First Edition - March 1985

This manual is an introduction and user's guide to the VAXELN toolkit.

VAXELN User's Guide

Document Order Number: **AA-EU38A-TE**

Software Version: 2.0

digital equipment corporation
maynard, massachusetts

First Edition, March 1985

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

Contents

List of Figures

List of Tables

Preface

The *VAXELN User's Guide* is an introduction to VAXELN concepts and a guide for program and system development using the VAXELN toolkit.

Manual Objectives

This manual contains programming language-independent concepts and features that are common to VAXELN systems, including system services, program development tools, and system development tools.

Intended Audience

This manual is designed for programmers and students who have a working knowledge of Pascal or the C programming language. A cursory understanding of the VAX/VMS DCL command language is also necessary.

Structure of this Document

This manual consists of 15 chapters and 4 appendices, organized as follows:

- Chapter 1, "VAXELN Concepts," answers general questions about what VAXELN is and how it is used to design and develop an application. The chapter also contains a walk-through example of building, down-line loading, and executing a simple VAXELN application.

- Chapter 2, "Kernel Objects," describes the types of VAXELN kernel objects: PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, and DEVICE.
- Chapter 3, "Processes and Jobs," discusses jobs and processes and the kernel services that affect the state of PROCESS objects, as well as memory management and memory allocation procedures, interjob data sharing, and the kernel services that affect the state of AREA objects.
- Chapter 4, "Synchronization," discusses synchronization in terms of the WAIT procedures, events, semaphores, and time representation, including the kernel services that affect the state of EVENT and SEMAPHORE objects.
- Chapter 5, "Interjob Communication," discusses messages and ports, message transmission, datagrams and circuits, and the kernel services that affect the state of MESSAGE, PORT, and NAME objects.
- Chapter 6, "I/O Devices and Interrupt Handling," discusses device interrupts, interrupt priority levels and procedures to manipulate them, recovery from power failure, the kernel services that affect the state of DEVICE objects, direct memory access UNIBUS and QBUS device handling procedures, and device register procedures.
- Chapter 7, "The Network Service," describes the VAXELN Network Service in functional terms, including network applications, application message services, name servers, node names and numbers, network management, and the facilities for communication with non-VAXELN nodes.

- Chapter 8, "System Security," discusses the VAXELN security features and how they can be used to protect resources and data.
- Chapter 9, "The File Service," discusses device specifications, volume names, file specifications, the File Access Listener, using file service volumes from VMS, file service operations, file utilities, disk and tape utilities, the interface with disk and tape drivers, and the Data Access Protocol.
- Chapter 10, "Device Drivers," discusses the features of VAXELN device drivers, including disk drivers, a tape driver, printer drivers, terminal drivers, and real-time device drivers.
- Chapter 11, "Exception Handling," discusses VAXELN exceptions and exception handling procedures, as well as status codes and the message processing features that handle the conversion of status codes into message text.
- Chapter 12, "Program Development," summarizes the use of the VAXELN Pascal and VAX C compilers and the VAX/VMS librarian and linker to prepare programs for inclusion in a VAXELN system.
- Chapter 13, "System Development," describes the VAXELN System Builder, including the EBUILD command, the System Builder menus, and the procedures for including images supplied by DIGITAL (drivers and services).
- Chapter 14, "Booting and Down-Line Loading," describes the procedure for booting the finished VAXELN system image on a target machine, as well as the procedure and preparatory steps for using the Ethernet (instead of portable disks or other media) to load systems onto target machines.

- Chapter 15, "Debugging," discusses the local and remote debugging methods provided with the VAXELN development system, including the general concepts for using the debuggers, the debugger syntax rules, and the VAXELN debugger commands.
- Appendix A, "VAX–11/750 Microcode Patch," explains the microcode control store patching procedure required on system power-up before running VAXELN on a VAX–11/750.
- Appendix B, "Kernel Procedures," summarizes the internal call notation of the procedures performed by the VAXELN kernel.
- Appendix C, "Status Values/Exception Names," lists the status values/exception names defined within VAXELN, including the source(s) and description of each exception.
- Appendix D, "VAXELN Performance Evaluation," evaluates the process synchronization and management, message passing, and file I/O performance for a VAXELN MicroVAX system.

Associated Documents

The following documents are relevant to VAXELN:

- *VAXELN Release Notes (AA-Z454C-TE)*
- *VAXELN Installation Manual (AA-EU37A-TE)*
- *VAXELN Pascal Language Reference Manual (AA-EU39A-TE)*
- *VAXELN C Run-Time Library Reference Manual (AA-EU40A-TE)*
- *VAXELN Application Design Guide (AA-EU41A-TE)*

- *VAX/VMS DCL Dictionary (AA-Z200A-TE)*
- *VAX/VMS I/O User's Reference Manual: Part 1 (AA-Z600A-TE)*
- *VAX/VMS I/O User's Reference Manual: Part 2 (AA-Z601A-TE)*
- *VAX Architecture Handbook (EB-19580-20)*
- *VAX Hardware Handbook 1982-1983 (EB-21812-20)*
- *DECnet DIGITAL Network Architecture General Description (AA-N149A-TC)*
- *DECnet–VAX System Manager's Guide (AA-H803C-TE)*
- *DECnet–VAX User's Guide (AA-H802B-TE)*
- *MicroVAX I Owner's Manual (EK-KD32A-OM)*
- *LSI–11 Analog System User's Guide (EK-AXV11-UG)*
- *DLV11–J User's Guide (EK-DLV1J-UG)*

Chapter 1
VAXELN Concepts

This chapter answers general questions about what VAXELN is and how it is used to design and develop an application. The last section of the chapter shows you how to actually build and down-line load a simple VAXELN application.

What Is VAXELN?

VAXELN is a software product for the development of dedicated, real-time systems for VAX processors. The development tools run on the VAX/VMS and MicroVMS operating systems.

For the purposes of this discussion, a *dedicated application* is one in which the computers are used to solve a specific problem or, possibly, a set of related problems. The term spans a wide range of applications, from "workstations" designed for a particular profession to automated industrial machinery and robots.

In the design of VAXELN, a *real-time application* is any in which the system's response to external events is critical. Such applications include the typical scientific and industrial data processing situations in which the computer's operation has to be precisely synchronized with machines and special input/output devices.

Traditionally, the design and development of such applications has required expert programmers. The control of external devices usually requires a programmer intimately familiar with the target computer. The precise timing and hardware usage requirements of these applications usually require features not provided in high-level languages, meaning that much of the device-specific code has traditionally been written in assembly language.

Many such applications are best implemented with sets of concurrently executing processes, which have not traditionally been supported by high-level languages. The execution of concurrent processes, along with other constraints, usually requires a host operating system as part of the application, to manage and schedule the processes.

In other words, a traditional application programmer in this area had to be an expert in several fields beside his own profession: the design of real-time systems, programming in assembly language, programming in high-level languages, operating systems, and computer architecture.

The aim of VAXELN is to change this picture, relying mostly on your expertise in your own field rather than on experience with sophisticated programming. It gives you the following capabilities:

- High-level programming languages
- No operating system "overhead"
- Concurrent programming
- Transparent network support
- File Service
- Program development and debugging with VAX/VMS

These capabilities are summarized in the following subsections.

High-Level Programming Languages

You can develop VAXELN systems entirely in a high-level language, including handling of devices, exceptions, timeouts, and power failure. The recommended languages are VAXELN Pascal, which is a superset of ISO-standard Pascal, or the VAX C programming language.

No Operating System "Overhead"

VAXELN systems execute directly on the VAX processor hardware, without the need for a host operating system. Part of every system is a small *kernel* executive that manages the system's resources, processes, and data. Its general principles are described later in this chapter.

Concurrent Programming

VAXELN provides *multitasking* in Pascal or C programs; that is, writing a program made up of several concurrently executing parts. *Multiprogramming* is also supported; that is, you can construct a system from several concurrently executing programs.

A full discussion of concurrent programming is beyond the scope of this book. However, its most basic principle is that parts of a program (multitasking), or programs within a system (multiprogramming), are written as if each part had the potential to execute simultaneously, or in parallel, with any other part.

Even in cases where the programs or program parts actually share the same computer (and so do not actually execute in parallel), concurrent programming has numerous advantages in system design, including performance advantages, compared with simpler models in which every program runs to completion before any other can run.

Transparent Network Support

Data communications between VAXELN jobs are transparent across a network, and facilities are provided to make it easy to distribute an application's programs among several network nodes. Changing the network location of a program typically requires no changes to the program.

The VAXELN Network Service provides the communication services using the DIGITAL Network Architecture DECnet protocols. Since DECnet is supported by all of DIGITAL's operating systems, VAXELN applications are capable of communicating with programs running on processors anywhere in a DECnet network.

File Service

The VAXELN File Service supports I/O operations from VAXELN programs to local file storage devices, as well as remote file access to and from other network nodes. I/O requests from the user's programs are interpreted by the File Service and performed by the appropriate device driver program.

The File Service uses the same "on-disk" structure as VAX/VMS and the same internal format as the VAX Record Management System (RMS). Accordingly, volumes from one environment are readable and writable on the other.

Files are sequentially organized, but can be accessed either sequentially or randomly. Neither RMS indexed file organization nor indexed access is supported. Also, RMS relative file organization is not supported.

An alternative to VAXELN's sequential file system is provided by VAX Rdb/ELN, a high-performance relational database management system that runs on VAXELN target systems. Using VAXELN and Rdb/ELN, you can develop a database that is shared by multiple nodes in an Ethernet local area network (LAN).

VAXELN provides a separate tape File Service that uses the same tape file structure as VAX/VMS. This service provides users with a convenient means of transporting files to and from VAX/VMS systems.

Note that the File Service is not required for I/O to printers or terminals and is only present on VAXELN systems that have storage media.

Program Development and Debugging with VAX/VMS

VAX/VMS is used as a host to develop VAXELN systems through a small set of utility programs. Target systems can be debugged remotely from the VAX/VMS host system when they are connected to the VAX/VMS system with the Ethernet, or they can be debugged directly on the target hardware.

Chapter 15, "Debugging," contains a full discussion of debugging with the VAXELN debuggers.

What Is a VAXELN System?

A VAXELN system is a set of programs executing on VAX hardware, along with standard code and data that manage the programs' execution.

The hardware includes one or more VAX processors, optional peripheral devices including disks and terminals, and communication hardware to support the execution of the programs on various nodes in a LAN.

In addition, a VAXELN hardware configuration may include special hardware you have designed or acquired, such as custom device interfaces.

The programs executing in a VAXELN system are of two kinds:

- User programs. These can include user-written device drivers or resource services, as well as typical computational programs.
- Programs (*services* and *drivers*) supplied by DIGITAL. Examples are the File Service and the Network Service. Included in this set of programs are drivers for the standard supported peripheral devices and a dynamic program loader.

You develop a new VAXELN system by writing whatever new programs are required in VAXELN Pascal, VAX C, or other VAX languages. Then, with simple VAX/VMS commands, you combine the programs with each other, with any of the standard programs you want, and with the VAXELN kernel, to form an executable system. (The commands are described in Chapter 12, "Program Development," and Chapter 13, "System Development.") Any number of executable programs, called *program images*, can be combined with the kernel to form a VAXELN system.

If you are programming for a set of computers linked by a network, you simply prepare a system for each connected machine, or node. In VAXELN, the term *application* is used to mean the complete complex of systems in such cases. In defining names, such as for message ports, *local* names are defined for the system in which they are created (that is, the network node), and *universal* names are defined for the entire application (that is, for any of the systems present on any of the nodes).

Once a VAXELN system has been prepared, the *system image* is ready to be booted on a target machine. A VAXELN system can be booted from a disk, from a TU58 tape cartridge, from a diskette, by using the Ethernet to down-line load the system onto a target machine, or from Read Only Memory (ROM).

Chapter 14, "Booting and Down-Line Loading," contains more information on booting or down-line loading a VAXELN system image.

Structure of a Running VAXELN Appplication

After booting or down-line loading a VAXELN system image onto each target machine in your local area network, you have a completely defined VAXELN application. The typical structure of such a network-based application consists of:

- A VAX processor running the VAX/VMS or MicroVMS operating system that serves as the host development system. This processor is used to develop and build each VAXELN system and it contains the VAXELN debugger, which can remotely access one or more VAXELN target system nodes at the same time for debugging purposes.
- One or more target machines connected by the Ethernet to the VAX processor serving as the host development system and to each of the other target machines. Each target machine is a node in the network, and each contains its own running VAXELN system.

Each VAXELN system is a collection of jobs, each executing a program. Each job contains a master process and zero or more subprocesses, all executing in parallel. (Jobs, processes, and subprocesses are discussed in detail in Chapter 3, "Processes and Jobs.")

The File Service, Network Service, device drivers, and user programs are all independent jobs in an operating VAXELN application. If the processes in a system or network require a complex service, it is provided in an independent program running in the system or network.

For example, a *file server*, providing file storage hardware and software for the network, is built by constructing a VAXELN system from the Network Service, File Service, and disk driver and running the system on a network node that has the actual disks. VAXELN programs anywhere in the network can then use the file server without having to know its network location.

Dynamic Program Loading

If your VAXELN system contains the dynamic program loader (supplied to you as a prepared image), you can load additional program images into your currently running system. The program loader provides the following advantages:

- System sizes are smaller, since all of the program images that may ever be executed need not be loaded into the system.
- You can decide at run time which program images need to be included in the system and load only those images.
- You can use the program loader as a debugging tool, making it unnecessary to rebuild and reload a system each time a program is changed.

The VAXELN Kernel

The VAXELN kernel is the layer of software that lies between the raw hardware and your application software. It is delivered to you as a prepared image, ready to be built into a system along with your programs.

The function of the kernel is to provide for the controlled sharing of system resources and to synchronize communication among the various programs in the system. The kernel provides all the mechanisms necessary for a wide range of applications, in a simple, straightforward way. For example, it provides the basic mechanism to communicate between processes. It also maintains all information about the system data and about the user programs defined for a particular system.

The VAXELN kernel provides most of its services through a set of *objects* and procedures to manipulate them. These objects are data structures that the kernel uses to maintain the context of some ongoing service. (The VAXELN kernel objects are discussed in detail in Chapter 2, "Kernel Objects.") Since the kernel objects are critical to the operation of the system, both the objects and the procedures that manipulate them are housed within the protective boundary of VAX kernel mode.

Procedures that create, delete, or otherwise affect the state of any of the VAXELN kernel objects are referred to as *kernel services*. These services are described in general terms in this manual, in the chapters containing the topics to which they relate.

The language-specific call formats and detailed argument descriptions for each kernel procedure are contained in the *VAXELN Pascal Language Reference Manual* and the *VAXELN C Run-Time Library Reference Manual*, as appropriate to the programming language in use. In addition, Appendix B, "Kernel Procedures," shows the internal call notation for use in constructing calls from other languages.

Creating a VAXELN Application

So far, this chapter has described the major concepts behind VAXELN. To see the power and simplicity of the VAXELN toolkit for yourself, follow the procedures described in this section to take a modest Pascal or C program from the text editing stage to execution on a target machine.

What You Need

To create this VAXELN application, you need:

- A VAX or MicroVAX host computer with the VMS operating system, DECnet–VAX, and VAXELN installed on it
- An account on the host computer, with at least OPER and NETMBX privileges
- A MicroVAX with a DEQNA (DIGITAL Ethernet QBUS Network Adapter) for a target computer
- An Ethernet between the host computer and the target MicroVAX

The Steps Involved

To create this, or any, VAXELN application, you must perform the following six steps:

1. Edit the program.
2. Compile the program.

3. Link the program.
4. Build the system.
5. Configure the host processor for down-line loading.
6. Down-line load and run the program.

The Sample Application

There are two parts to the procedure you are about to perform.

In Part 1, you down-line load the VAXELN application from the host computer onto the target computer, and run the application on the target computer without the debugger. The application's output is displayed on the target computer's console terminal.

In Part 2, you down-line load the VAXELN application again, but this time you control the application from the host using the debugger. The application's output is displayed on the host's terminal.

Part 1

For Part 1, the object is to write a program in Pascal or in C, and use VAXELN to build a VAXELN system that can be down-line loaded to your target MicroVAX and executed.

Step 1: Edit the program. From a terminal attached to your VAX/VMS host, use any text editor to create a file containing the source code shown below. If you are using Pascal, name the file SAMPLE.PAS and enter the code shown in the left-hand example. If you are using C, name the file SAMPLE.C and enter the code shown in the right-hand example.

Pascal

```
Program sample;

BEGIN
   writeln ('Hello!');
END.
```

C

```
sample()
{
   printf("Hello!\n");
}
```

Step 2: Compile the program. Still at your host terminal, enter this command to compile the Pascal program:

```
$ EPASCAL/DEBUG  SAMPLE.PAS
```

or this command to compile the C program:

```
$ CC/DEBUG  SAMPLE.C + ELN$:VAXELNC/LIB
```

These commands invoke the appropriate language compiler to compile your source program and produce a file called SAMPLE.OBJ. Actually, you do not have to type the filename extensions (for example, the .PAS and .C), but they are included here for clarity.

The DEBUG qualifier is specified so that debug information is included in SAMPLE.OBJ. This allows you to debug your program using the source statements.

Step 3: Link the program. At your host terminal, invoke the VMS linker by entering this command to link the Pascal program:

```
$ LINK/DEBUG  SAMPLE.OBJ + ELN$:RTLSHARE/LIB + -
  RTL/LIB
```

or this command to link the C program:

```
$ LINK/DEBUG  SAMPLE.OBJ + -
  ELN$:CRTLSHARE/LIB + RTLSHARE/LIB + RTL/LIB
```

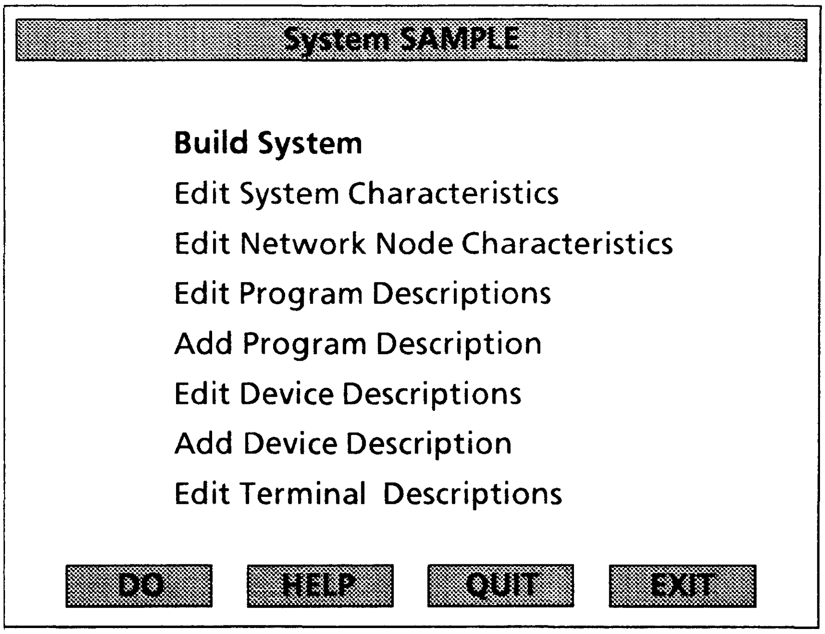
These are rather lengthy commands; at some point, you will probably want to create a command procedure for them so you do not have to type them each time.

