteristics: VMS usage, data type, access type, and passing mechanism. These characteristics are described in the *VMS System Services Reference Manual* and the *Introduction to the VMS Run-Time Library*.

Some arguments are optional and are indicated with square brackets ([ ]). In VAXELN Pascal and FORTRAN programs, you can omit such arguments at the end of an argument list. If an optional argument is not at the end of the argument list, you must either pass a zero by value or use a comma as a place holder to indicate the position of the omitted argument. In C programs, you must specify all arguments. For optional arguments you choose not to specify, you must supply the value NULL.

The following examples show how to call external routines from VAXELN Pascal, VAX C, and VAX FORTRAN programs. For language-specific information about calling external routines, see the appropriate language documentation.

VAXELN Pascal

```
MODULE emul;

FUNCTION LIB$EMUL (VAR multiplier: INTEGER;
                   VAR multiplicand: INTEGER;
                   VAR addend: INTEGER;
                   VAR product: LARGE_INTEGER):
                   INTEGER; EXTERNAL;

PROGRAM emul_prog( INPUT, OUTPUT );

VAR
  multiplier:  INTEGER;
  multiplicand:  INTEGER;
  addend:  INTEGER;
  product:  LARGE_INTEGER;
  status: INTEGER;
```

```
BEGIN
  multiplier := 4096;
  multiplicand := 268435456;
  addend := 0;
  status := LIB$EMUL(multiplier, multiplicand, addend, product);
  IF ODD(status) THEN
    BEGIN
      .
      .
      .
    END;
  ELSE
    WRITELN('Error calling LIB$EMUL');
END;
END.
```

VAX C

```
#include $vaxelnc

main( )
{
  int multiplier, multiplicand, addend, status;
  int lib$emul();
  LARGE_INTEGER product;

  multiplier = 4096;
  multiplicand = 268435456;
  addend = 0;
  status = lib$emul(&multiplier, &multiplicand, &addend, &product);

  if (status == TRUE)
  .
  .
  .
  else
  .
  .
  .
}
```

VAX FORTRAN

```
C
C This FORTRAN program demonstrates how to use LIB$EDIV.
C
 INTEGER divisor,dividend(2),quotient,remainder
```

```
C
C
C
 divisor = 4096
 dividend(1) = '12345678'x
 dividend(2) = '00000001'x
C
C
C
 return = LIB$EDIV(divisor,dividend,quotient,remainder)
 TYPE *,'The longword integer quotient of 4600387192/4096 is:'
 TYPE *,'    ',quotient
 TYPE *,'The longword integer remainder of 4600387192/4096 is:'
 TYPE *,'    ',remainder
 END
```

C.3 VMS System Service Emulation Routine Descriptions

This section briefly describes the VMS system services that the VAXELN Toolkit supports. For details, see the *VMS System Services Reference Manual.*

SYS$ASCTIM — Convert Binary Time to ASCII String

The SYS$ASCTIM system service converts an absolute or delta time from 64-bit system time to an ASCII string.

Call Format	Type	Access	Mechanism
SYS$ASCTIM			
[*timlen*]	Word (unsigned)	Write only	By reference
,*timbuf*	Character-coded text string	Write only	By descriptor
[,*timadr*]	Quadword (unsigned)	Read only	By reference
[,*cvtflg*]	Longword (unsigned)	Read only	By value

The *timbuf* argument specifies the buffer into which the ASCII string is to be written. The optional argument *timlen* receives the length (in bytes) of the ASCII string that the system service returns. The optional arguments *timadr* and *cvtflg* specify the time value the system service is to convert and a conversion indicator that specifies the date and time fields the system service is to return, respectively.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By reference

SYS$GETTIM — Get Time

The SYS$GETIM system service returns the current system time in 64-bit format.

Call Format	Type	Access	Mechanism
SYS$GETTIM			
timadr	Quadword (unsigned)	Write only	By reference

The *timadr* argument receives the current time in 64-bit format.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

SYS$UNWIND — Unwind Procedure Call Stack

The SYS$UNWIND system service removes a specified number of call frames from the procedure call stack. Optionally, it can return control to a new program counter (PC) after unwinding the stack. The SYS$UNWIND service is intended to be called from within a condition-handling routine.

Call Format	Type	Access	Mechanism
SYS$UNWIND			
[*depadr*]	Longword (unsigned)	Read only	By reference
[,*newpc*]	Longword (unsigned)	Read only	By reference

The optional arguments *depadr* and *newpc* specify the depth to which the procedure call stack is to unwind and the new value for the PC, respectively. The new PC value replaces the current value of the PC in the call frame of the procedure that receives control when the unwind operation is complete.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

C.4 LIB$ Emulation Routine Descriptions

This section briefly describes the LIB$ Runtime Library routines that the VAXELN Toolkit supports. For details, see the *VMS RTL Library (LIB$) Manual.*

LIB$ADD_TIMES — Add Two Quadword Times

The LIB$ADD_TIMES routine adds two time values in internal-time format.

Call Format	Type	Access	Mechanism
LIB$ADD_TIMES			
time1	Quadword (unsigned)	Read only	By reference
,time2	Quadword (unsigned)	Read only	By reference
,resultant-time	Quadword (unsigned)	Write only	By reference

The *time1* and *time2* arguments specify the times to be added. The *resultant-time* argument receives the sum.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$ADDX — Add Two Multiple-Precision Binary Numbers

The LIB$ADDX routine adds two signed two's complement integers of arbitrary length.

Call Format	Type	Access	Mechanism
LIB$ADDX			
addend-array	Unspecified	Read only	By reference (array)
,augend-array	Unspecified	Read only	By reference (array)
,resultant-array	Unspecified	Write only	By reference (array)
[*,array-length*]	Longword integer (signed)	Read only	By reference

The *addend-array* and *augend-array* arguments specify the multiple-precision, signed two's complement integers to be added. The *resultant-array* argument receives the multiple-precision, signed two's complement integer result of the addition. The optional argument *array-length* specifies the length of the arrays on which the routine is to operate.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$ANALYZE_SDESC — Analyze String Descriptors

The LIB$ANALYZE_SDESC routine extracts the length and the address at which the data starts for a variety of string descriptor classes.

Call Format	Type	Access	Mechanism
LIB$ANALYZE_SDESC			
input-descriptor	Character string	Read only	By descriptor
,data-length	Word (unsigned)	Write only	By reference
,data-address	Longword (unsigned)	Write only	By reference

The *input-descriptor* argument specifies the input descriptor from which the routine is to extract the data's length and starting address. The *data-length* and *data-address* arguments specify the length and starting address of the data, respectively. The routine extracts the length and address from the input descriptor.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$CREATE_USER_VM_ZONE — Create User-Defined Storage Zone

The LIB$CREATE_USER_VM_ZONE routine creates a new user-defined storage zone.