The LINK command links the object program SAMPLE.OBJ with the VAXELN run-time library and kernel, and produces a file called SAMPLE.EXE. SAMPLE.EXE is an executable program, but it is not yet a VAXELN application. First, it must be built into a VAXELN system.

Step 4: Build the system. At your host terminal, next invoke the VAXELN System Builder by entering this command:

```
$ EBUILD  SAMPLE
```

Your screen looks like this:


System SAMPLE
Build System
Edit System Characteristics
Edit Network Node Characteristics
Edit Program Descriptions
Add Program Description
Edit Device Descriptions
Add Device Description
Edit Terminal Descriptions
DO
HELP
QUIT
EXIT

This is the main menu screen of the System Builder. The System Builder does two things for you:

1. It creates a file called SAMPLE.SYS that is a bootable system image containing your SAMPLE.EXE program, other programs such as device drivers and services, and appropriate subroutines from the run-time library and the kernel.
2. It creates a file called SAMPLE.DAT that is a data file containing text descriptions of what you edit on the System Builder menu screens. The System Builder uses this information to determine what to display in the menus next time you invoke the System Builder to build another version of your SAMPLE system.

Notice on your screen that the item *Build System* is highlighted, and the blinking cursor is located after it. The highlighting indicates that *Build System* is the item currently selected. On all System Builder menus, the choices that are currently selected are highlighted.

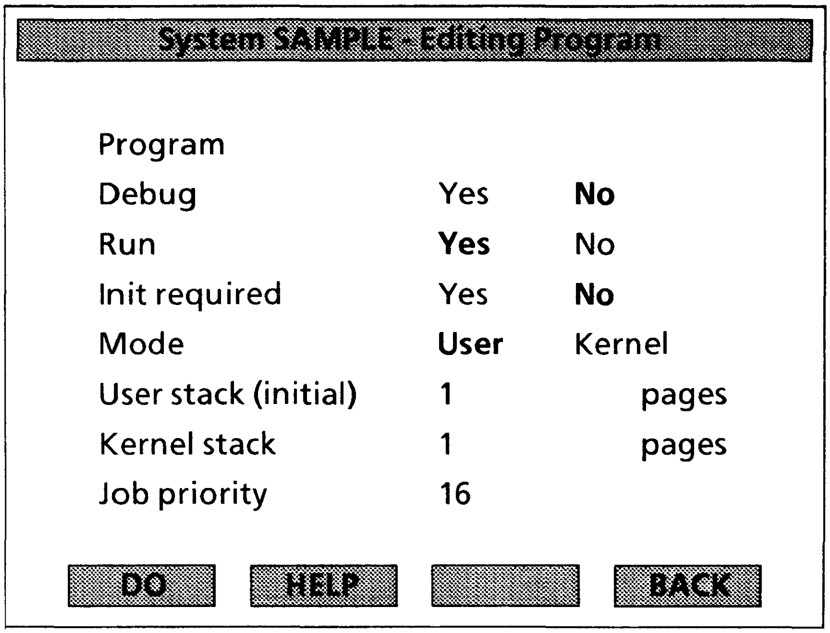
The first step in building a system is to describe it. Press the "down-arrow" cursor movement key (the key with the ↓ symbol on it) until the item *Add Program Description* is highlighted.

The boxes along the bottom of your screen with the words DO, HELP, QUIT, and EXIT, represent the PF1, PF2, PF3, and PF4 keys, respectively, on your keyboard.

Any time you are not sure what to do, you can press the PF2 key to get help with the System Builder menus.

Now you want to "DO" the *Add Program Description* item, so press the PF1 key on your keyboard.

After you press DO, your screen looks like this:



System SAMPLE - Editing Program

Program		
Debug	Yes	[✓ No]
Run	[✓ Yes]	No
Init required	Yes	[✓ No]
Mode	[✓ User]	Kernel
User stack (initial)	1	pages
Kernel stack	1	pages
Job priority	16	

All of the parameters are set to their defaults; these are the choices highlighted on the screen. The defaults are fine for this application. You only need to supply the name of your program. So type

```
SAMPLE.EXE
```

after *Program* (where the blinking cursor is) and press DO (the PF1 key).

This action returns you to the main System Builder menu where the *Add Program Description* item is highlighted.

Now you are ready to build the system, so press the "up-arrow" cursor movement key (the key with the ↑ symbol on it) until the item *Build System* is highlighted. Press DO (PF1).

After a few seconds, the name of the system you just built is displayed at the top of your screen:

```
System disk:[directory]SAMPLE.SYS;1
```

where *disk* and *directory* identify where SAMPLE.SYS is located. This is followed by a message stating the size of the system image in pages and in kilobytes (Kbytes).

Now that you have built the SAMPLE.SYS system image, your directory contains the files SAMPLE.DAT and SAMPLE.SYS.

If you like, enter this command to see what is in SAMPLE.DAT:

```
$ TYPE  SAMPLE.DAT
```

SAMPLE.DAT contains only the data

```
program SAMPLE.EXE
```

which is the only item you edited on the System Builder menus.

Step 5: Configure DECnet–VAX. Your network is probably already configured properly. To check this quickly, enter this command at your host terminal:

```
$ RUN  SYS$SYSTEM:NCP
```

This command invokes the Network Control Program (NCP) which returns the NCP> prompt. Following this prompt, enter the command:

```
NCP>SHOW NODE name CHARACTERISTICS
```

where *name* is the name of your MicroVAX target node. For instance, if your target node is named SHRIMP, you would type:

```
NCP>SHOW NODE SHRIMP CHARACTERISTICS
```

After you press Return, you get several lines of information. If this information includes a service circuit designation (something like "UNA-0") and a hardware address (something like "AA-00-03-01-12-49"), your network is configured properly; please skip the rest of this step and continue with Step 6 on page 1-19.

If you do not see a service circuit designation or hardware address, or you get an error message, please follow the configuration instructions below.

The first thing you need to do is determine the target node's DEQNA hardware address. The best way to do this is to go to the MicroVAX console terminal, place the MicroVAX in console mode, and examine a series of DEQNA device registers.

To place the MicroVAX in console mode, press the Halt button on the front panel twice—once to latch it in, and once to release it. The >>> prompt on the MicroVAX's console terminal indicates that it is in console mode.

Now examine the first DEQNA device register by typing this command:

```
>>>E/P/W 20001920
```

and pressing Return. The information that comes back looks like this:

```
P 20001920 0000FFAA
```

The last two characters, AA, are the first two characters of the DEQNA's hardware address.

Continue typing the command "E +" following the console mode prompt until you have five more sets of characters. Be sure to type a space between the "E" and the "+." Press Return to enter each command. The last two characters of each output string, when joined consecutively and separated by hyphens, form the DEQNA's hardware address.

For example:

```
>>>E/P/W 20001920
P 20001920 0000FFAA
>>>E +
P 20001922 0000FF00
>>>E +
P 20001924 0000FF03
>>>E +
P 20001926 0000FF01
>>>E +
P 20001928 0000FF12
>>>E +
P 2000192A 0000FF49
>>>
```

Here, the hardware address is AA-00-03-01-12-49. (The last two characters of each output string are not highlighted on your screen; they are only highlighted here for illustration.)

Once you have the MicroVAX's DEQNA hardware address, return to your host terminal, which should still be showing the NCP> prompt. (If not, enter the command RUN SYS$SYSTEM:NCP.)

Assuming your host is a VAX 11/730, 11/750, 11/780, 11/785, or 8600, enter these commands next:

```
NCP>SET NODE name HARDWARE ADDRESS #
NCP>SET NODE name SERVICE CIRCUIT UNA-0
```

where *name* is the node name of your MicroVAX target machine, and # is the MicroVAX's hardware address you just determined. For example, if SHRIMP's hardware address is AA-00-03-01-12-49, the correct commands would be:

```
NCP>SET NODE SHRIMP HARDWARE ADDRESS -
_  AA-00-03-01-12-49
```

```
NCP>SET NODE SHRIMP SERVICE CIRCUIT UNA-0
```

If your host is a MicroVAX, use the same SET NODE commands, but substitute QNA-0 for UNA-0. These SET NODE commands identify the target machine to the host computer.

Next, issue these commands to enable the host computer to recognize boot-request messages from the target machine:

```
NCP>SET CIRCUIT UNA-0 STATE OFF
NCP>SET CIRCUIT UNA-0 SERVICE ENABLED
NCP>SET CIRCUIT UNA-0 STATE ON
```

Again, if your host computer is a MicroVAX, use QNA-0 instead of UNA-0.

These SET NODE and SET CIRCUIT commands are in effect only until the host system is rebooted. To make them permanent, use the DEFINE and SET commands described under "Configuring a Host for Down-Line Loading" and "Adding the Target Machine to the Host Node Data Base" in Chapter 14 of this manual.

Step 6: Down-line load and run the program. First, enter this command following the NCP> prompt to tell the host computer what file to send to the target machine:

```
NCP>SET NODE name LOAD FILE -
_ disk:[directory]SAMPLE.SYS
```

where *name* is the name of the MicroVAX target node, and *disk* and *directory* identify where SAMPLE.SYS is located. For instance, if SAMPLE.SYS is in a directory named MALCOLM, and this directory is located on a disk named WRKD$, the command for target node SHRIMP is:

```
NCP>SET NODE SHRIMP LOAD FILE -
_ WRKD$:[MALCOLM]SAMPLE.SYS
```

Next, go to your target MicroVAX and place it in console mode by pressing the Halt button on the front panel twice—once to latch it in and once to release it. The prompt >>> appears on the MicroVAX's console terminal to indicate console mode.

Now enter this command at the MicroVAX's console terminal following the console mode prompt:

```
>>>B XQA0
```

A minute or two after you press Return, you should see the messages VAXELN V2.0 and Hello!. The Hello! is the output of your program. So congratulations! You've just written, down-line loaded, and executed your first VAXELN application.

If instead of Hello! you get the error message No response from load server XQA0, you may have entered the NCP SET NODE *name* LOAD FILE command incorrectly, or the network may not be configured properly. Try repeating Steps 5 and 6. If the down-line load still does not work, see Chapter 14, "Booting and Down-Line Loading."

Before continuing with Part 2, enter the EXIT command after the NCP> prompt at your host terminal to return to the VMS command interpreter.

Part 2

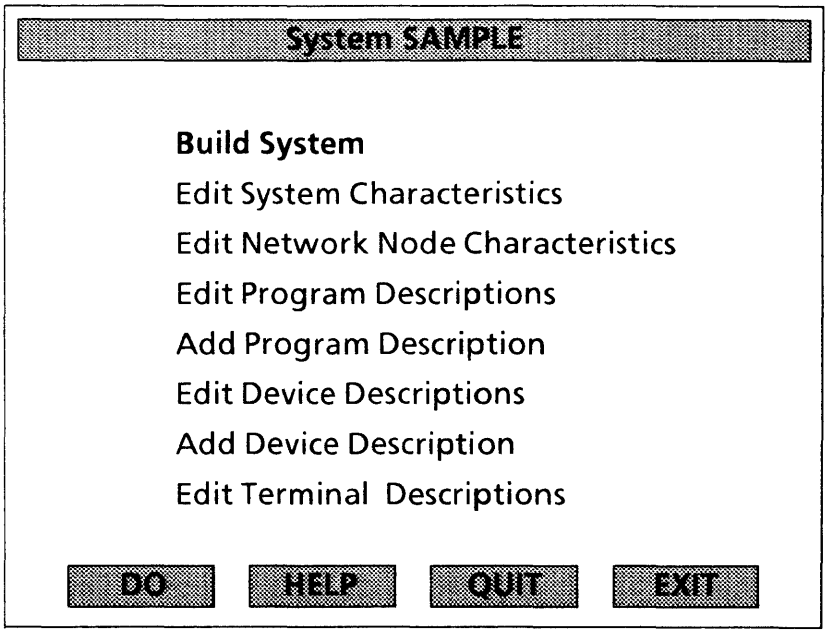
For Part 2, the object is to use the debugger to initiate the down-line load, and control the execution of the program on the target machine from the host terminal instead. You can also use the debugger to correct errors in the source code, although you will not need to in this example.

For this part, first be sure there are no bootable disks on the target machine. The easiest way to do this is to remove all diskettes from the diskette drives, and press the fixed disk Ready button on the MicroVAX front panel once so that it latches in. The green light in the center of the button turns off, indicating that the fixed disk is off-line. If your target MicroVAX has an additional fixed disk, place it off-line also by pressing the associated Ready button.

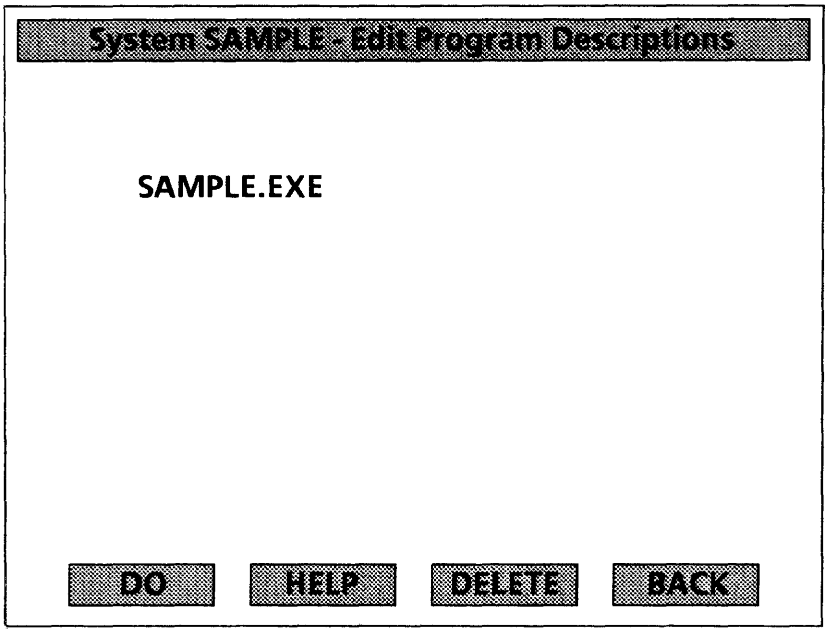
Step 1: Build a new system. You want to build a new system that can be down-line loaded to the target computer, but debugged from the host terminal, so enter this command to invoke the VAXELN System Builder:

```
$ EBUILD SAMPLE
```

Once again, your screen looks like this:


System SAMPLE
Build System
Edit System Characteristics
Edit Network Node Characteristics
Edit Program Descriptions
Add Program Description
Edit Device Descriptions
Add Device Description
Edit Terminal Descriptions
DO
HELP
QUIT
EXIT

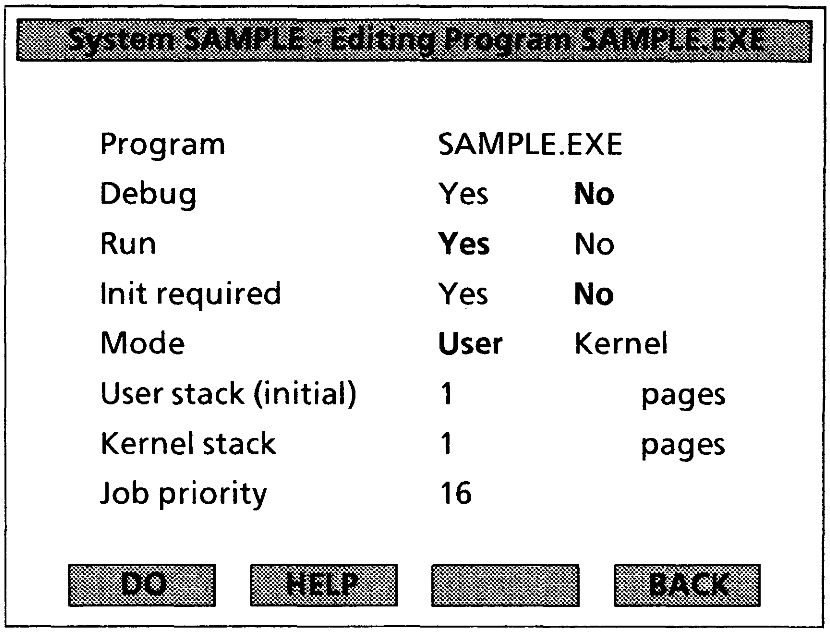
This time, press the "down-arrow" cursor movement key (the key with the ↓ symbol on it) until the item *Edit Program Descriptions* is highlighted, and press the DO key (PF1). Your screen now looks like this:


System SAMPLE - Edit Program Descriptions

SAMPLE.EXE

Press the DO key again to display the next screen.

Here is the next System Builder menu screen:


System SAMPLE - Editing Program SAMPLE.EXE

Program	SAMPLE.EXE	
Debug	Yes	**No**
Run	**Yes**	No
Init required	Yes	**No**
Mode	**User**	Kernel
User stack (initial)	1	pages
Kernel stack	1	pages
Job priority	16	

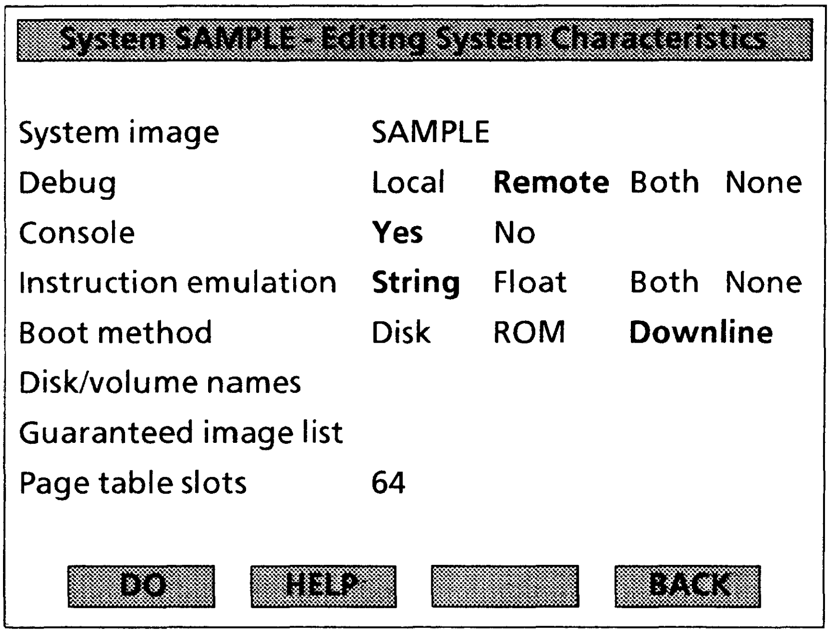
Notice that the information for *Program* is filled in for you with the value you entered last time.

On this screen, turn debugging on by pressing the "down-arrow" cursor movement key once (the key with the ↓ symbol on it), then the "left-arrow" cursor movement key once (the key with the ← symbol on it). After you do this, "Yes" is highlighted on the *Debug* line item.

Now press the DO key (PF1). This takes you back to the previous menu screen.

Now press the BACK key (PF4) to return to the main menu. On this screen, the item *Edit Program Descriptions* is highlighted, because that is what you were just doing.

Now press the "up-arrow" cursor movement key (↑) twice, so that the item *Edit System Characteristics* is highlighted. Press DO to get this screen:


System SAMPLE - Editing System Characteristics

System image	SAMPLE			
Debug	Local	**Remote**	Both	None
Console	**Yes**	No		
Instruction emulation	**String**	Float	Both	None
Boot method	Disk	ROM	**Downline**	
Disk/volume names				
Guaranteed image list				
Page table slots	64			

On this screen, press the "down-arrow" cursor movement key (↓) twice, then the "right-arrow" cursor movement key (→) once to set the *Console* option to "No". The combination of "Remote" *Debug* and "No" *Console* causes the program output to be displayed at the host terminal used for debugging.

Now press DO. This returns you to the main System Builder menu. Press the "up-arrow" cursor movement key (↑) once so that *Build System* is highlighted, and press DO. After a few seconds, the name of the system you just built is again displayed at the top of your screen:

System *disk*:[*directory*]SAMPLE.SYS;2

This is followed by a message stating the size of the system image in pages and in kilobytes (Kbytes).

You have just built a second SAMPLE.SYS system, and your directory now contains second versions of the files SAMPLE.DAT (SAMPLE.DAT;2) and SAMPLE.SYS (SAMPLE.SYS;2).

If you like, see what is in SAMPLE.DAT;2:

```
$ TYPE SAMPLE.DAT
```

SAMPLE.DAT now contains the data

```
characteristic /noconsole
program SAMPLE.EXE /debug
```

which are the items you edited on the System Builder menus.

Step 2: Down-line load and run the program. Now you are ready to enter the EDEBUG command at your host terminal, which invokes the VAXELN debugger and initiates the down-line loading of your program. Enter this command at your host terminal:

```
$ EDEBUG/LOAD = SAMPLE.SYS name
```

where *name* is the name of the MicroVAX target node. For instance, the command for target node SHRIMP is:

```
$ EDEBUG/LOAD = SAMPLE.SYS SHRIMP
```

In Part 1, you entered the NCP SET NODE *name* LOAD FILE command and bootstrapped the target MicroVAX from its console terminal. This time, the EDEBUG command does the NCP command for you, and causes the target node to reboot.

After a few seconds, the messages

```
Edebug V2.0
Loading "name".
Connecting to "name".
```

appear on your host terminal, where *"name"*, again, is the name of the MicroVAX target node.

The target node should reboot at this point. If it does not reboot, it is probably configured incorrectly, and additional error messages will appear on your host terminal.

After the Connecting to "name" message on your host terminal, you may see the message Retrying connect, depending on how long it takes the host to down-line load the target machine. This is only an informational message.

After a few more seconds, the screen on your host terminal looks like this if you used the Pascal program:

```
Job 5, process 1, program SAMPLE needs attention.
      Module SAMPLE
      3: BEGIN
>>    4:      writeln ('Hello!');
      5: END.
----------------------------------------------------------------






   Edebug V2.0
   Loading "name".
   Connecting to "name".
   Connected to "name", awaiting debug activity.
   Loading traceback data from: disk:[directory]SAMPLE.EXE;1
Edebug 5,1>
```

or like this if you used the C program:

```
Job 5, process 1, program SAMPLE needs attention.
    Module SAMPLE
    1: sample()
>>2: {
    3:      printf("Hello!\n");
    4: }
------------------------------------------------------------------




  Edebug V2.0
  Loading "name".
  Connecting to "name".
  Connected to "name", awaiting debug activity.
  Loading traceback data from: disk:[directory]SAMPLE.EXE;1
Edebug 5,1 >
```

You are now in the debugger, as indicated by the prompt Edebug 5,1>. The "5" means job 5, and the "1" means process 1. The source code for your program is displayed in the upper half of the screen. The two angle brackets (>>) point to the current line of code.

At this point, the debugger is waiting for you to enter a command, to debug your program.

There are many commands you can use in the debugger. These commands are explained in Chapter 15, "Debugging."

To finish this example and see the results of your program execution on your host terminal, enter the command GO following the Edebug 5,1> prompt.

Your final screen looks like this:

```
Job 5, process 1, program SAMPLE running.
------------------------------------------------------------------









   Edebug V2.0
   Loading "name".
   Connecting to "name".
   Connected to "name", awaiting debug activity.
   Loading traceback data from: disk:[directory]SAMPLE.EXE;1
Edebug 5,1 >go
   Job 5, process 1, program SAMPLE has exited.

Hello!
```

Note: On your screen, the program output Hello! might be displayed before the message: Job 5, Process 1, program SAMPLE has exited.

Once again, congratulations! You have just down-line loaded and controlled the execution of your program from your host terminal.

Now, to exit, press CTRL/C (that is, hold the CTRL key down and press the C key) to get the debugger's attention, then enter the command EXIT to return to the VMS command interpreter.

This concludes the example of building, down-line loading, and executing a VAXELN application. See the *VAXELN Application Design Guide* for more examples.

Chapter 2
Kernel Objects

Introduction

The VAXELN kernel services are object oriented. An *object* is a data structure that the kernel uses to represent a resource or some ongoing activity, such as a process's executions.

To ensure that the kernel can guarantee the integrity of the objects, the fields in the objects are not directly accessible to a program. Instead, when a 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. The program then uses the identifying value to refer to the object in further calls to kernel procedures. When the program no longer needs the object, the identifying value can be used again in a call to the DELETE procedure.

The types of VAXELN kernel objects are: PROCESS, AREA, EVENT, SEMAPHORE, MESSAGE, PORT, NAME, and DEVICE.

To make programming with kernel objects easy, the VAXELN Pascal language has predeclared system data types that represent the identifying values of each kind of object. A program thus declares variables of these types—for example, a variable of type PROCESS to hold the identifying value of a process.

For the C programmer, the data type definitions for the VAXELN kernel objects are contained in the #include module **$vaxelnc** in VAXELNC.TLB.

Objects are always created dynamically; there are no constants for the various object types. If you want to use an object throughout a job, you must call the appropriate CREATE kernel service at the beginning of the job, saving the returned object value in a variable. The variable's name can then be used throughout the program to name the object. For example,

```
CREATE_SEMAPHORE(main_lock)
```

creates a SEMAPHORE object at the beginning of the job, saving the returned value that identifies the semaphore in the variable main_lock. The semaphore can then be waited on or signaled, for example, anywhere in the program by using the variable's name to reference the object:

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

Finally, when the program no longer needs the object, it can be deleted:

```
DELETE(main_lock)
```

Note that, except for PORT values and AREA values for jobs on the same node, an object's identifying value is only valid within a job (even in cases where the object is known in more than one job).

This chapter discusses each of the types of VAXELN kernel objects, listing their associated properties, summarizing the operations in which their object values are used, and describing the internal representation of those values. In addition, the kernel's implementation of the objects is summarized at the end of the chapter.

PROCESS Object

A PROCESS object represents the current context of a thread of execution in a program within a job. A job refers to a family of cooperating processes that share memory and other resources; there can be any number of processes within a job.

A PROCESS object has the following associated 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)

These properties and the kernel services that affect the state of PROCESS objects are discussed in Chapter 3, "Processes and Jobs."

Operations with PROCESS Values

PROCESS values are used in the following operations:

- A process is created with the CREATE_PROCESS procedure, which returns an identifying PROCESS value.
- A process obtains its own value with the CURRENT_PROCESS procedure.
- A process's priority is altered by giving the value to SET_PROCESS_PRIORITY.
- A process's execution is suspended or resumed by giving the value to SUSPEND or RESUME.
- One process waits for another to terminate by giving the value to WAIT_ALL or WAIT_ANY.

- One process forces another process into an exception condition by giving the value to SIGNAL.
- An immediate exit from a process is forced by giving the value to EXIT.
- A process is deleted from the application by giving the value to DELETE.

Internal Representation of PROCESS Values

PROCESS values are represented internally as longwords (32 bits) that are used by the kernel. They are valid only within their own job.

AREA Object

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

An AREA object has the following associated properties:

- A character-string name of up to 31 characters that supplies a name for the area
- One of the states *signaled* or *free*
- A list of processes waiting for access to the area
- The associated region of memory

These properties and the kernel services that affect the state of AREA objects are discussed in Chapter 3, "Processes and Jobs."

Operations with AREA Values

AREA values are used in the following operations:

- An area is created or an existing area mapped with the CREATE_AREA procedure, which returns an identifying AREA value and a pointer to the region of memory.
- To gain exclusive access, a process waits for the signaling of an area by giving the value to WAIT_ALL or WAIT_ANY.
- An area is signaled by giving the value to SIGNAL.
- An area is deleted from the application by giving the value to DELETE.

Internal Representation of AREA Values

AREA values are represented internally as longwords (32 bits) that are used by the kernel to identify a particular area and its properties. An AREA object occupies one block (128 bytes) of kernel pool.

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

EVENT Object

An EVENT object records occurrences of events in real time and stores that information until explicitly cleared by a program.

An EVENT object has the following associated properties:

- One of the states *signaled* or *cleared*
- A list of processes waiting for the event to be signaled

These properties and the kernel services that affect the state of EVENT objects are discussed in Chapter 4, "Synchronization."

Operations with EVENT Values

EVENT values are used in the following operations:

- An event is created with the CREATE_EVENT procedure, which returns an identifying EVENT value.
- A process waits for the signaling of an event by giving the value to WAIT_ALL or WAIT_ANY.
- An event is signaled by giving the value to SIGNAL.
- An event is cleared by giving the value to CLEAR_EVENT.
- An event is deleted from the application by giving the value to DELETE.

Internal Representation of EVENT Values

EVENT values are represented internally as longwords (32 bits) that are used by the kernel to locate the actual data and its associated properties, such as its current signaled/cleared state. An EVENT object occupies one block (128 bytes) of system pool.

SEMAPHORE Object

A SEMAPHORE object is used to protect a resource (including other data) from simultaneous access or to control or "meter" the execution of processes that require some limited resource. The term *semaphore*, after the device that allows only one railroad train to proceed down a section of track, suggests the most widespread use in programming: guarding a single resource (the track, in that case) from simultaneous use.

A SEMAPHORE object has the following associated properties:

- A count of the number of processes that will be allowed to obtain the semaphore without waiting for some other process to signal it
- The maximum allowed value for count, which is the maximum number of processes that may simultaneously have the semaphore
- A list of processes waiting for the semaphore to be signaled

These properties and the kernel services that affect the state of SEMAPHORE objects are discussed in Chapter 4, "Synchronization."

Operations with SEMAPHORE Values

SEMAPHORE values are used in the following operations:

- A semaphore is created with the CREATE_SEMAPHORE procedure, which returns an identifying SEMAPHORE value.

- A process waits for the signaling of a semaphore by giving the value to WAIT_ALL or WAIT_ANY. Satisfying the wait decrements the semaphore count.
- A semaphore is signaled by giving the value to SIGNAL. This increments the semaphore count.
- A semaphore is deleted from the application by giving the value to DELETE.

Internal Representation of SEMAPHORE Values

SEMAPHORE values are represented internally as longwords (32 bits) that are used by the kernel to locate the actual object and its associated properties, such as its current count. A SEMAPHORE object occupies one block (128 bytes) of system pool.

MESSAGE Object

A MESSAGE object is used to send data from a job to a port, which will usually be in another job.

A MESSAGE object has the following associated properties:

- The message data
- The message length

These properties and the kernel services that affect the state of MESSAGE objects are discussed in Chapter 5, "Interjob Communication."

Operations with MESSAGE Values

MESSAGE values are used in the following operations:

- A message is created and its data mapped into the job's P0 address space with the CREATE_MESSAGE procedure, which returns an identifying MESSAGE value and a pointer to the data.

- A message is sent to a specified message port by giving the MESSAGE and PORT values to SEND, which removes the message data from the sending job's address space.
- A message is removed from a message port and its data mapped into the receiving job's P0 address space by giving the PORT value to RECEIVE, which returns an identifying MESSAGE value and a pointer to the message data.
- A message is deleted from the application by giving the value to DELETE.

Internal Representation of MESSAGE Values

MESSAGE values are represented internally as longwords (32 bits) that are used by the kernel to identify a particular message and its properties.

The associated message data is allocated from physically contiguous 512-byte pages of 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 make the message data ideally suited for a VAX DMA-device I/O buffer.) In addition, the fact that P0 is used allows the message data to be shared by all processes in a job.

PORT Object

A PORT object (or, informally, *message port*) is a destination for messages. Each port belongs to a particular job, but it can be referenced from any job in the local area network. In contrast to other object values, the identifying value of a port is meaningful in all jobs in all nodes in the network.

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

A PORT object has the following associated properties:

- The maximum number of queued messages
- A list of queued messages (which will be removed from the port by the RECEIVE procedure)
- The state of the port as regards circuit connection: unconnected, connected, or in one of the special states arising during establishment of a connection
- If connected, the PORT value identifying the port to which it is connected

These properties and the kernel services that affect the state of PORT objects are discussed in Chapter 5, "Interjob Communication."

Operations with PORT Values

PORT values are used in the following operations:

- A port is created with the CREATE_PORT procedure, which returns an identifying PORT value.
- A job obtains its own, unique PORT value for use in interjob communication with the JOB_PORT procedure.
- A process waits for the receipt of a message by giving the PORT value to WAIT_ALL or WAIT_ANY. When a message arrives at the port,

any process waiting on that port is allowed to continue if its wait conditions are otherwise satisfied. The receiver process uses the RECEIVE procedure to obtain the message. Note that only processes in the job that creates a port can receive messages from that port with RECEIVE.

- Ports are connected and disconnected in circuits with the CONNECT_CIRCUIT and DISCONNECT_CIRCUIT procedures, respectively. In addition, ACCEPT_CIRCUIT allows a process to wait for a circuit connection request on a specified port.
- A port is deleted from the application by giving the value to DELETE.

Internal Representation of PORT Values

PORT values are 128-bit values that uniquely identify a message port. The representation is shown in Figure 2-1.

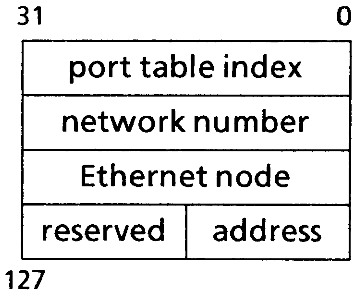


Figure 2-1. PORT Value Representation

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

NAME Object

A NAME object is an entry in a name table that associates character-string names with message ports. The local name table (maintained by the kernel) is used only within a node. The universal name table (maintained with the aid of the Network Service) establishes port names valid at all nodes in the local area network.

A NAME object has the following associated properties:

- A character string of up to 31 characters that names an existing message port
- The PORT value identifying the message port
- One of the properties *local* or *universal*

These properties and the kernel services that affect the state of NAME objects are discussed in Chapter 5, "Interjob Communication."

Operations with NAME Values

NAME values are used in the following operations:

- A name is created with the CREATE_NAME procedure, which returns an identifying NAME value.
- An associated PORT value is obtained by giving the name string to TRANSLATE_NAME, which returns the associated PORT value.
- A name is deleted from the application by giving the value to DELETE.

Internal Representation of NAME Values

The identifying NAME value is a longword (32 bits). The NAME object itself 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 Network Service that is the network's current *name server.* (See Chapter 7, "The Network Service," for more information.)

DEVICE Object

A DEVICE object provides the means for a program's interrupt service routine to signal the occurrence of a particular device controller interrupt to a waiting process. The interrupt service routine is called by the kernel each time the connected interrupt occurs; it can signal the DEVICE object to synchronize itself with processes in the job that created the object.

A DEVICE object has the following associated properties:

- A set of device characteristics established with the System Builder
- A communication region
- An interrupt service routine, which is invoked by the kernel when an appropriate interrupt occurs and is passed the DEVICE value and communication region

These properties and the kernel services that affect the state of DEVICE objects are discussed in Chapter 6, "I/O Devices and Interrupt Handling."

Operations with DEVICE Values

DEVICE values are used in the following operations:

- A DEVICE object is created with the CREATE_DEVICE procedure, which returns an identifying DEVICE value.
- A process waits for the signaling of a DEVICE object from an interrupt service routine by giving the value to WAIT_ALL or WAIT_ANY.
- A device is signaled from an interrupt service routine by giving the value to SIGNAL_DEVICE.
- A DEVICE object is deleted from the application by giving the value to DELETE.

Internal Representation of DEVICE Values

DEVICE values are represented internally as longwords (32 bits) that are used by the kernel to locate the actual object containing its associated properties, such as the address of its communication region. A DEVICE object occupies one block (128 bytes) of pool. If an interrupt service routine is connected, it also requires one block of pool for its dispatcher.

Kernel Implementation of Objects

Although it is usually not necessary in VAXELN programming to know the actual details of the kernel's implementation, a few points are useful in answering some system configuration questions:

- The kernel allocates all 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 all the blocks into system space, and links all the blocks into a list of free blocks. The fixed size of the blocks makes allocating and deallocating objects very 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 indices. The indices 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 table grows dynamically in size as the job creates more objects. The table itself is allocated from pool blocks and starts with one top-level block and one second-level block. The top-level block can point to 32 second-level blocks, and each second-level block can point to 32 objects. Therefore, a job can have up to 1024 objects created at one time.

The preceding description applies to all objects except ports. Because a PORT value is valid anywhere in the network, it also includes the Ethernet node address and additional fields reserved for future use. Thus, a PORT value is 128 bits long. Also, the indices in a PORT value are used for a table that describes all the ports in the system, rather than a per-job table. 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 representation of identifying values may seem complicated, it allows the validation of such a value to be done in a very small number of VAX instructions. Furthermore, the representation is not important from a programming standpoint.

Chapter 3
Processes and Jobs

Introduction

In VAXELN, a program is executed as a *job*. Jobs are created dynamically with the CREATE_JOB procedure to execute a specific program image that was included with the System Builder. When you build a system, you have the option of specifying program images to be executed automatically when the system is started on the target hardware; jobs are created automatically for these images.

The CREATE_JOB procedure is also used to execute program images that are loaded with the dynamic program loader after the initial system is built (see "Program Loader Utility Procedures," later in this chapter). In addition, the VAXELN debugger can create jobs with the debugger command CREATE JOB. (See Chapter 15, "Debugging," for more information.)

Processes are the execution agents for VAXELN programs or for concurrently scheduled parts of programs. The main thread of execution for a program is executed by a *master* process created implicitly by the kernel when the program is started. In VAXELN Pascal, the main routine of a job's master process is the PROGRAM block. In C, the main routine of a job's master process is the function **main**.

A *subprocess* of a job is created by a call to the CREATE_PROCESS procedure. Each subprocess executes a special routine that defines the executable code and data available to one or more dynamically created processes. In VAXELN Pascal, this routine is called a process block. In C, the routine is called a function.

Within a system, there can be any number of jobs executing the same program. Similarly, within each job, there can be any number of subprocesses executing the same process block or function. A job, therefore, contains one master process executing the main program and zero or more subprocesses.

The configurations of jobs in a running VAXELN application can be any combination of the following:

- A single job executing on a single processor (termed multitasking in the case where there are subprocesses executing with the main program)
- Multiple jobs executing concurrently on a single processor (multiprogramming)
- Multiple jobs executing on several single processors, which are connected by the Ethernet (distributed processing)

This chapter discusses jobs viewed as process families, process states, job and process scheduling and termination, and the VAXELN kernel services relating to jobs and processes. Memory management and memory allocation procedures are then discussed, followed by interjob data sharing and the kernel services relating to interjob data sharing.

Jobs Viewed as Process Families

The processes associated with a running program can be thought of as a "process family" in the following senses:

- There is a hierarchy implied in the way they are created. That is, a CREATE_JOB call creates a job that runs a specific program; that program then can call CREATE_PROCESS to execute any of its associated process blocks or functions. The execution of the master process holds the object values of all the subprocesses; thus, if the master process exits, the subprocesses are deleted, as are all memory and objects created by the job. There is no hierarchy implied by subprocesses, however. If a subprocess creates another subprocess and then exits, the new subprocess continues to run.