Call Format	Type	Access	Mechanism
LIB$CREATE_USER_VM_ZONE			
zone-id	Longword (unsigned)	Write only	By reference
[*,user-argument*]	Longword (unsigned)	Read only	By reference
[*,user-allocation-procedure*]	Procedure entry mask	Function call	By value
[*,user-deallocation-procedure*]	Procedure entry mask	Function call	By value
[*,user-reset-procedure*]	Procedure entry mask	Function call	By value
[*,user-delete-procedure*]	Procedure entry mask	Function call	By value
[*,zone-name*]	Character string	Read only	By descriptor

The *zone-id* argument specifies a zone identifier. The optional argument *user-argument* specifies a user argument. The optional arguments *user-allocation-procedure*, *user-deallocation-procedure*, *user-reset-procedure*, and *user-delete-procedure* specify user allocation, deallocation, reset zone, and delete zone routines, respectively. The optional *zone-name* argument specifies a name to be associated with the zone being created.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$CREATE_VM_ZONE — Create a New Zone

The LIB$CREATE_VM_ZONE routine creates a new storage zone according to specified arguments.

Call Format	Type	Access	Mechanism
LIB$CREATE_VM_ZONE			
zone-id	Longword (unsigned)	Write only	By reference
[,*algorithm*]	Longword integer (signed)	Read only	By reference
[,*algorithm-argument*]	Longword integer (signed)	Read only	By reference
[,*flags*]	Longword (unsigned)	Read only	By reference
[,*extend-size*]	Longword integer (signed)	Read only	By reference
[,*initial-size*]	Longword integer (signed)	Read only	By reference
[,*block-size*]	Longword integer (signed)	Read only	By reference
[,*alignment*]	Longword integer (signed)	Read only	By reference
[,*page-limit*]	Longword integer (signed)	Read only	By reference
[,*smallest-block-size*]	Longword integer (signed)	Read only	By reference
[,*zone-name*]	Character string	Read only	By descriptor
[,*number-of-areas*]	Longword (signed)	Read only	By reference
[,*get-page*]	Procedure entry mask	Read only	By value
[,*free-page*]	Procedure entry mask	Read only	By value

The *zone-id* argument specifies a zone identifier. The optional *algorithm* and *algorithm-argument* arguments specify the algorithm and algorithm arguments to be used to create the new zone. The optional *flags* argument specifies flag bits that control various options. The optional arguments *extend-size*, *initial-size*, *block-size*, *alignment*, *page-limit*, *smallest-block-size*, *zone-name*, and *number-of-areas* specify the zone's extend size, initial size, block size, block alignment, maximum page limit, smallest block size, name, and number of memory areas, respectively. The optional *get-page* and *free-page* arguments specify routines that allocate and deallocate pages of memory.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$CVT_DTB — Convert Numeric Decimal Text to Binary

The LIB$CVT_DTB routine returns a binary representation of the ASCII text string representation of a decimal number.

Call Format	Type	Access	Mechanism
LIB$CVT_DTB			
byte-count	Longword integer (signed)	Read only	By value
,numeric-string	Character string	Read only	By reference
,result	Longword integer (signed)	Write only	By reference

The *byte-count* argument specifies the byte count of the input ASCII text string. The *numeric-string* argument specifies the ASCII text string representation of a decimal number that the routine is to convert to binary representation. The *result* argument receives the binary representation of the input string.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$CVT_HTB — Convert Numeric Hexadecimal Text to Binary

The LIB$CVT_HTB routine returns a binary representation of the ASCII text string representation of a hexadecimal number.

Call Format	Type	Access	Mechanism
LIB$CVT_HTB			
byte-count	Longword integer (signed)	Read only	By value
,numeric-string	Character string	Read only	By reference
,result	Longword integer (signed)	Write only	By reference

The *byte-count* argument specifies the byte count of the input ASCII text string. The *numeric-string* argument specifies the ASCII text string representation of a hexadecimal number that the routine is to convert to binary representation. The *result* argument receives the binary representation of the input string.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$CVT_OTB — Convert Numeric Octal Text to Binary

The LIB$CVT_OTB routine returns a binary representation of the ASCII text string representation of an octal number.

Call Format	Type	Access	Mechanism
LIB$CVT_OTB			
byte-count	Longword integer (signed)	Read only	By value
,numeric-string	Character string	Read only	By reference
,result	Longword integer (signed)	Write only	By reference

The *byte-count* argument specifies the byte count of the input ASCII text string. The *numeric-string* argument specifies the ASCII text string representation of a octal number that the routine is to convert to binary representation. The *result* argument receives the binary representation of the input string.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$DELETE_VM_ZONE — Delete Virtual Memory Zone

The LIB$DELETE_VM_ZONE routine deletes a zone and returns all pages owned by the zone to the processwide page pool.

Call Format	Type	Access	Mechanism
LIB$DELETE_VM_ZONE			
zone-id	Longword (unsigned)	Read only	By reference

The *zone-id* argument specifies a zone identifier.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$EDIV — Extend-Precision Divide

The LIB$EDIV routine performs extended-precision division. This routine makes the VAX EDIV instruction available as a callable routine.

Call Format	Type	Access	Mechanism
LIB$EDIV			
longword-integer-divisor	Longword integer (signed)	Read only	By reference
,quadword-integer-dividend	Quadword integer (signed)	Read only	By reference
,longword-integer-quotient	Longword integer (signed)	Write only	By reference
,remainder	Longword integer (signed)	Write only	By reference

The *longword-integer-divisor* and *quadword-integer-dividend* arguments specify the divisor and dividend. The *longword-integer-quotient* and *remainder* arguments receive the quotient and remainder.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$EMUL — Extend-Precision Multiply

The LIB$EMUL routine performs extended-precision multiplication. This routine makes the VAX EMUL instruction available as a callable routine.

Call Format	Type	Access	Mechanism
LIB$EMUL			
longword-integer-multiplier	Longword integer (signed)	Read only	By reference
,longword-integer-multiplicand	Longword integer (signed)	Read only	By reference
,addend	Longword integer (signed)	Read only	By reference
,product	Quadword integer (signed)	Write only	By reference

The *longword-integer-multiplier*, *longword-integer-multiplicand*, and *addend* arguments specify the multiplier, multiplicand and addend, respectively. The *product* argument receives the product.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$FLT_UNDER — Floating-Point Underflow Detection

The LIB$FLT_UNDER routine enables or disables floating-point underflow detection for the calling routine activation and returns the previous setting as a function value.