- All the processes in a job can share externally declared data. In addition, data can be shared among jobs on a single node in a VAXELN network via the use of AREA objects. Otherwise, two jobs can exchange information only in the form of messages.

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, one VAXELN program can create a separate process family, where 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 entirely independent of all others, with its own data and code.

Process States

Each process in a VAXELN system is always in one of the following four *process states*:

Running. A process in this state has control of the CPU (is executing). Only one process can be running at any given time. If control is in an interrupt service routine, no processes are running.

Ready. A process in this state is not running but is ready to run as soon as possible. This is the initial state of every process, immediately after its creation. Any number of processes can be in the Ready state.

Waiting. A process in this state is waiting for some specified set of conditions to be satisfied. It is waiting as a result of calling a kernel service such as WAIT_ANY or RECEIVE that may wait for an event, for a particular amount of time to elapse, for the receipt of a message, and so forth.

Suspended. A process in this state is suspended as a result of some process calling the SUSPEND procedure. The process is not eligible to execute (that is, cannot enter the Ready state) until it is resumed explicitly with the RESUME procedure.

State Transitions

Transitions from one process state to another describe the behavior of the system according to the following rules:

- The initial state of every process is Ready.
- Among jobs with a Ready process, the VAXELN kernel continually selects the process with the highest priority within the job with the highest priority and changes its state to Running. The previously Running process becomes Ready.
- When the currently Running process calls a kernel service that waits for a set of conditions to be satusfied, the process enters the Waiting state.
- When the wait conditions are satisfied for a Waiting process, its state becomes Ready.
- When the currently Running process, any Waiting process, or any Ready process calls SUSPEND, the process enters the Suspended state.

- When RESUME is applied to a Suspended process, the process reenters its previous state. It becomes Ready if it was previously Ready. If it was previously Waiting, it reenters the Waiting state until the specified conditions are satisfied. (It is possible that the wait conditions were satisfied during the suspension; if so, the resumed process will then change from Waiting to Ready.)

Job and Process Scheduling

VAXELN uses scheduling methods to give the appearance and effect of simultaneous execution of its processes. You assign *priorities* to jobs and processes, where a high priority means that the job or process should be given preference over others when it is ready to execute.

There are 32 levels of job priority, numbered from 0 (highest) to 31. The default priority of a program is established with the System Builder or with the LOAD_PROGRAM procedure and can be reset within the job with the SET_JOB_PRIORITY procedure.

Within a job, processes have 16 levels of priority, numbered from 0 (highest) to 15. The default priority of a program's processes is established with the System Builder or with the LOAD_PROGRAM procedure and can be reset in a job with the SET_PROCESS_PRIORITY procedure.

Figure 3-1 illustrates the structure of job and process scheduling priorities.

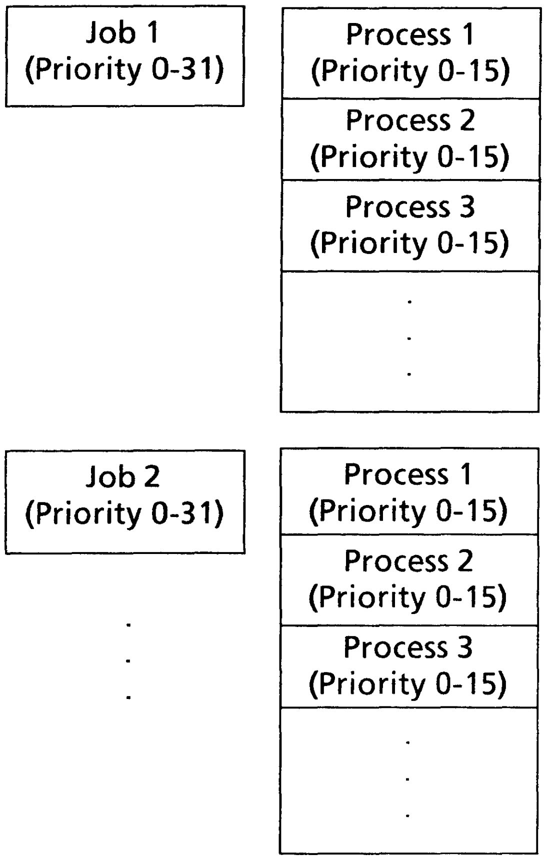


Figure 3-1. Job and Process Priorities

Job priority determines the 16-priority range in which that job's processes are scheduled. Jobs are rescheduled when, within a job not currently executing, a process enters the Ready state and that process's priority is higher than the priority of the current job's executing process.

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

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

In the case where two or more jobs have equal priority, the Ready process with the highest priority in any of the jobs is given control, preempting all lower priority processes. In other words, all jobs at a given priority are scheduled against each other, with current control always being passed to the job containing the Ready process with the highest priority.

This scheduling method implies that the job and process priorities are unified to form one of 512 possible combined priority values (32 job priorities x 16 process priorities) and that the processes are scheduled against each other using this combined value. In fact, jobs are scheduled first, then processes; the overall priority of a process, therefore, is always limited by the priority of its job.

Figure 3-2 illustrates the internal representation of the combined job and process priority values.

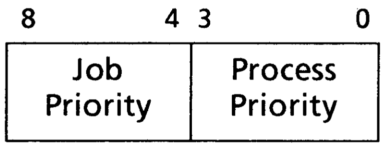


Figure 3-2. Combined Priority Representation

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 provides a broad mechanism by which, for example, a process could control the access to a data set. (A finer mechanism is the use of semaphores, discussed in the next chapter.)

Since processes are automatically rescheduled in a predictable way, you can design a system in which there are no important or noticeable delays in a program's operation, even though it spends at least some of its time sitting idle while another program executes.

The definition of "important delay" is a large part of the definition of "real-time performance" for your application. It is clearly impossible to exactly synchronize a computer or computer program with external phenomena; instead, to satisfy the practical definition of *real time*, 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 speaking, real-time systems are best designed by first assuring that 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. Most of the issues related to synchronizing processes with each other or with external events are summarized in Chapter 4, "Synchronization."

Initialization Programs and System Startup

When a VAXELN system is built, programs described to the System Builder (see Chapter 13, "System Development") can be given the characteristic *Init required.* This characteristic means that the program is an "initializing" program that will be started in order of job priority when the system is started.

If you say that initialization is required, the program is started and no jobs of a lower priority are started until it 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 started.

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 started, one at a time, in the order of their job priorities, followed by the non-initializing programs. Here, program1 is started first, followed by program3 (remember that with job priorities, low numbers mean high priorities). When program1 calls INITIALIZATION_DONE, program3 is started immediately; 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 complete or call INITIALIZATION_DONE. Program4 is not started automatically; it must be activated by a CREATE_JOB call from one of the other programs.

Program Loader Utility Procedures

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:

- LOAD_PROGRAM, which loads a specified image file into the currently 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 be resident on the system or on a remote node; there is no 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.

- UNLOAD_PROGRAM, which unloads the specified program from the system.

One restriction is that any shareable images that the dynamically loaded program references must be included in the system at system build time. The *Guaranteed image list* item 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 item on the same menu, *Dynamic program space*, specifies the number of memory pages that can be used by dynamically loaded programs. It is a quota and does not cause the pages to be allocated until the program is actually loaded. (For more information , see Chapter 13, "System Development.")

Job and Process Termination

A job persists as long as its master process exists. The job is terminated when its master process terminates. The termination of a job also means that all the subprocesses, data, and objects are deleted.

Once created, a process persists until it terminates, either when the end of its main routine is encountered, or as a result of one of the following explicit actions:

- The execution of the EXIT procedure.
- The deletion of the process with the DELETE procedure.
- The receipt of a QUIT exception (caused by invoking SIGNAL from this or another process).

- The occurrence of other unhandled signals.

When a process is terminated, only some of the resources that it acquired during its execution are freed. Any objects that 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 a job. The only resources that are freed are the process's private memory resources; that is, the process's P1 virtual address space or stack space (see "Memory Management," later in this chapter) and the kernel's pool space associated with the process's activation. Only when a job's master process is deleted are the undeleted objects acquired by the job's processes deleted.

Caution: For the reasons just described, care should be taken when using the DELETE procedure on a process. Processes that terminate in this way are not terminated in an orderly manner and cannot be restarted. Deletion of a process is intended as an emergency method to stop a process; SIGNAL and EXIT provide a more controlled means of forcing a process to stop.

Processes are terminated in an orderly manner with the EXIT procedure or when they return from the outermost procedure block. (See Chapter 11, "Exception Handling," for a discussion of VAX stack architecture and call frames.)

The orderly termination of a process causes two special events to happen:

- If the debugger is active in the process, the user is notified that the process is going away.
- A special *exit handler* feature (see below) is activated so that any dangling resources can be cleaned up by the code that allocated the resource.

Exit Utility Procedures

VAXELN provides two utility procedures that can be used to establish an exit handler to perform cleanup operations following the termination of a job with the EXIT procedure. The $EXIT_UTILITY module provides the following procedures:

- DECLARE_EXIT_HANDLER, which calls an exit handler routine defined by the program
- CANCEL_EXIT_HANDLER, which deletes a specific exit handler routine

Kernel Services for Processes and Jobs

The kernel services affecting the state of PROCESS objects are summarized below.

CREATE_JOB Procedure

The CREATE_JOB procedure creates a new job which executes a specified program image, returning the new job port value. This value can be used by the caller to send messages to the new job. The same value can be obtained within the new job by the JOB_PORT procedure. An optional list of string arguments can be passed to the program.

An optional argument identifies a port that receives notification 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). Note that an unhandled exception condition causes the value of the exception to be returned.

Note that 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 LOAD_PROGRAM procedure after the initial system is built.

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 zero 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. Note that an unhandled exception condition causes the value of the exception to be returned.

CURRENT_PROCESS Procedure

The CURRENT_PROCESS procedure returns the PROCESS value identifying the process from which it is called.

DELETE Procedure

The DELETE procedure removes the PROCESS object from the system. When a process is deleted, if any other 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 also deleted, along with all data and kernel objects created by any processes in the job. The exit status of a deleted process is KER$_NO_STATUS.

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.

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

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

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. Note that the procedures ENABLE_SWITCH and DISABLE_SWITCH keep a count of 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 given process.

EXIT Procedure

The EXIT procedure causes an immediate exit from the calling process. The procedure is similar to deleting the current process, except it can optionally return an exit status to the process that created it. If process switching was disabled by the process, it 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.

GET_USER Procedure

The 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 currently 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, GET_USER can only provide information 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* characteristic on the *Edit Network Node Characteristics* System Builder menu is "No"), GET_USER returns zero for the UIC parameter.

INITIALIZATION_DONE Procedure

The INITIALIZATION_DONE procedure informs the kernel that the calling program has completed an initialization sequence, and other programs can be started if specified. If initializing programs do not call this procedure, they run to completion before any other program can be started. If they do call this procedure, they continue running at the next statement, and the next program in the sequence can be started.

Note that if a program with the System Builder characteristic *Init required* does not call the INITIALIZATION_DONE procedure, and does not run to completion, no other program on the system can run (although programs already started continue to run).

RAISE_PROCESS_EXCEPTION Procedure

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

RESUME Procedure

The RESUME procedure resumes the execution of a suspended process. A resumed process is ready to run, but not necessarily running. If the process was waiting when it was suspended, the wait is repeated when it is resumed. Any 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.

SET_JOB_PRIORITY Procedure

This procedure sets the scheduling priority of the current job to an integer in the range 0–31. 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 LOAD_PROGRAM procedure; the default is 16.

SET_PROCESS_PRIORITY Procedure

This procedure sets the scheduling priority of the specified process to an integer in the range 0–15. 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 LOAD_PROGRAM procedure; the default is 8.

SET_USER Procedure

The 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 UIC to be associated with the process.

SIGNAL Procedure

A process can be signaled to quit with the SIGNAL procedure. The process must establish an exception handler for the exception KER$_QUIT_SIGNAL. If it does not handle the exception, it is forced to exit.

SUSPEND Procedure

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

WAIT_ANY and WAIT_ALL Procedures

The WAIT procedures are used to make a process wait for one to four objects. WAIT_ANY allows the invoking process to continue if any of the wait conditions 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.

Memory Management

VAXELN uses the VAX memory management hardware to map jobs in a virtual address space. Although knowledge of this subject is not essential, this section may be useful to you if you are already familiar with VAX memory management terminology.

Each job created by VAXELN executes a program image, a copy of all the code and initial data necessary to run the program. You build program images into the system image with the System Builder or load them dynamically with the program loader. The shareable run-time 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 containing all the program images, shareable run-time library images, and the kernel image into the System region of the VAX virtual address space. The System region maps the system image and kernel data, as shown in Figure 3-3.

When a job is created to run a program image, the kernel creates a P0 page table and maps the program image into the P0 region of the job's virtual address space. Each job's P0 region maps the program image, heap data, and message data, as shown in Figure 3-4.

The kernel also makes a copy of any read/write data in the program image, though no copy is made of read-only code and data. If there are multiple jobs in a system running the same program, there is only one copy of the read-only code and data, and as many copies of the read/write data, message data, and heap data as there are jobs running the program. Since the run-time library uses heap data for many of its data structures, the context of open Pascal and C files is also in P0.

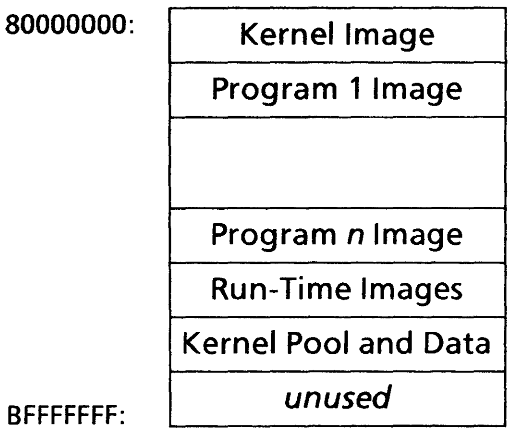


Figure 3-3. System Region

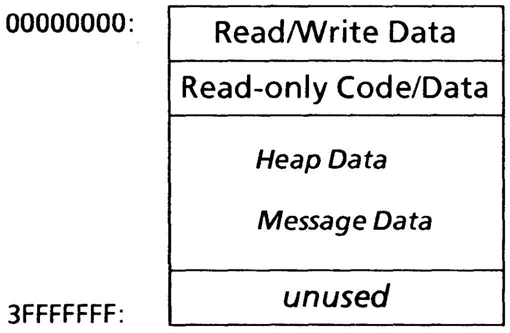


Figure 3-4. P0 (Program) Region

All processes in a job share the same P0 page table and, consequently, the same P0 region. This means that any data in the job's P0 region is accessible to other processes; assuming proper synchronization methods are used by the processes, a pointer to any data item in the P0 region can be passed to any process in the job. Note that 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.

For each process created for a job, the kernel allocates a P1 page table and maps the process's kernel mode and user mode stacks in the P1 region of the VAX virtual address space. These stacks are used by Pascal or C programs for all process-local variables (those declared inside a block) and procedure call frames. The P1 region does not map any area of the program image; it is exclusively for dynamic memory.

User mode processes have two stacks: a fixed kernel mode stack and a dynamically sized user mode stack. The VAXELN kernel automatically extends the size of the user mode stack as the process executes and demands a larger stack. The user mode stack grows downward in the P1 address space; its initial size is specified as a program attribute.

The kernel mode stack in a user mode process is two pages long. It 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. Overflowing the kernel mode stack causes an exception, named KER$_KERNEL_STACK. When this exception is delivered, the kernel stack pointer is reset to the top 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 also contains a page holding a process context block. This block is used by the VAXELN kernel, debugger, and run-time library routines to hold certain context information. The P1 region of the VAX virtual address space is shown in Figure 3-5.

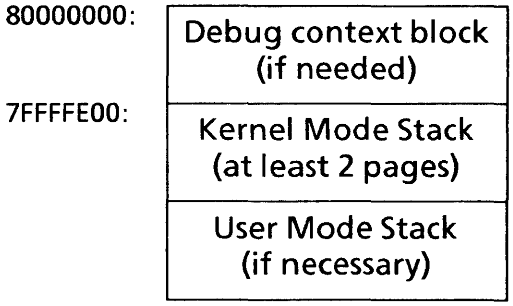

80000000:
7FFFFE00:
Debug context block (if needed)
Kernel Mode Stack (at least 2 pages)
User Mode Stack (if necessary)

Figure 3-5. P1 (Control) Region

Stack Utility Procedures

For most programs, the VAXELN stack management is exactly what is needed. There are two specific problem areas, however:

1. The stack usage of a process might vary widely during the execution of a process. When the process does not require a large stack, the previously allocated stack might be perceived as wasteful, since the kernel makes no attempt to automatically trim the stack. (Since it knows nothing about the behavior of the program, this is expected.) However, a program might want direct control over the extension and contraction of the stack.

2. For kernel mode programs, the stack size specified to the System Builder is allocated to each process in the job. Because each process may have widely different stack usage, this is again a potentially wasteful burden on the system's memory usage.

VAXELN provides utility procedures that can be used to explicitly manage the stack size during the execution of a program. The $STACK_UTILITY module provides the following procedures:

- ALLOCATE_STACK, which is called to verify the availability of a particular amount of stack space. If the stack space is not available, the procedure allocates the additional space needed. This procedure is most useful for a kernel mode process that demands more stack space than was allocated at sytem build time; it is not useful for a user mode process, since the kernel will automatically extend the stack as needed by the process.
- DEALLOCATE_STACK, which is called to trim the stack by up to the number of bytes specified. If the stack does not contain the specified space, the effect is to trim the stack back to the page in which the procedure is running. Therefore, specifying a large number causes the stack to be trimmed to the currently needed size.

Memory Allocation Procedures

The procedures summarized in this section are used for allocating and freeing memory.

ALLOCATE_MEMORY Procedure

The ALLOCATE_MEMORY procedure allocates physical memory pages into the 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 always done in units of memory pages (512-byte pages). The specified size is rounded up to page-sized units before the allocation. Allocation always begins on a page boundary.

When the kernel selects the allocation virtual address, it will be in the P0 or shared region of the job's virtual memory. The caller is free to specify any virtual address, so it is possible to allocate memory in the P1 or stack region as well as at a specific memory location in P0.

Most higher-level languages provide a more controlled way to allocate and free dynamic memory; for instance, the Pascal NEW procedure and the C **calloc** or **malloc** functions. ALLOCATE_MEMORY is intended to be used in the construction of these higher-level routines or by programs needing direct control of memory allocation.

The ALLOCATE_MEMORY procedure also allows a kernel mode caller to specify the exact physical address at which to start the allocation. This feature is intended for very specialized applications; for example, multiported memory or video bit-map 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 in use by some other part of the system.

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 all memory pages in the inclusive range from the base to the top.

Care should be used when freeing memory that was not explicitly allocated by the caller, since it is very difficult to determine the use of the virtual address range. For instance, deleting the process' stack can have unpredictable results.

Note that dynamically allocated memory is normally freed with the language-specific run-time library procedures provided; that is, the Pascal DISPOSE procedure and the C **free** or **cfree** functions. Any pointers to the freed memory become invalid.

MEMORY_SIZE Procedure

The 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 (all 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 very large MESSAGE or AREA objects, as these objects require contiguous memory for their data buffers.

While the MEMORY_SIZE procedure performs the memory scan, all other kernel operations are stopped; therefore, care should be taken to call this procedure only when necessary.

Interjob Data Sharing

The use of AREA objects provides a mechanism for interjob data sharing among jobs on a single node in a VAXELN network.

You can allocate and map a region of memory of a specified size into a job's virtual address space with the CREATE_AREA procedure. The area's memory is mapped into the creating job's P0 region at a virtual address specified by the caller, or if the caller does not care where the area is mapped, at an address selected by the kernel.

All the sharing jobs can map none, some, or all of the area's memory, depending on the size specified; however, each shares the memory region from its beginning. Associated with the memory is a binary semaphore, which can be used by the sharing jobs to synchronize access to the area's data.

AREA objects can be created, deleted, signaled, and waited on. When an AREA object is created, it is in a "signaled" state. Waiting on an area gives the waiter exclusive access to the area's memory. SIGNAL completes the exclusive access operation and allows other waiters to continue. Once an area is created, it remains available until all sharing jobs terminate or delete their references to the AREA object.

If a specific virtual address is specified by the caller, all sharing jobs will map the area's memory at the same virtual address. This feature makes the area *not* position independent; the sharing jobs can place real, fixed-pointer values in the region and they mean the same thing in each sharer's address space. The pointers must point to other addresses within the region if they are to be used by other jobs.

If a virtual address is not specified by the caller, the CREATE_AREA procedure will allocate a free P0 base address. Since the area could be in a different place in each sharer's space, fixed-pointer values cannot be used in the area. This is the typical case, with the area being used to hold one data structure.

Kernel Services for Interjob Data Sharing

The kernel services affecting the state of AREA objects are summarized below.

CREATE_AREA Procedure

The CREATE_AREA procedure creates a new area or maps an existing area of memory into the creating job's virtual address space, returning the AREA value that identifies the area. A string of up to 31 characters specifies the name of the area, which must be unique. A pointer variable receives a pointer to the beginning of the allocated memory.

An optional argument specifies the exact job virtual address, in the P0 region, at which the area is to be placed. If this address is specified by the creator, all sharing jobs will map the area's memory at the same virtual address. If the virtual address is not specified, the kernel allocates a free address for each job.

DELETE Procedure

The DELETE procedure removes a job's reference to the AREA object and unmaps the data from its address space. An area can be deleted by any process of a job that has created or mapped the area. The AREA object is actually deleted when the last referencer deletes its reference.

SIGNAL Procedure

When a referencing process is finished with its exclusive access to an area, the SIGNAL procedure allows the next waiting process to gain explicit access. It is an error to signal an area if the area is not "locked" by any process.

If the area is of zero length, the object represents a named interjob binary semaphore, in which case the semaphore count is incremented and tested. If the new count is greater than zero, the first waiting process in the semaphore's queue whose wait conditions can be satisfied is continued, and the count is decremented. If no processes are waiting, or if none of the waiting processes can continue, the count is not decremented.

WAIT–ANY and WAIT–ALL Procedures

A wait for an AREA object is satisfied when the object is signaled. Waiting for an area implies that the waiting process has exclusive access to the area until a complementary signal is sent. If the area is of zero length, the object represents a named interjob binary semaphore, in which case the semaphore count is decremented if the wait is satisfied by signaling the semaphore.

Chapter 4
Synchronization

Introduction

To synchronize any two things is to arrange them in such a way that they *appear* to occur simultaneously. For instance, a motion picture soundtrack is "synchronized" with the film if, at a normal viewing distance, the sound seems to correspond to what is happening on the screen. In real-time programming, four broad classes of synchronization questions occur:

- How to synchronize processes with each other; that is, how to make a particular set of processes ready to execute at the same time.
- How to synchronize a process with some phenomenon external to the software; for example, how to synchronize a process with interrupts from a particular device.
- How to *prevent* processes from performing some operations simultaneously (or simply in an unpredictable way); for example, how to prevent two processes from using shared data in an unpredictable way.
- How to sequence or serialize the execution of processes; that is, how to make a particular set of processes execute in a continuous, ordered series.

This chapter discusses synchronization in terms of the WAIT procedures, events, semaphores, and time representation, including the kernel services relating to synchronization objects.

The WAIT Procedures

Basically, dealing with synchronization requires that you have: first, a method of making a process wait for something; second, a definition of the things processes can wait for; and third, a definition of the way processes are allowed to proceed after they have waited.

The WAIT procedures provide the method for processes to wait and also provide the definition of what allows the processes to proceed again. The data types DEVICE, EVENT, SEMAPHORE, PORT, PROCESS, and AREA provide the definition of what processes can wait for. The two WAIT procedures are:

- WAIT_ANY accepts a list of up to four DEVICE, EVENT, SEMAPHORE, PORT, PROCESS, or AREA values (including mixtures). The calling process waits until *any one* of these objects allows it to continue (that is, to return to the Ready state). An optional result parameter receives the argument number of the argument that satisfied the wait. You can also supply 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.
- WAIT_ALL accepts the same arguments as WAIT_ANY, but it allows the process to proceed only if *all the conditions are satisfied simultaneously*. (Again, the timeout argument can allow the process to proceed regardless of the states of the objects.)

As for the conditions:

- Waiting for a port means waiting for a message to arrive at that port.

- Waiting for a PROCESS value means waiting for the identified process to terminate.
- Waiting for a DEVICE value means waiting for the connected interrupt to be signaled by an interrupt service procedure.
- Waiting for a synchronization object (EVENT or SEMAPHORE) or an AREA object means waiting for the object to be signaled.

Both procedures have the same effect on their arguments:

- Satisfying a wait on a semaphore causes the semaphore count to be decremented. At most, one process continues as the result of a semaphore's being signaled.
- Waiting for an event, port, or process causes no modification to the object, and all waiting processes continue if their wait conditions are otherwise satisfied.
- Waiting for a device causes the object to be cleared if the wait is satisfied by an interrupt service routine signaling the object. That is, only one process continues as a result of the action of an interrupt service routine.
- Waiting for an AREA object implies that the waiting process has exclusive access to the area until a complementary signal is sent. When a referencing job's main process is deleted, a check is made; if the process being deleted is the owner process, the area is implicitly signaled. If the process being deleted is the last referencer, the area is deleted.

The WAIT procedures return immediately if one of the argument objects does not exist or is deleted. Both procedures also return immediately if the necessary conditions were satisfied already (before the call). That is, the elapsed time is only the time required to perform a procedure call, and any specified timeout value is irrelevant.

Deadlock Prevention

WAIT_ALL waits for a number of conditions to be *simultaneously* satisfied; therefore, there is no potential *deadlock*, as occurs in some systems having similar features. Briefly, deadlock occurs when two or more processes wait for the same set of resources, "holding" them in some way as they become available. Since WAIT_ALL will not hold some resources while waiting for others to become available, deadlock is not a problem if 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 than multiple calls to WAIT_ANY.

Events

An EVENT object represents the occurrence of an event and stores that information until explicitly cleared by a program. A process asserts, or "signals" the occurrence of the event with the SIGNAL procedure. Other processes can wait for the signaling of the event with the WAIT_ANY and WAIT_ALL procedures. If the event is cleared at that point, the processes will wait until the event is signaled. When an EVENT object is signaled, *any and all processes waiting for it become Ready* (that is, if their wait conditions are otherwise satisfied).

The CREATE_EVENT procedure can initialize the EVENT value to either CLEARED or SIGNALED. EVENT values are valid only within the job in which they are created.

The practical meaning of an EVENT object (that is, the particular real-time event that it represents) is determined by the programmer. The conditions under which the EVENT object is signaled define its relationship to a real-time, real-world event. Literally, though, the EVENT object has only the properties "signaled" and "cleared." The point is that there is nothing intrinsic in the EVENT object that determines which process can signal it or what the signal means to waiting processes. Rather, you must ensure that signaling and awaiting an event occur in a manner consistent with your application. For example, in Pascal,

```
VAR
   LIGHTS_ON: EVENT;
BEGIN
   CREATE_EVENT(LIGHTS_ON,EVENT$CLEARED);
   .
   .
   SIGNAL(LIGHTS_ON);
```

creates a new EVENT object, assigns an identifier for it to LIGHTS_ON, and (after determining that the lights are on) signals any processes that may be waiting for that event.

The satisfaction of a wait has no effect on the properties of an EVENT object. In other words, once an EVENT object is signaled, it remains signaled until it is cleared explicitly by the CLEAR_EVENT procedure. Meanwhile, any processes that wait for the event will have that part of their wait conditions satisfied immediately. (This is in contrast to AREA, SEMAPHORE, and DEVICE objects, whose properties are changed both by signaling *and* by the satisfaction of a wait.)

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. There is one master process that controls the sequencing of several other worker subprocesses. Each subprocess executes independently until it completes a particular 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). It 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 decides it is time to perform the next control step, it 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.

Semaphores

A semaphore acts as a gate, controlling access to a resource. It maintains a count of the currently available units of the resource, such as, perhaps, the number of disk drives available, the number of gates available at an airport, or, in the case of an actual railroad semaphore, the "number of tracks" (0 or 1) available to a train going in a particular direction. You set the initial and maximum counts when you create the semaphore. The semaphore count interacts with the WAIT and SIGNAL procedures to control the execution of processes that wait for it.

Unlike events, *semaphores are changed by the action of the WAIT procedures*. Specifically:

- When a semaphore is signaled with SIGNAL, the count is incremented. Exceeding the maximum count is an error.
- When a process waits for the semaphore with WAIT_ANY or WAIT_ALL, it keeps waiting until the count is greater than zero. When the count exceeds zero (and, in the case of WAIT_ALL, all other wait conditions are satisfied), the WAIT procedure returns and the process can proceed. Furthermore, when the wait is satisfied as a result of signaling the semaphore, the WAIT procedures decrement the semaphore count.

Together, these operations on the count mean that *at most one process becomes Ready when a semaphore is signaled.*

A process that wants to use the controlled resource waits for the semaphore. If the semaphore count is greater than zero, it is decremented, and the process becomes Ready. If the count is zero, the process waits until some other process signals the semaphore. If several processes wait for the same semaphore, they are queued in the order in which their WAIT procedures were executed.

A process signals a semaphore when it no longer requires the resource; SIGNAL increments the semaphore count and the first waiting process, if any, is allowed to proceed. (This is in contrast to events, with which *all* processes waiting for the same event can continue as a result of a single signal.) Thus, the maximum count of a semaphore represents the *available units* of the resource being controlled.

Depending on your choice of the maximum count, semaphores have two uses which, although related, are different enough that different techniques and terminology apply:

- A semaphore with a maximum count of 1 is used to guard a single item (often, a shared variable) from access by more than one process. This usage is more closely related to the railroad metaphor. This kind is called a *binary semaphore*. Here, the semaphore is used like a gate which allows only one process at a time to "get through" to the resource behind it. Signaling a binary semaphore opens the gate for one process, which closes the gate behind itself.
- A semaphore with a maximum count greater than 1 is called a *counting semaphore* in this manual. Conceptually, a counting semaphore can be viewed either as a gate that can allow *more than one* process through, or as a *meter* that keeps a count of the currently available units of some finite resource.

In both cases, the initial count simply determines the action of processes that wait for the semaphore before it is signaled by anyone.

A SEMAPHORE value is valid (that is, identifies the same semaphore) everywhere in the job that creates it. Multiple processes in the job can use the semaphore by sharing a variable or by passing the SEMAPHORE value as a process argument.

For example, in Pascal,

```
VAR
   UNIT–AVAILABLE: SEMAPHORE;
BEGIN
   CREATE–SEMAPHORE(UNIT–AVAILABLE,10,10);
```

creates a semaphore representing currently available units of a type of disk drive and puts the SEMAPHORE value in the variable UNIT–AVAILABLE. There are a total of 10 disk drives; initially, all 10 are available.

Now, other processes needing disk drives wait for the semaphore UNIT–AVAILABLE with, say, WAIT_ANY. Because the initial count is 10, the first 10 processes to wait will continue immediately, and each time one continues, the count is decremented. The eleventh and subsequent processes (assuming that the first 10 have not yet released their drives) will remain in the Waiting state, because the count is now zero.

When each process is through using its drive, it "returns" it to the pool of available drives by signaling UNIT–AVAILABLE. The next process that waits (or the first one that is already waiting) will continue, because the count is now above zero again. The semaphore is *metering* the disk drives available because, at any time, its count is the number of drives not in use.

The following scenario describes 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 a maximum and initial counts of 1.

- The master process then creates worker subprocesses, as the need arises, to handle incoming data base inquiries.
- Each subprocess waits for 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 will be allowed access to the data base at a time. All others will wait until the current worker signals the semaphore. When the semaphore is signaled, only the next process is "let through the gate" until it, too, signals the semaphore.

Another term used to represent a binary semaphore in VAXELN is *mutex*, which is an abbreviation for "mutually exclusive" semaphore. Mutexes are used in VAXELN Pascal and C as a way to achieve the same effect as a binary semaphore without calling the kernel service unless contention occurs.

Kernel Services for Synchronization Objects

The kernel services affecting the state of EVENT objects and SEMAPHORE objects are summarized below.

CLEAR_EVENT Procedure

The CLEAR_EVENT procedure sets the state of the EVENT object to "cleared."

CREATE_EVENT Procedure

The CREATE_EVENT procedure creates and initializes an event with an initial state of "signaled" or "cleared," returning the EVENT value that identifies the event.

CREATE_SEMAPHORE Procedure

The CREATE_SEMAPHORE procedure creates and initializes a semaphore with an initial count and a maximum count supplied by integer expressions, returning the SEMAPHORE value that identifies the semaphore. The initial count must not exceed the maximum count; signaling the semaphore beyond this count is an error.

DELETE Procedure

The DELETE procedure removes the EVENT or SEMAPHORE object from the system. When an event or semaphore is deleted, any waiting processes are removed from their wait states immediately, with the completion status KER$_BAD_VALUE.

SIGNAL Procedure

SIGNAL sets the state of an event to "signaled" and continues all waiting processes whose wait conditions can be satisfied.

Signaling a semaphore increments and then tests the semaphore count. If the new count is greater than zero, the first waiting process in the semaphore's queue whose wait conditions can be satisfied is continued, and the count is decremented. If no processes are waiting, or if none of the waiting processes can continue, the count is not decremented. At most, one process continues as a result of signaling a semaphore.

WAIT_ANY and WAIT_ALL Procedures

A wait for an event is satisfied when the object is signaled. Waiting for an event causes no modification to the object, and all waiting processes continue if their wait conditions are otherwise satisfied.

A wait for a semaphore is satisfied when the object is signaled. Waiting for a semaphore causes the sema-phore count to be decremented if the wait is satisfied by signaling the semaphore.

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.

Time Representation

The VAXELN kernel maintains a system time as a 64-bit binary number. The system time is interpreted as the number of 100-nanosecond intervals since the base time, hour 00:00:00.00, November 17, 1858. Because it is maintained by the kernel, using the processor's interval timer, the system time is in effect for all jobs running on that processor.

Note that the time representation in VAXELN is identical to the representation of time in VAX/VMS.

SET_TIME and GET_TIME Procedures

You can set the system time for a processor with the SET_TIME procedure and can obtain it at any point with the GET_TIME procedure. SET_TIME sets a new system time. GET_TIME returns the current system time.

The system time is not necessarily preserved across power failures and is not set to any default value by the kernel or other system software. Therefore, you should initialize the system time in an initialization job (see Chapter 13, "System Development") and in a handler for the KER$_POWER_SIGNAL exception.

You can also set and display the system time with the debugger commands SET TIME and SHOW TIME (see Chapter 15, "Debugging").

Timeout in WAIT Procedures

With the WAIT_ANY and WAIT_ALL procedures, you can specify a timeout argument, meaning that the calling process stops waiting either at a specific date and time (an *absolute time*) or after a specified interval relative to the current system time. Both time intervals and absolute time are expressed as signed, 64-bit time values.

The VAXELN Pascal and C language run-time libraries provide routines for dealing with time values conveniently; for example, converting a time value to an ASCII string for printing.

By convention, negative time values represent time intervals; nonnegative values are absolute times. The system time is nonnegative and is therefore an absolute time.

Chapter 5
Interjob Communication

Introduction

In a VAXELN application, every job has a unique and protected virtual address space. Within a single processor, the kernel separates each job's virtual address space using the VAX memory management hardware, as explained in Chapter 3, "Processes and Jobs." Within a network, each job's virtual address space is separated by virtue of the fact that the jobs may exist in the memories of different computer systems.

One of the principal reasons for dividing an application into separate jobs is to aid the migration and distribution of the jobs within the network. In order for the jobs to work together on the same application, they must be able to share data, but the only way of moving data from the memory of one processor to another in a network is by packaging the data in a message and directing the network hardware to move the message to the destination memory.

To make the network movement of data between jobs the same as in the nonnetwork case, message passing is the principal means of interjob communication provided by VAXELN. The kernel provides a number of facilities to make messages an efficient and transparent means of communication. For those cases where memory sharing between jobs is a necessity and where the network distribution of the jobs is not an issue, the kernel supports the use of AREA objects, as explained in Chapter 3.

It should be noted that message passing, as a communication mechanism, provides more than just the movement of data. Since the act of sending a message is an asynchronous real-time event, message passing can also be used to synchronize and coordinate multiple processes and jobs. In fact, most of the VAXELN services use message passing to organize their work, making them "message-driven."

This chapter discusses interjob communication in terms of messages and ports, including message transmission, datagrams and circuits, and the kernel services relating to message transmission.

Messages

A "message" as recognized by most network devices is a block of contiguous bytes of memory. Usually, network devices, particularly Ethernet devices, also impose a maximum size on a message. A network message also typically requires some 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 associated data are both created by calling the CREATE_MESSAGE procedure.

Because message passing is a key principle in VAXELN programming, the kernel was designed to make message operations very efficient. Message data is allocated by the kernel from physically contiguous, page-aligned blocks of memory; this allows the kernel to store the complete description of a message of any 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.

If a message is sent to a job on the sending job's local node, the kernel does not copy the message data. Instead, the kernel unmaps the data from the sending job's virtual address space and remaps it into the receiver's space.

If a message is sent to a remote node, the kernel again unmaps the data and uses an appropriate network device to send the message to the remote system, where the reverse operations cause the message to be finally remapped in the receiver's space.

Message Ports

A PORT object represents a system-maintained message queue. A port is unique in that its identifying value is valid anywhere in the application (including on other network nodes), not just within a particular job. 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. Messages are removed from a port by the RECEIVE procedure, in first-in/first-out order. If more messages than the maximum are sent, they are lost. (However, see "Port Limits and Flow Control," later in this chapter.) Note that there is no intrinsic overhead associated with the message limit itself; that is, 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, in Pascal,

```
VAR
   newport: PORT;
BEGIN
   CREATE-PORT(newport, LIMIT := 10);
```

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 subsequent SEND, RECEIVE, and other message operations that require a PORT value.

Named Message Ports

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

Any process in the application that expects to communicate with others outside its job can "broadcast" the necessary information about one or more of its message ports by creating names for them. If it may need to communicate with any process on any network node, it creates a *universal* name; if all communication occurs within a single node, a *local* name suffices.

These names are created with the CREATE_NAME procedure and can be deleted with DELETE. A NAME value specifies a 1- to 31-character-string name for an associated port. The name string is used for obtaining the actual 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.

A common use of such names is by application-wide services such as a disk. The disk driver makes available certain names for its message ports (for example, "DUA0") so that any job or process can communicate with it easily. That is, any process can quickly translate the name into a PORT value for use in sending messages. Note that in the specific case of a disk, program I/O is typically done with language-specific I/O procedures, whose run-time 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 are also used directly in some cases (for example, as a parameter to the CONNECT_CIRCUIT procedure), in which case the translation is done by the procedure.