Call Format	Type	Access	Mechanism
LIB$FLT_UNDER			
new-setting	Longword (unsigned)	Read only	By reference

The *new-setting* argument specifies the new floating-point underflow enable setting.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$FREE_VM — Free Virtual Memory

The LIB$FREE_VM routine deallocates an entire block of contiguous bytes in the program region that were allocated by a previous call to LIB$GET_VM. The arguments passed are the same as for LIB$GET_VM.

Call Format	Type	Access	Mechanism
LIB$FREE_VM			
number-of-bytes	Longword integer (signed)	Read only	By reference
,base-address	Longword (unsigned)	Read only	By reference
[,zone-id]	Longword (unsigned)	Read only	By reference

The *number-of-bytes* and *base-address* arguments specify the number of contiguous bytes to be deallocated and the address of the first byte to be deallocated, respectively. The optional argument *zone-id* specifies a zone identifier created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$FREE_VM_PAGE — Free Virtual Memory Page

The LIB$FREE_VM_PAGE routine deallocates a block of contiguous pages that were allocated by a previous call to LIB$GET_VM_PAGE.

Call Format	Type	Access	Mechanism
LIB$FREE_VM_PAGE			
number-of-pages	Longword integer (signed)	Read only	By reference
,base-address	Longword (unsigned)	Read only	By reference

The *number-of-pages* argument specifies the number of contiguous pages to be deallocated. The *base-address* argument specifies the address of the first byte of the first page to be deallocated.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$GET_INPUT — Get Line from SYS$INPUT

The LIB$GET_INPUT routine gets one record of ASCII text from the current controlling input device, specified by SYS$INPUT.

Call Format	Type	Access	Mechanism
LIB$GET_INPUT			
resultant-string	Character string	Write only	By descriptor
[*,prompt-string*]	Character string	Read only	By descriptor
[*,resultant-length*]	Word (unsigned)	Write only	By reference

The *resultant-string* argument receives a string from the input device. The optional argument *prompt-string* specifies a prompt message that is to be displayed on the controlling terminal. The optional *resultant-length* argument receives a value indicating the number of bytes written into the *resultant-string* argument.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$GET_VM — Allocate Virtual Memory

The LIB$GET_VM routine allocates a specified number of contiguous bytes in the program region and returns the virtual address of the first byte allocated.

When calling the LIB$GET_VM routine, you can specify the address of a longword that contains a zone identifier. If you do not specify this argument or if the longword contains the value 0, the default zone is used. The default zone has a set of attributes, two of which are the initial size and the area extension size. The values for these attributes differ for VAXELN systems. For VAXELN systems, the initial size is zero pages, and the area extension size is two pages. If you need to allocate over 1000 pages in a single request, you should call KER$ALLOCATE_MEMORY instead of LIB$GET_VM_PAGE.

Call Format	Type	Access	Mechanism
LIB$GET_VM			
number-of-bytes	Longword integer (signed)	Read only	By reference
,base-address	Longword (unsigned)	Write only	By reference
[*,zone-id*]	Longword (unsigned)	Read only	By reference

The *number-of-bytes* argument specifies the number of contiguous bytes the routine is to allocate. The *base-address* argument receives the first virtual address of the contiguous block of bytes the routine allocates.

The optional argument *zone-id* specifies the address of a longword that contains a zone identifier created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$GET_VM_PAGE — Get Virtual Memory Page

The LIB$GET_VM_PAGE routine allocates a specified number of contiguous pages of memory in the program region and returns the virtual memory address of the first page allocated.

LIB$GET_VM_PAGE allocates blocks of contiguous (512-byte) pages in the program region. The LIB$GET_VM_PAGE routine is designed for memory allocation request sizes ranging from one page to a few hundred pages. If not enough contiguous free pages are available to satisfy a request, the system calls the kernel procedure KER$ALLOCATE_MEMORY (instead of the VMS system service SYS$EXPREG). If you need to allocate over 1000 pages in a single request, you should call KER$ALLOCATE_MEMORY instead of LIB$GET_VM_PAGE.

Call Format	Type	Access	Mechanism
LIB$GET_VM_PAGE			
number-of-pages	Longword integer (signed)	Read only	By reference
,base-address	Longword (unsigned)	Write only	By reference

The *number-of-pages* argument specifies the number of contiguous pages to be allocated. The *base-address* argument receives the address of the first byte of the allocated block of pages.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$INT_OVER — Integer Overflow Detection

The LIB$INT_OVER routine enables or disables integer overflow detection for the calling routine activation and returns the previous integer overflow enable setting.

Call Format	Type	Access	Mechanism
LIB$INT_OVER			
new-setting	Longword (unsigned)	Read only	By reference

The *new-setting* argument specifies the new integer overflow enable setting.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$LEN — Length of String Returned

The LIB$LEN routine returns the length of a string as a longword value.

Call Format	Type	Access	Mechanism
LIB$LEN			
source-string	Character string	Read only	By descriptor

The *source-string* argument specifies the source string whose length the routine is to return.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$MATCH_COND — Match Condition Values

The LIB$MATCH_COND routine checks to see if a given condition value matches a list of condition values that you supply.

Call Format	Type	Access	Mechanism
LIB$MATCH_COND			
match-condition-value	Longword (unsigned)	Read only	By reference
,compare-condition-value, . . .	Longword (unsigned)	Read only	By reference

The *match-condition-value* argument specifies the condition value to be matched. The *compare-condition-value* argument specifies the condition values to be compared to *match-condition-value*.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$MULT_DELTA_TIME — Multiply Delta Time by Scalar

The LIB$MULT_DELTA_TIME routine multiplies a delta time by a longword integer scalar.

Call Format	Type	Access	Mechanism
LIB$MULT_DELTA_TIME			
multiplier	Longword (signed)	Read only	By reference
,delta-time	Quadword (unsigned)	Modify	By reference

The *multiplier* argument specifies the value by which the routine is to multiply the delta time. The *delta-time* argument specifies the delta time to be multiplied.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$MULTF_DELTA_TIME — Multiply Delta Time by an F_Floating Scalar

The LIB$MULTF_DELTA_TIME routine multiplies a delta time by an F_floating scalar.

Call Format	Type	Access	Mechanism
LIB$MULTF_DELTA_TIME			
multiplier	F_floating	Read only	By reference
,delta-time	Quadword (unsigned)	Modify	By reference

The *multiplier* argument specifies the value by which the routine is to multiply the delta time. The *delta-time* argument specifies the delta time to be multiplied.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$PUT_OUTPUT — Put Line to SYS$OUTPUT

The LIB$PUT_OUTPUT routine writes a record to the current controlling output device, specified by SYS$OUTPUT.

Call Format	Type	Access	Mechanism
LIB$PUT_OUTPUT			
message-string	Character string	Read only	By descriptor

The *message-string* specifies the message string that the routine is to write to the current controlling output device.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$RESET_VM_ZONE — Reset Virtual Memory Zone

The LIB$RESET_VM_ZONE routine frees all blocks of memory that previously were allocated from the zone.