Names can be either local to a node or universal. A local name is guaranteed to be unique within the local node. Universal names are guaranteed to be unique throughout the entire application. The translation and other maintenance of universal names is a function of the Network Service, as described in Chapter 7, "The Network Service."

Message Transmission

To send a message, you declare a pointer to the type of data you want to send, supply the pointer to CREATE_MESSAGE, use the pointer to fill in the message data, and supply the MESSAGE value to the SEND procedure. For example, in Pascal:

```
VAR  mtext: ↑VARYING-STRING(80);
  command: MESSAGE;
  destination: PORT;
  .
  .
  BEGIN
  CREATE-MESSAGE(command,mtext);
  mtext↑ := 'START';
  .
  .
  SEND(command,destination);
  END.
```

Note that in Pascal, the size of the message is implied by the pointer, while in other languages (in particular, the C language) the size is given explicitly to the SEND procedure.

The SEND procedure removes the message data from your job's address space and places the MESSAGE object in the destination port. It 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.

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. (Possibly, the receiver uses the same message data to form the reply; it need not create a new message.)

Note that the receiver process must know beforehand the formats of all 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 of some kind, it must know a set of predefined commands to which it will respond; it can return an error message to the sender (or, more likely, to an operator's terminal) if it receives a message that does not contain a valid command.

Expedited Messages

A distinct form of message, called an *expedited message*, is recognized by the kernel and the Network Service. An expedited message can be used to 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 will not need to use expedited data messages, particularly because they are very restrictive and there is no guarantee that an expedited message will actually 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 existing protocols such as the Data Access Protocol.

The following facilities and restrictions are related to expedited data messages:

- An expedited data message is sent by specifying a Boolean value to the EXPEDITE parameter of the SEND procedure.
- The size of an expedited data message must be 16 bytes or less.
- Only one unreceived expedited message is allowed in a port when the port's maximum message count is at its limit. If a second message is sent before the first is received, it has the same effect as a normal data message; 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 an alternate success, KER$_EXPEDITED. Therefore, if a receiver process needs to know if a message is an expedited or a normal message, and the protocol being used does not indicate which it is, the receiving process can simply compare the status to KER$_EXPEDITED.
- Any expedited data messages queued to a port are received by the RECEIVE procedure before any normal data messages are received.

Datagrams and Circuits

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

- One process can obtain the value of a port anywhere in the system (including in other jobs) or in a different system running on a different Ethernet node. The process can send the port a message with the SEND procedure. This is called the *datagram* method.
- Any two ports (usually in different jobs) can be bound into pairs called *circuits*. In this case, having established 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. In this case, the sender sends the message to its own port, and it is routed automatically to the other port in the circuit. Processes can both send to and receive from a circuit port.

In the datagram method, as well as in the circuit method, a process can use the WAIT procedures to wait for the receipt of a message on the specified port.

The datagram method requires no connection sequence like circuits, but cannot guarantee that a message is actually received at the destination. (However, it does guarantee with high probability that received messages are correct.) In addition, 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.

At the cost of setting up and handling a circuit connection, circuits offer several 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 either the message arrives at the destination port regardless of its location, or 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 of the proper length, transmit the

segments in sequence, and reassemble them at the destination node.

- A user interface via 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.

There is no performance penalty with circuits for messages transmitted on the same network node and very little over the network. For full generality, programs should assume that the sending and receiving jobs may be distributed on different nodes in a network. Circuits are the preferred means of sending messages in almost every practical case.

Programming with Circuits

Circuits are established between two ports by the CONNECT_CIRCUIT and ACCEPT_CIRCUIT procedures. A process that wants to establish a circuit calls CONNECT_CIRCUIT and designates a destination port in another process. A special connection-request message is automatically sent to the designated port. For example, in Pascal:

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

Here, myport is a port in the calling process that will form its half of the circuit. The destination is specified by the NAME string 'request-server', which is translated automatically by CONNECT_CIRCUIT to designate the destination port.

The call to CONNECT_CIRCUIT causes a process to wait for the connection request to be accepted. The maximum time that is allowed to lapse before a circuit connection must be accepted is specified by the *Connect* *time* characteristic on the *System Characteristics* System Builder menu.

Elsewhere, an ACCEPT_CIRCUIT call causes a process to wait for a connection-request message on the designated port:

```
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);
```

At this point, the acceptor can take a variety of actions to communicate with the requestor, such as creating a subprocess to continue the dialog and passing it the PORT value (receiver–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 still pending.

Circuits are broken when either partner calls the DISCONNECT_CIRCUIT procedure. The SEND and RECEIVE procedures both notify their callers if the designated port was disconnected.

Note: If more than one process in a job is waiting for a message on a circuit port, confusion is possible. For example, some process may use WAIT_ANY with the port, and when it is released from the wait and calls RECEIVE, there is no message to be received. The reason may be that the message was a connection request and was removed from the port by some other process in the job that called ACCEPT_CIRCUIT.

The RECEIVE procedure has a "no message" status to help detect this mistake; however, the correct practice, strictly speaking, is to avoid having multiple processes wait for a port that will receive connection requests. Instead, use one port in the job for this purpose; the acceptor process can accept the requests on that port and form the actual circuits with other ports in the job. A typical example of this is when a process waits for circuit requests on its job port but establishes new circuits on other ports, thus leaving the job port free for more requests.

Port Limits and Flow Control

One of the advantages of using a circuit for a message exchange is that the kernel and Network Service provide a function called *flow control*. The basic idea behind flow control is that the flow of messages from the transmitting process to the 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 the specified destination port. If the transmitting process can produce messages faster than the receiving process can consume them, and if there is no limit on the number of messages that can be queued, the messages might potentially use all the available memory. For this reason, ports have a limit on the number of unreceived messages that can be queued at any one time; the limit is specified when the port is created.

Flow Control with Unconnected Ports

If a port contains its limit of messages (that is, it is "full") and is not connected in a circuit, and a SEND is attempted to the port, the SEND returns a failure status (or exception). (If the port is not on the same node, the message could be lost.)

Flow Control with Circuits

If a port is connected in a circuit and 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 this 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 wait.

Kernel Services for Message Transmission

The kernel services affecting the state of MESSAGE, PORT, and NAME objects are summarized below.

ACCEPT_CIRCUIT Procedure

The ACCEPT_CIRCUIT procedure causes the invoking process to wait for a circuit connection. When the wait is satisfied (that is, on successful completion), the circuit is established between two ports.

The invoker's half of the circuit can be either 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 actual connection; in this way, the server could create a name for the first port to establish simultaneous circuits with several different processes or jobs. Note that the only valid message that can be received at the connection-request port is the kernel's internel connection request; all other messages are discarded by the system.

By default, when a process sends a message on a circuit (with SEND), it 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 in length.

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 together in a circuit. The destination port can be specified either via a name string established by the CREATE_NAME procedure, or by 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, if the partner already contains its limit of unreceived messages); this is the type of *flow control* 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 any data supplied by the accepting process in its ACCEPT_CIRCUIT call.

Note: The maximum time that is allowed to lapse before a circuit connection must be accepted is specified by the *Connect time* characteristic on the *System Characteristics* System Builder menu.

CREATE_MESSAGE Procedure

The CREATE_MESSAGE procedure creates a MESSAGE object and allocates and maps its associated 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.

CREATE_NAME Procedure

The CREATE_NAME procedure creates a 1- to 31-character-string name for a specified port as an entry in a name table, returning the NAME value that identifies the name. An optional argument specifies that the new name is either local (valid only in this system, or node), universal (valid throughout the application, or on any node), or both; local is the default. If the system does not contain the Network Service, all names are placed in the local name table.

Names created by this procedure are guaranteed to be unique within the specified name space (local or universal). If you attempt to create a name that is not unique, no NAME object is created, and an error status is returned.

When a universal name is created in a network configuration, the Network Service on each node in the local area network makes the name available to the system running on that node.

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.

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 any pointers to the message data become invalid.

When a port is deleted, any connected port (when the deleted port is in a circuit) is disconnected, any 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.

DISCONNECT_CIRCUIT Procedure

The DISCONNECT_CIRCUIT procedure is used to break the circuit connection between two ports. If any 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.

JOB_PORT Procedure

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

RECEIVE Procedure

The RECEIVE procedure removes a message from the designated 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 (normally the same value supplied by the sender for the receiver's port).

An optional integer argument receives the size in bytes of the message data (this argument is only optional for Pascal).

SEND Procedure

The SEND procedure removes the message data from the sender's address space and queues the MESSAGE value identifying the message in the message port supplied by the PORT value identifying the destination. If the message is being sent through a circuit, this port 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.

Optional arguments specify the length in bytes of the message data to be sent (this argument is only optional for Pascal), a PORT value identifying a reply port, and whether to expedite the message. The size of an expedited message must not exceed 16 bytes.

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 name's associated 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. If the Network Service is not present in the system, any attempts to translate universal names will return the status "no such name," since, in effect, there is no universal name table without the Network Service. (See Chapter 7, "The Network Service," for more information.)

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 either 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 some other object, it can call a WAIT procedure specifying the port. When the wait is satisfied, it can call ACCEPT_CIRCUIT.

Chapter 6
I/O Devices and Interrupt Handling

This chapter discusses the handling of device interrupts, interrupt priority levels and procedures to manipulate them, recovery from power failure, and the kernel services relating to devices. The procedures relating to direct memory access UNIBUS and QBUS device handling are then summarized, followed by device register procedures.

Handling Device Interrupts

Interrupt service routines are procedures used to handle device interrupts and power recovery in programs that control devices. With the CREATE_DEVICE procedure, you can connect a device interrupt to an interrupt service routine. The CREATE_DEVICE procedure creates 1 to 16 DEVICE objects that can be used as semaphores which are signaled by the interrupt service routine and awaited by a process.

When connected to a device interrupt, an interrupt service routine is called by the kernel directly on the occurrence of each interrupt and can take any actions necessary to service the interrupt. The interrupt service routine can communicate with a program through an area of memory called the *communication region* established by CREATE_DEVICE. For example, the interrupt service routine can use this region to give the program a value it has obtained from a device register. Only data placed in the communication region is available to an interrupt service routine.

All communication regions are potentially accessible to all interrupt service routines. 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 YCDRIVER.PAS, delivered with the development system.)

To synchronize itself with processes in the job that created the object, the interrupt service routine can signal any of its DEVICE objects with the SIGNAL_DEVICE procedure.

The processes can use the WAIT_ANY and WAIT_ALL procedures as follows:

- When a process waits for the DEVICE object with WAIT_ANY or WAIT_ALL, it keeps waiting until the interrupt service routine signals the object with SIGNAL_DEVICE.
- When the object is signaled, the waiting process continues.

If several processes wait for the same device, they are queued in the order in which their WAIT procedures were executed. When the DEVICE object is signaled, the signal allows *at most* one process to continue, the first process in the queue.

Viewed as synchronization objects, therefore, DEVICE objects are similar to binary semaphores. That is, only one process continues as a result of a SIGNAL_DEVICE call. The interrupt service routine can call SIGNAL_DEVICE for any of its 1 to 16 DEVICE objects.

A DEVICE object is also associated transparently (that is, by the kernel) with a description of the physical device. Device descriptions consist of a character-string name for the device, its bus-request priority, and the addresses of the device's interrupt vector and control/status register. They are entered in the system, once, using the System Builder, and are then used transparently by programs via the DEVICE object.

For the purpose of writing device drivers for multiple units on a single controller, the CREATE_DEVICE procedure also allows you to create up to 16 DEVICE objects that can be signaled by an interrupt service routine.

Interrupt Priority Levels

The VAX processor defines 32 levels of interrupt priority levels (IPLs). IPL 0 is the lowest priority; IPL 31 is the highest. Table 6-1 lists the interrupt priority levels at which various system events occur.

Setting the processor IPL allows a process to synchronize itself with an interrupt service routine. This 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 very efficient.

Raising and lowering the interrupt priority level of the processor is achieved with the DISABLE_INTERRUPT and ENABLE_INTERRUPT procedures.

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	Queue asynchronous exception
5	Kernel debugger
4	Job scheduler
3	Process scheduler
2	Deliver asynchronous exception
1	Unused
0	User process level

Note: The current interrupt priority level is part of the processor-wide state of a VAX computer. Disabling interrupts of a certain priority also disables all other system activities that occur at or below that priority level. In essence, if the IPL is raised by a process to block device interrupts, that process is the only activity, other than interrupt service routines, that can execute until the process lowers the IPL by calling ENABLE_INTERRUPT.

IPL Procedures

The procedures summarized in this section are used to raise or lower interrupt priority levels.

DISABLE_INTERRUPT Procedure

DISABLE_INTERRUPT prevents interrupts from a device, by raising the IPL of the processor to the IPL of the device. While interrupts are disabled, no kernel procedures can be called; attempting to do so causes unpredictable results. It can be used only by programs running in kernel mode.

If a program has powerfail exceptions enabled and the power fails while interrupts are disabled, the IPL is set to zero before the KER$POWER_SIGNAL exception is raised. This exception is handled like any other synchronous exception (see Chapter 11, "Exception Handling"), but continuing from the exception if it occurs with interrupts disabled has unpredictable effects.

ENABLE_INTERRUPT Procedure

ENABLE_INTERRUPT allows interrupts from a device by lowering the IPL of the processor to minimum priority (0). It can be used only by programs running in kernel mode.

Power-Recovery Handling

Devices normally need special attention following a power failure, and the necessary speed and synchronization requirements cannot be met by the general power-recovery exception (KER$_POWER_SIGNAL). Therefore, you can specify, in a CREATE_DEVICE call, the name of an interrupt service routine that is called when the processor enters its power-recovery sequence. Such a routine is called before any other process or ordinary interrupt service routine is restarted. Typically, the operations that must be performed on power recovery are as follows:

1. Reinitialize the device controller to a known state.
2. Assure that no partially completed I/O operations are started, since the device has been reinitialized.
3. Signal any processes that are waiting for device interrupts, since none will occur now that the device has been reinitialized.

All three operations can be performed by a power-recovery routine. Since power recovery occurs at unpredictable times, the interrupt service routine and main program must synchronize themselves with the action of the power-recovery routine to retry any 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 it to synchronize itself with the power-recovery routine. Once a power-failure interrupt has been posted, the processor has only about 4 milliseconds before power is shut down, so the interrupt should not be disabled for more than a few instructions.

Kernel Services for Devices

The kernel services affecting the state of DEVICE objects are summarized below.

CREATE_DEVICE Procedure

The CREATE_DEVICE procedure establishes a connection between a physical device, a program, and an interrupt service routine. It creates one or more DEVICE objects, which are used to synchronize the program with the device. CREATE_DEVICE can be called only from a program running in kernel mode.

You specify the actual device and its characteristics to the System Builder when you create the system. Among other things, you assign a 1- to 30-character string as the device's name. This name is specified to the CREATE_DEVICE procedure to retrieve the device's characteristics.

If the device has multiple units (such as a disk controller with two or more drives), CREATE_DEVICE can create an array of DEVICE objects, and the interrupt service routine can signal any object to identify the interrupting unit. Meanwhile, the program can dedicate a subprocess to each device unit.

An optional argument specifies which vector of a multiple-interrupt-vector device should be connected to the interrupt service routine. If it is omitted, the default is 1 (first vector).

Optional arguments also return pointers to the first device control register, the first adapter control register, and the interrupt vector in the system control block.

Other optional arguments return the interrupt priority level (IPL) of the device and the name of a power-recovery routine that is called before any process or interrupt service routine is restarted, if the processor enters a power-recovery sequence.

DELETE Procedure

The DELETE procedure removes the DEVICE object from the system. When a DEVICE object is deleted, the memory used for its communication region is deleted, and any pointers to that memory thus become invalid. The interrupt service routine is disconnected from the interrupt vector. Any waiting processes are removed from their wait states immediately, with the completion status KER$_BAD_VALUE.

SIGNAL_DEVICE Procedure

The SIGNAL_DEVICE procedure signals a DEVICE object from an interrupt service routine. It can be called only from an interrupt service routine or a subroutine thereof. An optional argument identifies the element in a DEVICE array to be signaled.

WAIT_ANY and WAIT_ALL Procedures

A wait for a DEVICE object is satisfied when the state of the object is "signaled" (the result of the SIGNAL_DEVICE procedure, called from an interrupt service routine). Waiting for a device causes the DEVICE object to be cleared if the wait is satisfied by the DEVICE object; that is, only one process continues as a result of the action of an interrupt service routine.

DMA Device Handling Procedures

The procedures summarized in this section are used in programs that control direct memory access (DMA) UNIBUS and QBUS devices.

ALLOCATE_MAP Procedure

The ALLOCATE_MAP procedure allocates a contiguous block of UNIBUS or QBUS map registers for use by a program to map VAX memory to UNIBUS or QBUS memory addresses, respectively. It can be called only from programs running in kernel mode.

The procedure returns a pointer to the first register allocated and returns the starting map register number (0–495). Optionally, it 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.

ALLOCATE_PATH Procedure

The ALLOCATE_PATH procedure allocates a UNIBUS adapter buffered datapath for use by a direct memory access UNIBUS device. It can be called only from programs running in kernel mode.

The procedure returns a pointer to the allocated datapath register and the allocated datapath register number (1–3). An argument supplies the DEVICE value that identifies the device for which the datapath is allocated.

A buffered datapath can be used to optimize the use of memory by a DMA device that does strictly sequential address transfers. (For additional information on buffered datapaths, see the *VAX Hardware Handbook*.)

The VAX-11/750 is the only processor supported by VAXELN that has UNIBUS buffered datapaths.

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

When a UNIBUS buffered datapath is used for a DMA transfer, the datapath must be "purged" when the transfer has completed. This is accomplished by writing a value of 1 to the datapath register, identified by the returned register pointer.

FREE_MAP Procedure

The FREE_MAP procedure frees a set of previously allocated UNIBUS or QBUS map registers. It can be called only from a program running in kernel mode. Any 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 ALLOCATE_MAP), and the DEVICE value that identifies the device for which the registers are freed.

FREE_PATH Procedure

The FREE_PATH procedure frees a previously allocated UNIBUS adapter buffered datapath. It can be called only from programs running in kernel mode. The VAX-11/750 is the only processor supported by VAXELN that has UNIBUS buffered datapaths. Arguments supply the datapath register number (such as the one returned by ALLOCATE_PATH) and the DEVICE value that identifies the device for which the datapath is freed.

LOAD_UNIBUS_MAP Procedure

The LOAD_UNIBUS_MAP procedure is used in device driver programs to load UNIBUS or QBUS map registers for use by a direct memory access UNIBUS or QBUS device, respectively. This is an alternate procedure to the more commonly used UNIBUS_MAP procedure.

The procedure assumes that sufficient map registers have been allocated by the calling program using the ALLOCATE_MAP procedure (UNIBUS_MAP allocates them for the caller). It also assumes that one 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 QBUS map register allocated by ALLOCATE_MAP, the I/O buffer, and the buffer size. An optional argument is a pointer to the System Page Table (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 datapath to be used for the transfer. If this argument is not supplied, datapath 0, the direct datapath, is used.

PHYSICAL_ADDRESS Function

The PHYSICAL_ADDRESS function is used for DMA devices on the MicroVAX, returning the physical address of an identified variable. Programs using this function must include the module $PHYSICAL_ADDRESS.

UNIBUS_MAP Procedure

The UNIBUS_MAP procedure is used in device driver programs to map memory buffers for direct memory access by UNIBUS or QBUS devices. That is, the specified buffer is mapped into the UNIBUS or QBUS address space, and the procedure returns the 18-bit UNIBUS address or the 22-bit QBUS 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 adapter datapath to use; the default is 0, specifying the unbuffered data path.

Note: The procedure allocates the correct number of map registers by calling ALLOCATE_MAP. It 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 datapath other than 0 is specified, it is stored in the map registers as well. Although the map registers are allocated by UNIBUS_MAP before use, a nonzero datapath number is assumed to be unused by any other device.

UNIBUS_UNMAP Procedure

The UNIBUS_UNMAP procedure is used in device driver programs to unmap memory buffers previously mapped for direct memory access by a UNIBUS or QBUS device. The procedure deallocates the correct number of map registers by calling 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 QBUS address of the mapped buffer.

Device Register Procedures

The procedures summarized in this section are used to read and write device registers and internal processor registers.

MFPR Function

The MFPR function returns the current contents of a VAX processor register. The calling program must be running in kernel mode.

MTPR Procedure

The MTPR procedure moves a specified value into a specified VAX internal processor register. It can be called only from a program running in kernel mode.

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

READ_REGISTER Function

The READ_REGISTER function returns the value of a variable reference. The operation is performed by a single machine instruction and is not affected by any compiler optimizations. This is the only safe method for reading a device register, and it can also be used safely to read a shared variable.

This function should always be used, instead of a direct assignment, to read the fields in a device register. This is required because the VAX architecture does not permit certain instructions to be used to read device registers (in particular, the variable-length bit-field instructions). Using READ_REGISTER ensures that the compiler will generate only the allowed instructions.

WRITE_REGISTER Procedure

The WRITE_REGISTER procedure loads a specified value or group of values into a specified target variable reference. The write operation is performed by a single machine instruction and is not affected by any compiler optimizations. This is the only safe method for writing device registers, and it can also be used to safely write a shared variable.

This procedure should always be used, instead of a direct assignment statement, to write the fields in a device register. This is required because the VAX architecture does not allow certain instructions (in particular, the variable-length bit-field instructions) to be used to write device registers. Calling WRITE_REGISTER ensures that the compiler generates only the allowed instructions.

Chapter 7
The Network Service

Introduction

The Network Service is a set of services provided by the VAXELN Ethernet Datalink Driver (XEDRIVER or XQDRIVER). It routes messages sent between two network nodes and manages the list of universal names for the network. The Network Service calls the Datalink Driver to transmit a message; in turn, the Datalink Driver calls the Network Service to dispatch a received message.

When a process obtains 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. When the message is received at the actual destination it 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 NAME string that is not defined on the local node (that is, a universal name), the Network Service and kernel cooperate to obtain the translation. The Network Service also provides for communication with other DECnet network nodes and implements certain functions for managing nodes in the network.

Multinode VAXELN systems are configured with a Network Service at each node. However, the methods by which a program sends and receives messages are the same in this case as in the case of jobs communicating within a single node; data transmission across network nodes is transparent to the user's program.

The Network Service is also included implicitly by the System Builder when you select the remote debugging option for a system under development (see Chapter 13, "System Development," and Chapter 15, "Debugging").

This chapter discusses the Network Service in functional terms and covers the following topics:

- Network applications and message transmissions between nodes
- The basic message-sending method and its relationship to VAXELN kernel procedures
- The management of universal names (the concept of *name servers*)
- The identification of nodes in file specifications (*node names* and *node numbers*)
- An overview of network management
- The facilities for communication with non-VAXELN nodes on the same network

Network Applications

A *network application* is one in which jobs send messages to jobs on other nodes explicitly with the SEND procedure, or implicitly through I/O operations that need services, drivers, and hardware on a different node.

In VAXELN network applications, you include a Network Service in the system that runs on each node, as shown in Figure 7-1.

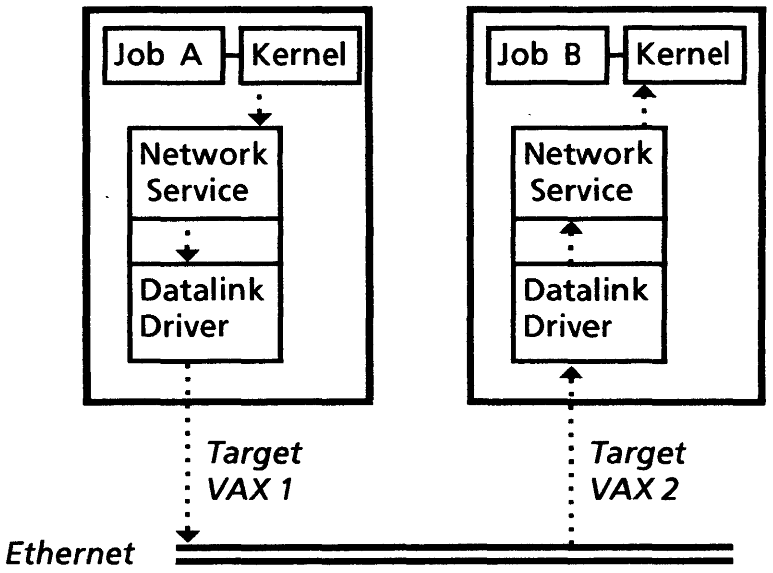


Figure 7-1. A Two-Node VAXELN Network

When Job A sends a message to Job B, the Network Service on machine 1 locates the destination in the network and delivers a formatted message to the Datalink Driver on its machine, to be transmitted on the Ethernet.

Part of the formatted message is the 48-bit Ethernet address of the destination node, machine 2. The Datalink Driver on that machine recognizes its Ethernet address in the message and forwards the message to the Network Service on its machine, which, in turn, delivers the message to a message port in the destination job, B.

Note that neither the sending nor the receiving job communicates directly with the Network Services. Instead, the kernel on each node determines that, for example, an outgoing message is not destined for any of the message ports on its node and forwards the message to its Network Service.

Among other things, this means that when jobs on the same machine send messages to each other, the Network Service is not involved, and in non-network applications, it can therefore be omitted from the system.

Although the use of circuits is strongly recommended, especially in network applications, 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 a network is used, they guarantee that messages are delivered if the physical connection is intact (which is *not* guaranteed by the Ethernet itself), 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 and reassembled into messages of the original size.

Generally speaking, these guarantees are especially important in networks. If your application seems to require communication without circuits, you probably will have to program some of these guarantees yourself, at least the guarantee of delivery.

Application Message Services

The VAXELN Network Service provides transparent message-passing extensions to the VAXELN kernel procedures, using Phase IV DECnet protocols. (An overview of the Phase IV DECnet protocols can be obtained from the *DECnet DIGITAL Network*

Architecture General Description; that document also contains a list of the Phase IV procotol specifications you can order.)

The Network Service provides extensions to the following datagram and circuit kernel procedures:

- ACCEPT_CIRCUIT
- CONNECT_CIRCUIT
- DISCONNECT_CIRCUIT
- RECEIVE
- SEND

End-Node Routing

The Network Service uses the Phase IV DECnet Routing Protocol Version 2.0 to route system-level datagrams between VAXELN nodes and other DECnet nodes.

Its Routing module provides Ethernet *end-node* routing. This means that a VAXELN system can have only one Ethernet Datalink Controller (for example, a DEUNA or DEQNA). End-node routing also means that a VAXELN system can communicate directly over the Ethernet with any other DECnet node on the same Ethernet, or, if there is a full routing system on the Ethernet (for example, a VMS system), it can communicate through the routing system to any other nodes on the entire network.

Network Services Protocol (NSP)

The Network Service uses the Phase IV DECnet Network Services Protocol (NSP) Version 4.0 and Session Control Protocol Version 1.0 to provide transparent application-level circuits to remote nodes.

The Network Service's NSP module then uses the Routing module to deliver messages to remote systems.

Logical Links

In NSP and Session Control terminology, an application-level circuit is called a *logical link*. A logical link connects two remote application-level (or *session-level*) ports together. Therefore, calling the VAXELN CONNECT_CIRCUIT procedure with a remote destination port causes the VAXELN Network Service to create an NSP logical link with the destination. The ACCEPT_CIRCUIT procedure also allows the caller to accept logical links from remote destination ports.

Datagram Size

The Network Service also uses NSP datagrams to deliver application-level datagrams. Since there is a maximum size both for messages on the Ethernet and for messages in a general DECnet network, VAXELN application-level datagrams have a maximum size.

This maximum size is also referred to as the *segment size* in NSP terminology. The segment size is a System Builder characteristic, and you should set it to the same value in all the VAXELN systems on a particular network. This characteristic should also correspond to the EXECUTOR BUFFER SIZE on non-VAXELN systems.

Because there is a header that must be prefixed to a remote datagram by the Network Service, the actual maximum size for a remote datagram is the segment size minus 32. For example, the default segment size is 576 bytes, so the largest remote datagram that can be sent is 576 – 32, or 544 bytes.

Name Servers

When you build a VAXELN system, the System Builder gives you the option of saying that the target machine for the system can be the *name server* for the VAXELN network.

A name server is a target machine whose system software includes the Network Service and which is responsible for managing the network's list of universal names. Universal names allow a job to make character-string names for its message ports available to all other jobs in the network; this way, the other jobs can identify a message destination by name, without having to know or maintain the actual PORT object values for other jobs' message ports.

In effect, the Network Service also provides a "name service," with the following functions:

- Creating universal names (the result of calling the CREATE_NAME procedure)
- Translating universal names (the result of calling TRANSLATE_NAME for a universal name)
- Deleting universal names (the DELETE procedure)

Note: The set of universal names in a VAXELN local area network is known only to the VAXELN nodes in that network, not on nodes running other systems such as VAX/VMS, nor to other VAXELN nodes not directly connected to the local area network's Ethernet.

At a particular time, there is only one node in a VAXELN network application that acts as a name server; it is responsible for the management of universal names.

Interaction with the Kernels and Network Services

The kernel and Network Service on each node interact with each other and with the name server as follows: messages sent between the Network Services and name server are retried if necessary, until a valid reply is received. That is, these transmissions are reliable.

Name Creation

When a job creates a universal name, the kernel on its node sends a message to its node's Network Service. The Network Service then sends the name and associated PORT object value to the name server. The name server enters the universal name in its table and sends an acknowledgement back to the Network Service. The Network Service waits for the acknowledgement from the name server (a message indicating the success or failure of the name creation) and forwards the reply back to its local kernel. This status is then reflected in the user's program as the completion status of CREATE_NAME.

Name Deletion

Again, the kernel informs its local Network Service of the universal name deletion, and the Network Service informs the name server. The name server removes the name from the table (unless it was already deleted) and replies to the Network Service. The caller of DELETE receives the completion status in the same way as for creations.

Name Translation

The kernel procedure TRANSLATE_NAME sends a message to the Network Service if it cannot find a translation of a name. The Network Service forwards the translation request to the name server. The name server translates the name to a PORT object value, which it returns to the Network Service in its reply. Again, via the Network Service and kernel, the caller of TRANSLATE_NAME receives the completion status of the translation.

Name Server Election

This subsection describes the way universal names are preserved in the event of the current name server shutting down.

As mentioned previously, you can designate in the System Builder that the target machine can be a name server. This characteristic really means that the node can "volunteer" or "nominate" itself as a name server, not that it will necessarily be one.

Generally, you should specify at least two such systems per VAXELN network. Having several volunteers helps assure the continuing validity of universal names if the current name server shuts down for some reason.

The process by which a machine is "elected" as the 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.
- Every node's Network Service retains the list of universal names that it has created; this is true whether or not the node's system has the *Name server* characteristic.
- Every node listens for the name server's periodic broadcast. If a time-out interval elapses with no broadcast heard, the node with the highest Ethernet address is elected as the current name

server, and every node's Network Service sends its current list of universal names to the new server.

Assuming that a name server is elected, this algorithm assures that when a VAXELN system is running on a node, the names for its message ports are available to the other nodes. On the other hand, the failure of a node does not prevent other nodes from using universal names.

Clearly, it is possible for every node that is a name server volunteer to be down at once, so as far as failure protection is concerned, the more volunteers, the better. On the other hand, the election algorithm causes many messages to be sent, in very large networks, if every node has the characteristic. *Name server* is a default characteristic in the System Builder, and we suggest that in networks with fewer than 20 nodes, you give every system this characteristic.

Node Names and Numbers

When VAXELN and non-VAXELN nodes 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.

Note, by the way, that node names are *not* needed 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 name or other identifier in the file specification; the network locations of VAXELN jobs are transparent to one another. Node names or numbers are needed only for communication between VAXELN and non-VAXELN nodes.

In DECnet networks, nodes are identified both by node name and by node number. A node name has a maximum of six characters, and a node number is an integer. Either is a unique identification of a node. The VAX/VMS command SHOW NETWORK displays both the name and number of all the nodes known to the DECnet–VAX software.

VAXELN nodes are given node names and numbers on the VAX/VMS system as usual, with the DECnet–VAX Network Control Program, or NCP. (For a brief introduction to NCP, see "Network Management," later in this chapter.)

Once NCP has been used to establish the VAXELN node in the DECnet–VAX database, it will be displayed by the SHOW NETWORK command any time it is running and contains the Network Service.

Use of Node Names in VAX/VMS

You can use a VAXELN node name from a non-VAXELN system to display directories, to perform other directory- or file-related operations on File Service volumes, and to perform network management operations.

For example, the following VAX/VMS command displays the directory [analog.data] on disk volume DISK$A on node README, which is presumably a VAXELN file-server node:

```
$ DIR README::DISK$A:[analog.data]
```

If you have used this feature on VAX/VMS before, this is the familiar syntax for network file and directory operations; the word preceding the double colon (::) is the node containing the specified directory or file.

Use of Node Numbers in VAXELN

DECnet–VAX does not have a name service facility comparable to the VAXELN Network Service's, and so, when working from a VAXELN node, you must specify a VAX/VMS node by number rather than by name. Suppose, for example, you want to open a file on the VAX/VMS node RVAXAA. The SHOW NETWORK command on the VAX/VMS system might display something like:

```
$ SHOW NETWORK
    Node        Links  Cost   Hops  Next Hop to Node

    10 RVAXAA   0      0      0     (Local)
     3 ELN1     1      3      1     UNA-0
```

Here, presumably, ELN1 is the node from which you want to access a file on RVAXAA. The file can be opened as usual, with the OPEN procedure appropriate to the language, but with the node number of RVAXAA in the file specification. For instance, in Pascal:

```
OPEN(pasvar,
    FILE_NAME := '10::sys$library:digital.dat');
```

or in C:

```
#include stdio
FILE *file_ptr;
file_ptr = fopen("10::sys$library:digital.dat","r");
```

Network Management

The Network Service provides facilities that aid the management of a network of nodes running DECnet software. The facilities are the Network Managment Listener (NML) and the Loopback Mirror.

Network Managment Listener

The NML provides a subset of DECnet Network Management Version 4.0. It provides network monitoring and control for DECnet systems. The NML functions are invoked with the VMS Network Control

Program (NCP). This program is described fully in the *DECnet–VAX System Manager's Guide*. This section describes the features of NCP that are supported remotely by VAXELN.

In order to use the NCP to invoke the VAXELN NML, the VAXELN system's node name and address must first be defined in the VMS system's network node database. This operation is usually performed when the network is installed, but you should always be sure each node in your network has a unique address and name. The following VMS commands would permanently define a VAXELN system for use by network management:

```
$ RUN SYS$SYSTEM:NCP
NCP> DEFINE NODE FRED ADDRESS 5
NCP> SET NODE FRED ALL
```

Once the node has been defined, its existence in the network can be displayed using the NCP SHOW NODE and SHOW CIRCUIT displays. (Note that the use of the term *circuit* in NCP refers to the datalink-level circuits between nodes, not the application-level circuits referred to in VAXELN programs.)

```
NCP> SHOW NODE FRED

Node Volatile Summary as of 8-JUL-1983 12:44:41

Node      State      Active  Delay  Circuit  Next Node
                      Links

5 (FRED)  reachable                 UNA-0    5 (FRED)
```

To use NCP to invoke the VAXELN NML, the NCP SET EXECUTOR command or the TELL prefix is used.

For example:

```
NCP> SET EXECUTOR NODE FRED
NCP> SHOW EXECUTOR

Node Volatile Summary as of  8-JUL-1983 10:48:00

Executor node  =  5 (FRED)

State   =   on
Identification =  VAXELN V2.0
```

The following NCP commands and options are supported by the VAXELN NML (the *italic* parts of the commands are optional and, in most cases, 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

Loopback Mirror

The Loopback Mirror can test the Network Service on a VAXELN node, from a non-VAXELN node or from another VAXELN node. 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, 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, the following command could be used:

```
NCP> LOOP NODE FRED COUNT 100
```

To test the communication between two VAXELN systems, the following command might be used:

```
NCP> TELL FRED LOOP NODE BILL COUNT 100
```

Connections with VAX/VMS Nodes

A complete explanation of VMS network I/O is beyond the scope of this manual, but the information in this section provides the VAXELN-specific details to help you get started. For full information, see the documentation for VAX/VMS and DECnet–VAX, in particular, the *DECnet–VAX User's Guide.*

Requesting the Connection from VAXELN

The kernel procedure CONNECT_CIRCUIT can be used to request a connection with a VAX/VMS program on the same DECnet network by giving the following kind of string for the DESTINATION_NAME parameter of CONNECT_CIRCUIT:

```
'nodenumber::objectname'
```

where nodenumber is as explained previously in the section "Node Names and Numbers," and objectname is the name of the "object" on the VAX/VMS system that will handle the connection.

In the default DECnet directory on the VMS system, there has to be a command procedure named objectname.COM, which should run the desired VMS program image. The command procedure is executed when DECnet–VAX gets a request for a connection to the specified object. The VMS image will then handle the connection.

Accepting the Connection on VAX/VMS

The VMS program image has two general ways of waiting for and accepting the connection from VAXELN (the operation comparable to VAXELN's ACCEPT_CIRCUIT procedure).

If the VAX/VMS program was written in a high-level language, it can use that language's OPEN procedure or equivalent to open with the name 'SYS$NET.' (In a VAX MACRO program, the $ASSIGN system service can be used.) The connection can be broken by a DISCONNECT_CIRCUIT call in the VAXELN program or a "close" operation in the VAX/VMS program.

Transparent and Nontransparent Communication

The method described so far is called *transparent communication* in DECnet–VAX because it allows the VAXELN and VMS programs to exchange information with standard I/O statements, ignoring the fact that they are separated on the network. It offers the basic mechanism for establishing a *single* connection, exchanging messages, and breaking the connection.

The alternative method is called *nontransparent communication*, which allows the VMS program to use network-specific features to handle the message exchange. The features available are a superset of those available in the transparent case, although they require more knowledge of DECnet, and more sophisticated programming. (For example, the 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 the remote VMS program as part of the NSP connection requests and acceptances.)

Nontransparent communication on VMS uses VMS 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 full information and examples, see the *DECnet–VAX User's Guide*.)

Requesting the Connection on VAX/VMS

A VMS program can request a connection with a VAXELN program by using its language's 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.

Accepting the Connection on VAXELN

The VAXELN program needs to do nothing special to accept a connection from a remote VMS program. It 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.