Call Format	Type	Access	Mechanism
LIB$RESET_VMS_ZONE			
zone-id	Longword (unsigned)	Read only	By reference

The *zone-id* argument specifies the identifier of a zone created by a previous call to LIB$CREATE_VM_ZONE or LIB$CREATE_USER_VM_ZONE.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$SCOPY_DXDX — Copy Source String Passed by Descriptor to Destination

The LIB$SCOPY_DXDX routine copies a source string passed by descriptor to a destination string.

Call Format	Type	Access	Mechanism
LIB$SCOPY_DXDX			
source-string	Character string	Read only	By descriptor
,destination-string	Character string	Write only	By descriptor

The *source-string* argument specifies the source string to be copied to the destination string and the *destination-string* argument specifies the destination string to which the source string is to be copied.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$SCOPY_R_DX — Copy Source String Passed by Reference to Destination

The LIB$SCOPY_R_DX routine copies a source string passed by reference to a destination string.

Call Format	Type	Access	Mechanism
LIB$SCOPY_R_DX			
word-integer-source-length	Word (unsigned)	Read only	By reference
,source-string-address	Character string	Read only	By reference
,destination-string	Character string	Read only	By descriptor

The *word-integer-source-length* argument specifies the length of the source string. The *source-string-address* and *destination-string* arguments specify the source string to be copied to the destination string and the destination string to which the source string is copied, respectively.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$SIGNAL — Signal Exception Condition

The LIB$SIGNAL routine generates a signal that indicates that an exception condition has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program will exit.

Call Format	Type	Access	Mechanism
LIB$SIGNAL			
condition-value1	Longword (unsigned)	Read only	By value
[*,number-of-arguments1*]	Longword integer (signed)	Read only	By value
[*,FAO-argument1* . . .]	Unspecified	Read only	By value
[*,condition-value2*]	Longword (unsigned)	Read only	By value
[*,number-of-arguments2*]	Longword integer (signed)	Read only	By value
[*,FAO-argument2* . . .]	Unspecified	Read only	By value

The *condition-value1* and *condition-value2* arguments specify VAX 32-bit condition values. The optional arguments *number-of-arguments1* and *number-of-arguments2* specify the number of formatted ASCII output (FAO) arguments associated with the first and second condition values. The optional arguments *FAO-argument1* and *FAO-argument-2* specify optional FAO arguments associated with the first and second condition value.

Returns: None

LIB$SIG_TO_RET — Signal Converted to a Return Status

The LIB$SIG_TO_RET routine converts a signaled condition value to a value returned as a function. The signaled condition is returned to the caller of the user routine that established the handler that is calling LIB$SIG_TO_RET. This routine may be established as or called from a condition handler.

Call Format	Type	Access	Mechanism
LIB$SIG_TO_RET			
signal-arguments	Unspecified	Read only	By reference (array)
,mechanism-arguments	Unspecified	Read only	By reference (array)

The *signal-arguments* and *mechanism_arguments* arguments specify the addresses of arrays that are the signal argument and mechanism argument vector stacks, respectively.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$STOP — Stop Execution and Signal the Condition

The LIB$STOP routine generates a signal that indicates that an exception condition has occurred in your program. Exception conditions signaled by LIB$STOP cannot be continued from the point of the signal.

Call Format	Type	Access	Mechanism
LIB$STOP			
condition-value1	Longword (unsigned)	Read only	By value
[*,number-of-arguments1*]	Longword integer (signed)	Read only	By value
[*,FAO-argument1* . . .]	Unspecified	Read only	Unspecified
[*,condition-value2*]	Longword (unsigned)	Read only	By value
[*,number-of-arguments2*]	Longword integer (signed)	Read only	By value
[*,FAO-argument2* . . .]	Unspecified	Read only	Unspecified

The *condition-value1* and *condition-value2* arguments specify VAX 32-bit condition values. The optional arguments *number-of-arguments1* and *number-of-arguments2* specify the number of formatted ASCII output (FAO) arguments associated with the first and second condition values. The optional arguments *FAO-argument1* and *FAO-argument-2* specify optional FAO arguments associated with the first and second condition value.

Returns: None

LIB$SUB_TIMES — Subtract Two Quadword Times

The LIB$SUB_TIMES routine subtracts two time values in internal-time format.

Call Format	Type	Access	Mechanism
LIB$SUB_TIMES			
time1	Quadword (unsigned)	Read only	By reference
,time2	Quadword (unsigned)	Read only	By reference
,resultant-time	Quadword (unsigned)	Write only	By reference

The *time1* argument specifies the time from which the routine subtracts another time. The *time2* argument specifies the time that the routine subtracts from the first time. The *resultant-time* argument receives the difference.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

LIB$SUBX — Multiple-Precision Binary Subtraction

The LIB$SUBX routine performs subtraction on signed two's complement integers of arbitrary length.

Call Format	Type	Access	Mechanism
LIB$SUBX			
minuend-array	Unspecified	Read only	By reference (array)
,subtrahend-array	Unspecified	Read only	By reference (array)
,difference-array	Unspecified	Write only	By reference (array)
[*,array-length*]	Longword integer (signed)	Read only	By reference

The *minuend-array* and *subtrahend-array* specify the minuend and subtrahend, multiple-precision, signed two's complement integers. The *difference-array* argument receives the difference, a multiple-precision, signed two's complement integer. The opitional argument *array-length* specifies the length of the arrays on which the routine is to operate.

Returns:

Type: Longword (unsigned)
Access: Write only
Mechanism: By value

C.5 STR$ Emulation Routine Description

This section briefly describes the STR$ Runtime Library routine STR$ANALYZE_SDESC. For details, see the *VMS RTL String Manipulation (STR$) Manual.*

STR$ANALYZE_SDESC — Analyze String Descriptor

The STR$ANALYZE_SDESC routine extracts the length and starting address of the data for a variety of string descriptor classes.

Call Format	Type	Access	Mechanism
STR$ANALYZE_SDESC			
input-descriptor	Character string	Read only	By descriptor
,word-integer-length	Word (unsigned)	Write only	By reference
,data-address	Longword (unsigned)	Write only	By reference

The *input-descriptor* argument specifies the input descriptor from which the routine is to extract the length of the data and the address at which the data starts. The *word-integer-length* and *data-address* arguments specify the length and address of the data. The routine extracts the length and address from the descriptor.

Returns: None

Appendix D

SCSI Port Driver Interface Routines

The VAXELN Toolkit provides a set of interface routines for programming communication between user-written SCSI class drivers and the VAXELN SCSI port driver. The port driver interface routines let you allocate and build a SCSI command request packet and send it to a device on a SCSI bus. The interface also provides routines for freeing resources after a SCSI command performs its operations.

This appendix provides descriptions of the SCSI port driver interface routines. Each description provides an overview; call formats and argument information for Pascal, C, and FORTRAN; argument descriptions; a description of the routine's return value; and examples.

The call formats for the Pascal interface show how to invoke the routines using the INVOKE function. A call to INVOKE specifies a pointer to the routine's entry mask, the name of a function type declared for the routine, and the routine's arguments. Descriptions are provided for only the port interface routine arguments; the argument descriptions listed for each routine do not include the INVOKE function's entry mask pointer or function type arguments.

Before a class driver written in Pascal can invoke the interface routines, the driver must declare the routine addresses as follows:

```
TYPE
  PORT_ROUTINES = RECORD
                    ctx    : ^ANYTYPE;
                    init   : ^ANYTYPE;
                    issue  : ^ANYTYPE;
                    alloc  : ^ANYTYPE;
                    free   : ^ANYTYPE;
                    map    : ^ANYTYPE;
                    unmap  : ^ANYTYPE;
                    exit   : ^ANYTYPE;
                  END;
```

```
VAR
  routine_addresses : [EXTERNAL] port_routines;
```

Similarly, before a class driver written in FORTRAN can invoke the interface routines, the driver must declare the type definition for the routine addresses defined in $SCSIPORT.H as follows:

```
STRUCTURE /PORT_ROUTINES/
  INTEGER*4  ctx
  INTEGER*4  init
  INTEGER*4  issue
  INTEGER*4  alloc
  INTEGER*4  free
  INTEGER*4  map
  INTEGER*4  unmap
  INTEGER*4  exit
END STRUCTURE

RECORD /PORT_ROUTINES/ routine_addresses
```

Prior to the calling the routines, the FORTRAN class driver must also delcare the variables *routine_addresses* and *lock_device* as external data, using the EXTERNAL statement as follows:

```
EXTERNAL routine_addresses
EXTERNAL lock_device
```

These statements ensure that the symbols are resolved such that they are the addresses for the SCSI port interface callback routines as declared by the VAX C global definition (globaldef) storage class. For more information, see Section 14.5.3.2.

A SCSI class driver gains access to the interface routines by using their addresses. One way of doing this from a FORTRAN driver is to apply the usual method for dealing with pointers in FORTRAN. For example:

- Pass the external variable *routine_addresses* to a subroutine that declares the variable as a RECORD /PORT_ROUTINES/. This enables the driver to access the necessary fields of the *routine_addresses*.
- As appropriate, pass a routine address (for example, *routine_addresses.alloc*) by value to another subroutine that redeclares the address to be EXTERNAL. The driver can then call the routine by using the name of the dummy argument.

For information about developing a user-written class driver, see Section 14.5.3. See SAMPLE_SCSIDRIVER.PAS, SAMPLE_SCSIDRIVER.C, and SAMPLE_SCSIDRIVER.FOR in the VAXELN ELN$ directory for sample user-written SCSI class drivers.

PORT$ALLOCATE_DEVICE

Allocates a virtual device (SCSI command request packet) for the calling SCSI class driver and returns the virtual device number.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following function type:

```
FUNCTION scsiport$allocate_device(ctx : ^ANYTYPE;
                                  scsi_target : INTEGER;
                                  cmd_bcnt : INTEGER;
                                  cmd_ptr : ^ANYTYPE;
                                  sts_ptr : ^ANYTYPE): INTEGER;
FUNCTION_TYPE;
```

Pascal Format

INCLUDE SCSIPORT

INCLUDE $SCSI_UTILITY

virtual_device := INVOKE

(routine_addresses.alloc,
scsiport$allocate_device,
routine_addresses.ctx,
scsi_target,
cmd_bcnt,
cmd_ptr,
sts_ptr)

argument information

routine_addresses.ctx : ^ANYTYPE
scsi_target : INTEGER
cmd_bcnt : INTEGER
cmd_ptr : ^ANYTYPE
sts_ptr : ^ANYTYPE

PORT$ALLOCATE_DEVICE

C Format **#include $scsiport**

#include $scsi_utility

virtual_device = (*routine_addresses.ctx_a_alloc)

(routine_addresses.ctx,
scsi_target,
cmd_bcnt,
cmd_ptr,
sts_ptr)

argument information

char **routine_addresses.ctx*
char *scsi_target*
int *cmd_bcnt*
unsigned char ***cmd_ptr*
unsigned char ***sts_ptr*

FORTRAN Format **INCLUDE 'ELN$:FORTRAN_DEFS.FOR'**

virtual_device = alloc_routine

(%val(routine_addresses.ctx),
%val(scsi_target),
%val(cmd_bcnt),
cmd_ptr,
sts_ptr)

argument information

INTEGER*4 *routine_addresses.ctx*
INTEGER*4 *scsi_target*
INTEGER*4 *cmd_bcnt*
INTEGER*4 *cmd_ptr*
INTEGER*4 *sts_ptr*

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

scsi_target
Specifies the SCSI device ID for the device on the SCSI bus that is to handle the command request.

cmd_bcnt
Specifies the number of bytes to be allocated for the request packet's SCSI command buffer. The command buffer can store up to 256 bytes of command data.

cmd_ptr
Receives a pointer to the request packet's command buffer. A driver must use the returned pointer to place a SCSI command in the request packet.

sts_ptr
Receives a pointer to the request packet's SCSI status buffer. The status buffer receives a status code from the target device, as defined in the ANSI SCSI specification, after the completion of a SCSI command.

Returns

An integer in the range 0 to 15 that identifies the SCSI command request packet.

Examples

1.
```
virtual_device := INVOKE(routine_addresses.alloc,
                         scsiport$allocate_device,
                         scsi_target,
                         cmd_bcnt,
                         ADDRESS(cmd_ptr),
                         ADDRESS(sts_ptr));
```

Shows a call to PORT$ALLOCATE_DEVICE in VAXELN Pascal.

2.
```
virtual_device = (*routine_addresses.ctx_a_alloc)
                        (routine_addresses.ctx_a_context,
                         scsi_target,
                         cmd_bcnt,
                         &cmd_ptr,
                         &sts_ptr);
```

Shows a call to PORT$ALLOCATE_DEVICE in C.

3.
```
call_allloc = alloc_routine(%val(routine_addresses.ctx),
                            %val(scsi_target),
                            %val(cmd_bcnt),
                            cmd_ptr,
                            sts_ptr)
```

Shows a call to PORT$ALLOCATE_DEVICE in FORTRAN. The *call_alloc* identifier is the name of the function that is to handle the address of the PORT$ALLOCATE_DEVICE callback routine.

PORT$EXIT_HANDLER

Terminates the SCSI port driver and returns all port driver resources back to the system.

NOTE

> An application should call this function only if the port driver needs to be terminated. This function frees all resources associated with the port and disconnects the device from the interrupt service routine (ISR). Digital recommends that user-defined class drivers not invoke this function.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following function type:

```
FUNCTION scsiport$exit_handler(ctx : ^ANYTYPE): INTEGER;
FUNCTION_TYPE;
```

Pascal Format

INCLUDE $SCSIPORT
INCLUDE $SCSI_UTILITY
status := INVOKE

(routine_addresses.exit,
scsiport$exit_handler,
routine_addresses.ctx)

argument information

routine_addresses.ctx : ^ANYTYPE

C Format

#include $scsiport
#include $scsi_utility
result = (*routine_addresses.ctx_a_exit)

(ctx)

PORT$EXIT_HANDLER

argument information

char **routine_addresses.ctx*

FORTRAN Format

INCLUDE 'ELN$:FORTRAN_DEFS.FOR'

result = exit_routine
(%val(routine_addresses.ctx))

argument information

INTEGER*4 *routine_addresses.ctx*

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

Returns

A status value returned by kernel routine calls.

Examples

1.
```
status := INVOKE(routine_addresses.exit,
                 scsiport$exit_handler,
                 routine_addresses.ctx);
```

Shows a call to PORT$EXIT_HANDLER in VAXELN Pascal.

2.
```
status = (*routine_addresses.ctx_a_exit)
             (routine_addresses.ctx_a_context)
```

Shows a call to PORT$EXIT_HANDLER in C.

3.
```
call_exit = exit_routine(%val(routine_addresses.ctx))
```

Shows a call to PORT$EXIT_HANDLER in FORTRAN. The *call_exit* identifier is the name of the function that is to handle the address of the PORT$EXIT_HANDLER callback routine.

PORT$FREE_DEVICE

Returns a SCSI command request packet to the list of free SCSI command request packets. If another process is waiting for a request packet, PORT$FREE_DEVICE signals that process.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following function type:

```
FUNCTION scsiport$deallocate_device(ctx : ^ANYTYPE;
                                    virtual_device : INTEGER) : INTEGER;
FUNCTION_TYPE;
```

Pascal Format

INCLUDE $SCSIPORT
INCLUDE $SCSI_UTILITY
status := INVOKE

(routine_addresses.free,
scsiport$deallocate_device,
routine_addresses.ctx,
virtual_device)

argument information

routine_addresses.ctx : ^ANYTYPE
virtual_device : INTEGER

C Format

#include $scsiport
#include $scsi_utility
status = (*routine_addresses.ctx_a_free)

(routine_addresses.ctx,
virtual_device)

argument information

char **routine_addresses.ctx*
int *virtual_device*

FORTRAN Format

INCLUDE 'ELN$:FORTRAN_DEFS.FOR'

status = free_routine

(%val(routine_addresses.ctx),
%val(virtual_device))

argument information

INTEGER*4 *routine_addresses.ctx*
INTEGER*4 *virtual_device*

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

virtual_device
Specifies the packet ID for the SCSI command request packet to be returned to the request packet free list. You must specify a packet ID returned by PORT$ALLOCATE_DEVICE.

Returns

An integer status value of SS$_NORMAL.

Examples

1.
```
status := INVOKE(routine_addresses.free,
                 scsiport$deallocate_device,
                 routine_addresses.ctx,
                 virtual_device)
```

Shows a call to PORT$FREE_DEVICE in VAXELN Pascal.

2.
```
status = (*routine_addresses.ctx_a_free)
             (routine_addresses.ctx_a_context,
              virtual_device);
```

   Shows a call to PORT$FREE_DEVICE in C.

3.
```
call_free = free_routine(%val(routine_addresses.ctx),
                         %val(virtual_device))
```

   Shows a call to PORT$FREE_DEVICE in FORTRAN. The *call_free* identifier is the name of the function that is to handle the address of the PORT$FREE_DEVICE callback routine.

PORT$INITIALIZE_CONTROLLER

Asserts the SCSI RST signal on the SCSI bus. This signal causes all devices on the SCSI bus to release all asserted signals and places the bus in a BUS FREE state.

NOTE

> A class driver should not call this routine unless the SCSI bus is hung.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following function type:

```
FUNCTION scsiport$initialize_controller(ctx : ^ANYTYPE;
            scsi_target : INTEGER): INTEGER;
FUNCTION_TYPE;
```

Pascal Format

INCLUDE $SCSIPORT
INCLUDE $SCSI_UTILITY
status := INVOKE

(routine_addresses.init,
scsiport$initialize_controller,
routine_addresses.ctx,
scsi_target)

argument information

routine_addresses.ctx : ^ANYTYPE
scsi_target : INTEGER

PORT$INITIALIZE_CONTROLLER

C Format **#include $scsiport**
#include $scsi_utility
status = (*routine_addresses.ctx_a_init)

(routine_addresses.ctx,
scsi_target)

---

argument information

char **routine_addresses.ctx*
char *scsi_target*

FORTRAN Format **INCLUDE 'ELN$:FORTRAN_DEFS.FOR'**
status = init_routine

(%val(routine_addresses.ctx),
%val(scsi_target))

---

argument information

INTEGER*4 *routine_addresses.ctx*
INTEGER*4 *scsi_target*

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

scsi_target
Specifies the SCSI device ID for a working SCSI target device.

Returns

An integer status value of SS$_NORMAL.

Examples

1.
```
status := INVOKE(routine_addresses.init,
                 scsiport$initialize_controller,
                 routine_addresses.ctx,
                 scsi_target);
```

Shows a call to **PORT$INITIALIZE_CONTROLLER** in VAXELN Pascal.

2.
```
status = (*routine_addresses.ctx_a_init)
            (routine_addresses.ctx_a_context,
             scsi_target)
```

Shows a call to PORT$INITIALIZE_CONTROLLER in C.

3.
```
call_init = init_routine(%val(routine_addresses.ctx),
                         %val(scsi_target))
```

Shows a call to PORT$INITIALIZE_CONTROLLER in FORTRAN. The *call_init* identifier is the name of the function that is to handle the address of the PORT$INITIALIZE_CONTROLLER callback routine.

PORT$ISSUE_COMMAND

Arbitrates and selects a device on the SCSI bus, issues the SCSI command that is in the specified SCSI command request packet, and performs the tasks necessary to complete the operation.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following constants and function type:

```
CONST
  SCSI$K_DISCONNECT = 0;
  SCSI$K_NODISCONNECT = 1;
  SCSI$K_RETRY = 0;
  SCSI$K_NORETRY = 1;

FUNCTION scsiport$issue_command(ctx : ^ANYTYPE;
                                virtual_device : INTEGER;
                                disconnect : INTEGER;
                                disable_retry : INTEGER;
                                phase_timeout : INTEGER;
                                disconnect_timeout : INTEGER) : INTEGER;
FUNCTION_TYPE;
```

The constants SCSI$K_DISCONNECT, SCSI$K_NODISCONNECT, SCSI$K_RETRY, and SCSI$K_NORETRY are declared in the modules **$scsi_utility** and ELN$:FORTRAN_DEFS.FOR for drivers written in C and FORTRAN, respectively.

Pascal Format

INCLUDE $SCSIPORT

INCLUDE $SCSI_UTILITY

status := INVOKE

(routine_addresses.issue,
scsiport$issue_command,
routine_addresses.ctx,
virtual_device,
disconnect,
disable_retry,
phase_timeout,
disconnect_timeout)

argument information

routine_addresses.ctx : ^ANYTYPE
virtual_device : INTEGER
disconnect : INTEGER
disable_retry : INTEGER
phase_timeout : INTEGER
disconnect_timeout : INTEGER

C Format **#include $scsiport**

#include $scsi_utility

status = (*routine_addresses.ctx_a_issue)

(routine_addresses.ctx,
virtual_device,
disconnect,
disable_retry,
phase_timeout,
disconnect_timeout)

argument information

char **routine_addresses.ctx*
int *virtual_device*
int *disconnect*
int *disable_retry*
int *phase_timeout*
int *disconnect_timeout*

FORTRAN Format **INCLUDE 'ELN$:FORTRAN_DEFS.FOR'**

status = issue_routine

(%val(routine_addresses.ctx),
%val(virtual_device),
%val(disconnect),
%val(disable_retry),

%val(phase_timeout),
%val(disconnect_timeout))

argument information

INTEGER*4 *routine_addresses.ctx*
INTEGER*4 *virtual_device*
INTEGER*4 *disconnect*
INTEGER*4 *disable_retry*
INTEGER*4 *phase_timeout*
INTEGER*4 *disconnect_timeout*

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

virtual_device
Specifies the packet ID for the SCSI command request packet that contains the command being issued. You must specify a packet ID returned by PORT$ALLOCATE_DEVICE.

disconnect
Specifies whether the target device can disconnect during command execution. Specify SCSI$K_DISCONNECT to allow the device to disconnect. Specify SCSI$K_NODISCONNECT to prevent the device from disconnecting.

disable_retry
Specifies whether the port driver should attempt to repeat a command that fails due to a timeout, bus parity, or invalid phase transition error. Specify SCSI$K_RETRY to allow the port driver to retry commands. When this characteristic is set, the port driver will attempt three retries. Specify SCSI$K_NORETRY to disable retries.

phase_timeout
Specifies the amount of time a target device has to change to another SCSI bus phase. You can specify from 0 to 420 seconds. If you specify 0 seconds or an invalid value, the driver uses a timeout value of 20 seconds.

disconnect_timeout
Specifies the amount of time a target device has to reselect an initiator to proceed with a disconnected data transfer. You can specify from 0 to 420 seconds. If you specify 0 seconds or an invalid value, the driver uses a timeout value of 20 seconds.

Returns

The status value SS$_CTRLERR, SS$_TIMEOUT, or SS$_NORMAL.

Examples

1.
```
status := INVOKE(routine_addresses.issue,
                 scsiport$issue_command,
                 routine_addresses.ctx,
                 virtual_device,
                 SCSI$K_DISCONNECT,
                 SCSI$K_RETRY,
                 phase_timeout,
                 disconnect_timeout);
```

Shows a call to PORT$ISSUE_COMMAND in VAXELN Pascal.

2.
```
status = (*routine_addresses.ctx_a_issue)
            (routine_addresses.ctx_a_context,
             virtual_device,
             SCSI$K_DISCONNECT,
             SCSI$K_RETRY,
             phase_timeout,
             disconnect_timeout);
```

Shows a call to PORT$ISSUE_COMMAND in C.

3.
```
call_issue = issue_routine(%val(routine_addresses.ctx),
                           %val(virtual_device),
                           %val(SCSI$K_DISCONNECT),
                           %val(SCSI$K_RETRY),
                           %val(phase_timeout),
                           %val(disconnect_timeout))
```

Shows a call to PORT$ISSUE_COMMAND in FORTRAN. The *call_issue* identifier is the name of the function that is to handle the address of the PORT$ISSUE_COMMAND callback routine.

PORT$MAP_BUFFER

Searches the 128-Kbyte SCSI DMA RAM bitmap for a specified amount of contiguous data bytes, updates the SCSI command request packet with the appropriate mapping information, and marks the bitmap pages as unavailable.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following constants and function type:

```
CONST
  SCSI$K_WRITE = 0;
  SCSI$K_READ = 1;

FUNCTION scsiport$map_buffer(ctx : ^ANYTYPE;
                             virtual_device : INTEGER;
                             buffer : ^ANYTYPE;
                             buf_len : INTEGER;
                             pad_len : INTEGER;
                             direction : INTEGER) : INTEGER;
FUNCTION_TYPE;
```

The constants SCSI$K_READ and SCSI$K_WRITE are declared in the modules **$scsi_utility** and ELN$:FORTRAN_DEFS.FOR for drivers written in C and FORTRAN, respectively.

Pascal Format

INCLUDE $SCSIPORT
INCLUDE $SCSI_UTILITY
status := INVOKE

(routine_addresses.map,
scsiport$map_buffer,
routine_addresses.ctx,
virtual_device,
buffer,
buf_len,
pad_len,
direction)

argument information

routine_addresses.ctx : ^ANYTYPE
virtual_device : INTEGER
buffer : ^ANYTYPE
buf_len : INTEGER
pad_len : INTEGER
direction : INTEGER

C Format **#include $scsiport**
#include $scsi_utility
status = (*routine_addresses.ctx_a_map)

(routine_addresses.ctx,
virtual_device,
**buffer,*
buf_len,
pad_len,
direction)

argument information

char **routine_addresses.ctx,*
int *virtual_device*
unsigned char **buffer*
int *buf_len*
int *pad_len*
int *direction*

FORTRAN Format **INCLUDE 'ELN$:FORTRAN_DEFS.FOR'**
status = map_routine

(%val(routine_addresses.ctx),
%val(virtual_device),
%ref(buffer),
%val(buf_len),

```
            %val(pad_len),
            %val(direction))
```

argument information

```
INTEGER*4 routine_addresses.ctx
INTEGER*4 virtual_device,
CHARACTER*n buffer
INTEGER*4 buf_len
INTEGER*4 pad_len
INTEGER*4 direction
```

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

virtual_device
Specifies the packet ID for the SCSI command request packet for which a data buffer is to be mapped. You must specify a packet ID returned by PORT$ALLOCATE_DEVICE.

buffer
Specifies a pointer to the data buffer to be mapped.

buf_len
Specifies the size of the data buffer to be mapped. Specify a value in the range 1 to 65536.

pad_len
Specifies the pad size needed for a SCSI command that requires a transfer size that is larger than the size specified by *buf_len*. If the amount of data requested in a SCSI command exceeds the space allocated for the data buffer, the pad size accounts for the difference.

direction
Specifies whether the data transfer is a read or write operation. Specify SCSI$K_WRITE for a write operation; specify SCSI$K_READ for a read operation.

Returns

An integer status value of SS$_NORMAL.

Examples

1.
```
status := INVOKE(routine_addresses.map,
                 scsiport$map_buffer,
                 routine_addresses.ctx,
                 virtual_device,
                 buffer,
                 buf_len,
                 pad_len,
                 SCSI$K_READ);
```

Shows a call to PORT$MAP_BUFFER in VAXELN Pascal.

2.
```
status = (*routine_addresses.ctx_a_map)
            (routine_addresses.ctx_a_context,
             virtual_device,
             *buffer,
             buf_len,
             pad_len,
             SCSI$K_READ);
```

Shows a call to PORT$MAP_BUFFER in C.

3.
```
call_map = map_routine(%val(routine_addresses.ctx),
                       %val(virtual_device),
                       %ref(buffer),
                       %val(buf_len),
                       %val(pad_len),
                       %val(SCSI$K_READ))
```

Shows a call to PORT$MAP_BUFFER in FORTRAN. The *call_map* identifier is the name of the function that is to handle the address of the PORT$MAP_BUFFER callback routine.

PORT$UNMAP_BUFFER

Returns the memory used for a SCSI command request packet data buffer back to the 128-Kbyte DMA RAM bitmap and marks the returned pages as available. If another process is waiting for DMA RAM memory, PORT$UNMAP_BUFFER signals that process.

To invoke this routine from a class driver written in Pascal, the driver must first declare the following function type:

```
FUNCTION scsiport$unmap_buffer(ctx : ^ANYTYPE;
                               virtual_device : INTEGER;
                               buffer : ^ANYTYPE;
                               buf_len : INTEGER;
                               pad_len : INTEGER): INTEGER;
FUNCTION_TYPE;
```

Pascal Format

INCLUDE $SCSIPORT
INCLUDE $SCSI_UTILITY
status := INVOKE

(routine_addresses.unmap,
scsiport$unmap_buffer,
routine_addresses.ctx,
virtual_device,
buffer,
buf_len,
pad_len)

argument information

routine_addresses.ctx : ^ANYTYPE
virtual_device : INTEGER
buffer : ^ANYTYPE
buf_len : INTEGER
pad_len : INTEGER

C Format **#include $scsiport**

#include $scsi_utility

status = (*routine_addresses.ctx_a_unmap)

(routine_addresses.ctx,
virtual_device,
**buffer,*
buf_len,
pad_len)

argument information

char **routine_addresses.ctx,*
int *virtual_device*
unsigned char **buffer*
int *buf_len*
int *pad_len*

FORTRAN Format **INCLUDE 'ELN$:FORTRAN_DEFS.FOR'**

status = unmap_routine

(%val(routine_addresses.ctx),
%val(virtual_device),
%ref(buffer),
%val(buf_len),
%val(pad_len))

argument information

INTEGER*4 *routine_addresses.ctx*
INTEGER*4 *virtual_device,*
CHARACTER**n buffer*
INTEGER*4 *buf_len*
INTEGER*4 *pad_len*

PORT$UNMAP_BUFFER

Arguments

routine_addresses.ctx
Specifies the pointer to the port driver's data structures.

virtual_device
Specifies the packet ID for the SCSI command request packet for which a data buffer is to be unmapped. You must specify a packet ID returned by PORT$ALLOCATE_DEVICE.

buffer
Specifies a pointer to the data buffer to be unmapped.

buf_len
Specifies the size of the data buffer to be unmapped. Specify a value in the range 1 to 65536.

pad_len
Specifies the pad size of the data buffer to be unmapped.

Returns

An integer status value of SS$_NORMAL.

Examples

1.
```
status := INVOKE(routine_addresses.unmap,
                 scsiport$unmap_buffer,
                 routine_addresses.ctx,
                 virtual_device,
                 buffer,
                 buf_len,
                 pad_len);
```

Shows a call to PORT$UNMAP_BUFFER in VAXELN Pascal.

2.
```
status = (*routine_addresses.ctx_a_unmap)
             (routine_addresses.ctx_a_context,
              virtual_device,
              *buffer,
              buf_len,
              pad_len);
```

Shows a call to PORT$UNMAP_BUFFER in C.

3.
```
call_unmap = unmap_routine(%val(routine_addresses.ctx),
                           %val(virtual_device),
                           %ref(buffer),
                           %val(buf_len),
                           %val(pad_len))
```

Shows a call to PORT$UNMAP_BUFFER in FORTRAN. The *call_unmap* identifier is the name of the function that is to handle the address of the PORT$UNMAP_BUFFER callback routine.

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua NH 03061
Rest of U.S.A. and Puerto Rico[1]	800–DIGITAL	

[1]Prepaid orders from Puerto Rico, call Digital's local subsidiary (809–754–7575)

From	Call	Write
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order Desk
Internal orders (for software documentation)	DTN: 241–3023 508–874–3023	Software Supply Business (SSB) Digital Equipment Corporation Westminster MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Services (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro MA 01532

Reader's Comments

VAXELN Runtime Facilities Guide
AA–JM81E–TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as described)	☐	☐	☐	☐
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)	☐	☐	☐	☐

What I like best about this manual: ______________________________

____

What I like least about this manual: ______________________________

____

My additional comments or suggestions for improving this manual:

____

____

I found the following errors in this manual:

Page Description

_____ ______________________________

_____ ______________________________

_____ ______________________________

Please indicate the type of user/reader that you most nearly represent:

☐ Administrative Support	☐ Scientist/Engineer
☐ Computer Operator	☐ Software Support
☐ Educator/Trainer	☐ System Manager
☐ Programmer/Analyst	☐ Other (please specify) __________
☐ Sales	

Name/Title ______________________ Dept. __________

Company ______________________ Date __________

Mailing Address ______________________________

______________________ Phone __________

10/87

— Do Not Tear — Fold Here and Tape —

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
PKO3-1/D30
129 PARKER STREET
MAYNARD, MA 01754-2198

— Do Not Tear — Fold Here —

Cut Along Dotted Line