Connections Using DECnet Object Numbers

A VAXELN program can both connect and accept connections using 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'
```

where objectnumber is the object number (in ASCII). Once the name is created, connections can be accepted as usual.

User-Level Datagrams

VAXELN systems cannot exchange user-level datagrams transparently with non-VAXELN (VAX/VMS) nodes. However, if datagrams must be exchanged between two such systems, they can be sent nontransparently by having the user program send messages directly to the Datalink Driver.

The DEUNA and DEQNA drivers for both VAXELN and VAX/VMS allow multiple users of the datalink, providing multiplexing via the Ethernet protocol type. The Pascal interface to the VAXELN Datalink Driver (XEDRIVER or XQDRIVER) is in DATALINK.PAS and can be used in programs by including the module $DATALINK from RTLOBJECT.OLB in the compilation. (For information on the VAX/VMS driver interface, consult the *VAX/VMS I/O User's Reference Manual.*)

Chapter 8
System Security

Introduction

VAXELN includes a number of features that can be used to provide system security. In this context, the term security means that system resources and data are protected from use, examination, or modification by "unauthorized" people.

Since VAXELN is primarily intended for dedicated applications where security may be more of a nuisance than a benefit, VAXELN provides no security by default. To use the security features they must be explicitly included and enabled by the application designer. If, for example, a VAXELN system is to be included as part of a larger network of systems, it is recommended that the security features be included.

To understand the VAXELN security features and how they work to provide protection, it is important to understand what is and is not being protected from whom.

Since VAXELN is NOT intended to provide a multi-user time-sharing environment, there is no enforced protection among programs running on a single system. That is, although the VAX memory management ensures that errant programs cannot accidently modify the memory allocated to other programs, the VAXELN kernel and run-time services make no 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.

This means that the programs running on a system are in complete control of the resources of the system. Therefore, if a VAXELN-based application is to be used by unexperienced or even malicious users, the application should ensure that it protects itself and the system. Also, if protection of system resources is required, users should not be allowed to run their own programs.

However, as mentioned, many VAXELN systems exist in a larger network. This implies that programs must protect the resources of a system 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 VAXELN, 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 database of all 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, it can query the database 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 only accepts incoming circuit connections from users that are authorized in the Authorization Service's database. The File Service provides read, write, and delete protection for files on disk volumes that it controls. The Authorization Service itself uses the database to protect the database. Likewise, application programs can use the service to protect their resources and data.

This chapter describes in detail how the VAXELN security features work and how they can be used to protect resources and data.

Users

Associated with each process in an VAXELN system is a user name string and a user identification code value, or 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 any desired values by calling the SET_USER kernel procedure.

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 VAX/VMS use. On VAXELN and VAX/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, as follows: "[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 purposes. Also, group numbers less than or equal to octal 10 are considered part of the "system" groups. The use of UICs is described more fully in the section "File Service Security," later in this chapter.

A process can determine its own user name and UIC by calling the GET_USER kernel procedure. Since, as described above, the security features in VAXELN are really 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 GET_USER, though the port object connected in the circuit is then one of the arguments.

Authorization Service

The Authorization Service is the key component of the VAXELN security facilities. It maintains a database of all the remote users that are authorized to use the resources of a particular VAXELN system or network of systems. Along with answering the question of "is a particular user authorized," the service also maintains a UIC for each authorized user.

The Authorization Service's primary task, therefore, is to determine the identity of the requestor of a network connection request. To do this, the service gets the requestor's host system user name and node name and looks them up in the authorization database. It can also accept a specific user name and password ("access control string") and look them up in the database.

The following example describes how the service works, as illustrated in Figure 8-1.

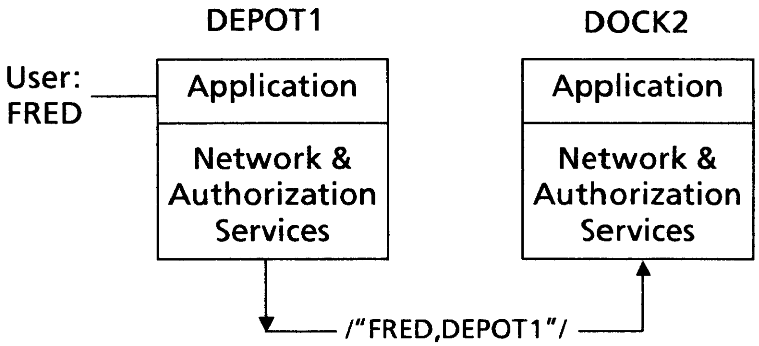

DEPOT1
DOCK2
User: FRED
Application
Network & Authorization Services
Application
Network & Authorization Services
/"FRED,DEPOT1"/

Figure 8-1. Authorization Service Example

If a user named FRED executing a program on a VAXELN node named DEPOT1 issues a request for a service on another node named DOCK2, the Network Service on node DEPOT1 sends FRED and DEPOT1 in the connection request message to the Network Service on node DOCK2. The 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, and if "Yes," it returns the UIC the user is to be identified with.

This type of authorization is termed "proxy" access control. It means that 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 (or file open) specifies a specific user name and password with the connection request. This specification is called an "access control" string. This feature provides a means of assuming a new identity on the remote system.

Both of these types of VAXELN authorization are provided compatibly by VAX/VMS. Other DIGITAL operating systems currently only support the destination authorization provided with access control strings.

The CONNECT_CIRCUIT procedure allows a remote destination to be specified as a string using the DESTINATION_NAME optional 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 single space.

To specify the remote destination by object name, the string can have the following forms:

```
'nodenumber::objectname'

'nodenumber"username password"::objectname'

'nodenumber"username"::objectname'

'nodenumber"[ggg,mmm] password"::objectname'
```

For example,

```
CONNECT_CIRCUIT(p, DESTINATION_NAME :=
    '3"FRED SWIZZLE"::TESTOR');
```

would connect to object TESTOR on node number 3 using a user name of FRED and a password of SWIZZLE.

To specify the remote destination by object number, the string can have the following forms:

```
'nodenumber::objectnumber'

'nodenumber"username password"::objectnumber'
```

```
'nodenumber"username"::objectnumber'

'nodenumber"[ggg,mmm]
   password"::objectnumber'
```

For example,

```
CONNECT_CIRCUIT(p, DESTINATION_NAME : =
   '4"[10,150] QUAKE"::129');
```

would connect 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.

For connection to a port in a VAXELN system, the string can also have 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,

```
CONNECT_CIRCUIT(p, DESTINATION_NAME : =
   'NODEA"FRED ABC"::TEST');
```

would connect to object TEST on node name NODEA using a user name of FRED and a password of ABC.

Since OPEN 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,

```
OPEN(f, FILE–NAME : =
   '3"FRED SWIZZLE"::FILE99.DAT');
```

would open FILE99.DAT on node number 3 using a user name of FRED and a password of SWIZZLE.

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. These steps should be followed:

1. Set the *Authorization required* entry on the *Edit Network Node Characteristics* menu to "Yes" or "No." When this characteristic is set to "Yes," the Network Service will not allow inbound circuit connections unless it can authorize the user via an Authorization Service. When set to "No," the Network Service will not authorize inbound connections via the Authorization Service.

   This feature should be set to "Yes" if security is being used.

2. Set the *Authorization service* entry on the *Edit Network Node Characteristics* menu to either "Local," "Network," or "None." When set to "Local," the service is included in the system image where it only handles authorization for the local (target) system. When set to "Network," the service is included, but it handles authorization for any node in the local area network that does not have its own service. When set to "None" (the default), no service is included in the system image.

There can be only one node with a "Network" Authorization Service running at one time on a particular local area network. The "Network" characteristic is implemented using the VAXELN universal name feature, so there must be at least one names server node in the network for a "Network" Authorization Service to properly function.

3. Set the *Authorization file* entry on the *Edit Network Node Characteristics* menu to specify the authorization data file. The data file must exist on either the same node as the Authorization Service or one that the service is authorized to access (for example, one 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, it creates a new one. The file should only be modified by using the maintenance procedures described in the section "Authorization Service Maintenance," later in this chapter.

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 only be modified 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.

Authorization Procedures

The procedures summarized in this section are used to set or return the user identity of processes.

SET_USER Procedure

The SET_USER procedure sets the user identity of the current process. A string of 1-20 characters specifies the user name to be associated with the process. An integer supplies the UIC to be associated with the process.

The primary use of a process' user name and UIC is to enable its remote requests to be properly authorized on a remote system.

When a port is used in a call to CONNECT_CIRCUIT with a remote destination, only the calling process' user name is sent to the destination system. The user name is then authorized on the destination system. The UIC available to the destination partner process via GET_USER is then the UIC authorized by the destination Authorization Service. That is, the UIC set by SET_USER is only valid for use on the local system; it is up to the remote destination to determine the proper UIC at the destination.

If the CONNECT_CIRCUIT procedure is called with a remote destination name parameter that includes a user name and password, such as

```
DESTINATION_NAME: = 'NODEA"FRED ABC"::TEST'
```

the specified user name and password is sent to the remote system rather than the user name set via SET_USER. The user name and password is then authorized on the remote system. The UIC available to the destination partner via GET_USER is then the UIC authorized by the destination system.

GET_USER Procedure

The GET_USER procedure returns the user identity of either the calling process or the partner process connected via a circuit to the caller's port. An optional argument specifies a port connected in a circuit. Other optional arguments return the user name string and the UIC.

If the circuit parameter is specified, the port must be currently 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, GET_USER can only be used to provide information about the object of a connection, not the subject.

If the circuit is from a remote user, but there is no Authorization Service available in the system (the *Authorization required* characteristic is "No"), GET_USER returns zero for the UIC parameter.

Although the Network Service ensures that an inbound connection request is from an authorized user, it is up to the application program that accepts the request to use the user's identity to protect its resources. The GET_USER procedure should be used for this purpose.

For example, the following segment of a Pascal program 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
   .
   .
```

Authorization Example

The following example illustrates the two types of authorization and the use of SET_USER and GET_USER.

Suppose there are two nodes: DEPOT1 and DOCK2. The Authorization Service database for node DOCK2 includes two entries:

User: FRED	Host node: DEPOT1	UIC: [1,2]	
User: SAM	Host node:	UIC: [1,3]	Password: NOODLE

The first entry is a proxy authorization because it includes a host node name; the second entry is a destination authorization because it includes a password instead of a node name.

If program "A" on node DEPOT1 executes the following:

```
KER$SET–USER(username : = 'FRED');
CONNECT–CIRCUIT(destination–name
     : = 'DOCK2::TESTOR');
```

And program "B" on node DOCK2 executes:

```
CREATE–NAME(p, 'TESTOR');
ACCEPT–CIRCUIT(p);
KER$GET–USER(circuit : = p,
     username : = partner–user, uic : = partner–uic);
```

Then, program "B" will get a user name value of FRED returned in variable PARTNER_USER and a UIC value of [1,2] (%X00010002) returned in variable PARTNER_UIC. The Authorization Service has used the proxy entry to authorize the remote user.

If instead, program "A" on node DEPOT1 executes the following:

```
CONNECT_CIRCUIT(destination_name
     := 'DOCK2"SAM NOODLE"::TESTOR');
```

And program "B" on node DOCK2 executes:

```
CREATE_NAME(p, 'TESTOR');
ACCEPT_CIRCUIT(p);
KER$GET_USER(circuit := p,
     username := partner_user, uic := partner_uic);
```

Then, program "B" will get a user name value of SAM returned in variable PARTNER_USER and a UIC value of [1,3] (%X00010003) returned in variable PARTNER_UIC. The Authorization Service has used the destination entry to authorize the remote user.

Authorization Service Utility Procedures

The Authorization Service provides the capability to maintain the authorization database. 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, VAXELN includes a set of utility procedures that handle the protocol, eliminating the need for user programs to explicitly code the protocol.

These procedures all assume that the calling program has connected a circuit to the Authorization Service's AUTH$MAINTENANCE port.

For example, in Pascal:

```
VAR
   c: PORT;
   user: auth$username;
   uic: integer;
   node: auth$nodename;
   .
   .
CREATE_PORT(c);
CONNECT_CIRCUIT(c, DESTINATION_NAME
     : = 'AUTH$MAINTENANCE');
user : = 'FRED';
node : = 'DEPOT1'
uic : = %X00010002;
ELN$AUTH_ADD_USER(CIRCUIT : = c,
     USERNAME : = user,
     NODENAME : = node, UIC : = uic);
```

To use the following maintenance utility procedures include the $AUTHORIZE_UTILITY module in the compilation. In the following procedures, the host node name parameter can be specified as either an alphanumeric string for VAXELN nodes (for example, DEPOT1) or a numeric node address for non-VAXELN nodes (for example, 5 or 3.5).

AUTH_ADD_USER Procedure

The AUTH_ADD_USER procedure adds a new user record to the authorization database. This procedure requires that the caller is authorized with a system group UIC (that is, a UIC of less than or equal to %X0008FFFF or [10,177777]).

Arguments uniquely specify the user name and node name of the new user. If a node name is specified, the database record represents a proxy authorization; if a node name is not specified, the record represents a destination authorization. If a destination authorization record is added, a password is stored with the record. Passwords are always stored in a "hashed" form so they cannot be read once stored.

Additional arguments specify the port connected to the Authorization Service's AUTH$MAINTENANCE port, the UIC that is associated with the user, and an arbitrary string of user-specified data that is stored with the user record for use by applications.

The reserved name $ANY can be specified for either or both the user name and node name. If $ANY is specified for a user name, it means that any user from the specified node that does not match one of the explicit user names is authorized with the specified UIC. If $ANY is specified for a node name, it means that any user with the specified name from any node that does not match one of the explicit node names is authorized with the specified UIC. If $ANY is specified for both, it means that all users that don't match an explicit user name / node name combination are authorized with the specified UICs.

AUTH–MODIFY–USER Procedure

The AUTH–MODIFY–USER procedure modifies an existing user record in the authorization database. This procedure requires that the caller is authorized with a system group UIC (that is, a UIC of less than or equal to %X0008FFFF or [10,177777]).

Arguments uniquely specify the user name and node name of the record to be modified, as well as the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port. Other arguments optionally specify values to replace the current values in the record. Particular fields can be modified without changing other fields in the record.

Since the hashing algorithm for the password includes the user name, if the user name is modified, the password must be reset as well.

AUTH–REMOVE–USER Procedure

The AUTH–REMOVE–USER procedure removes an existing user record from the authorizaton database. This procedure requires that the caller is authorized with a system group UIC (that is, a UIC of less than or equal to %X0008FFFF or [10,177777]).

Arguments uniquely specify the user name and node name of the user to be removed, as well as the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

AUTH–SHOW–USER Procedure

The AUTH–SHOW–USER procedure returns the authorization database information for the specified user or users. Arguments uniquely specify the user name and node name of the user records to be accessed, as well as the port connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

This procedure calls a user-specified routine with the values of a specified user record or all the records in the authorization data file. The routine is invoked only if the specified user entry is found in the authorization database.

File Service Security

The File Service uses the VAXELN features described in this chapter to protect the disk volumes and files that it manages. Since the File Service uses the Files–11 on-disk structure, it uses the standard Files–11 protection facilities.

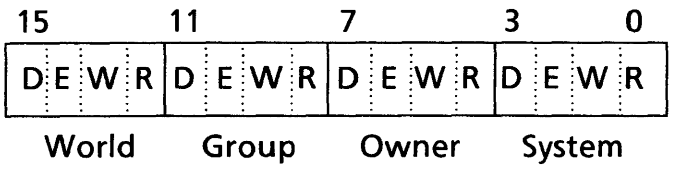
These facilities are compatible with VMS and are described below:

- 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, there is no Authorization Service), 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" which describes how the File Service is to protect the file from the following categories of users:

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 above catagories.

  The protection mask is a 16-bit word that is composed of four fields. Each of the four fields corresponds to one of the four categories of user described above. Each of the four fields consists of 1-bit indicators that specify the access allowed to the category: read, write, execute, and delete.

  The protection mask is shown in Figure 8-2.



8-2. Protection Mask

If a bit is set in a category's field, then users in that category are denied the corresponding access. For example, if bit 1 is set, then system users are denied write access.

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, 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 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 INIT–VOLUME procedure.

- The default protection mask for files on a new disk volume can be specified as a parameter to the INIT–VOLUME procedure.
- If the File Service is unable to determine the UIC of a user requesting access to a file (for example, there is no Authorization Service), it allows unprotected access by the user. (See the description of the *Authorization required* Network Node Characteristic earlier in this chapter for a means of preventing this unprotected access.)

Chapter 9
The File Service

Introduction

The File Service is a set of services provided by the disk and tape drivers in a system to enable using disk or tape files, respectively, for program I/O. It is not used for I/O with terminals or printers.

The File Service consists of a disk File Service and a separate tape File Service:

- The disk File Service provides Files–11 On-disk Structure Level 2 file services. It is compatible with the VAX/VMS Version 4.0 file system and RMS–32.
- The tape File Service is based upon Version 3 of the ANSI standard for magnetic tapes. It provides users with a convenient means of transporting files to and from VAX/VMS systems, since it is compatible with the VAX/VMS Version 4.0 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 the section "Interface with Disk and Tape Drivers," later in this chapter.

When several VAXELN systems are running on nodes in a local area network, only one 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.

In the case of 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 stored on a non-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 device specifications, volume names, file specifications, the File Access Listener, the use of file service volumes from VMS, file service operations, file utilities, disk and tape utilities, the interface with disk and tape drivers, and the Data Access Protocol.

Device Specifications

The devices used by a VAXELN application must be described to the System Builder on its *Device Description* menus, as explained in Chapter 13, "System Development."

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.

Table 9-1 lists the storage device types used in VAXELN programming.

Table 9-1. 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, RQDX (MicroVAX) interface to RX50 diskettes and RD51 or RD52 Winchester disk, or RC25 fixed and removable disks
MU	TK50 streaming cartridge tape drive

Note that the device types shown in Table 9-1 are simply conventional names for these devices; you can use any names you like as long as they are used consistently in the System Builder and in user programs.

Volume Names

After you enter the device specifications for the drives the File Service uses, you can supply volume names (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 Chapter 13, "System Development").

The volume name is paired with a device specification; for example:

```
"DUA1 TEST1", "DUA0 TEST2"
```

Here, TEST1 is established as the volume to be mounted on drive DUA1 and TEST2 as the volume for DUA0. The first volume specified in the list (here, TEST1) identifies the *default volume* for the File Service. That is, any file specification that lacks a volume name or device name refers to this volume.

The controller name (here, DUA) is also supplied as an argument to the driver; Chapter 13 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 will attempt to mount whichever volume is placed in the drive. If the volume name specified is not the same as the one with which the volume was initialized, the File Service will mount it anyway and print an informational message on the target machine's console terminal.

If no argument is supplied for a drive, it is not initially mounted by the File Service, but can be mounted dynamically with the MOUNT_VOLUME procedure or with the MOUNT_TAPE_VOLUME procedure, as appropriate. If the drive is a disk, it can also be used directly (for non-file "logical I/O") by opening it as a file, with the OPEN procedure. (Logical I/O is explained in Chapter 10, "Device Drivers.")

Note: If you attempt to mount a VAX/VMS disk volume that was improperly dismounted (for example, if the VAX/VMS system crashed), the File Service prints a warning message on the target machine's console. The volume should be remounted on VAX/VMS, which rebuilds it, then it can be mounted on the VAXELN system. You can successfully mount a VAXELN or VAX/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.

File Specifications

When used in programs (for example, in the Pascal OPEN procedure), file specifications with volume labels of the form

DISK$name or TAPE$name

are referred by the File Service to a particular mounted disk or tape, respectively, on the target machine configuration. If you supplied volume names with the System Builder's *Edit System Characteristics* menu, name must match a volume name you defined with the System Builder. If you did not use this menu, or if a different volume was 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 MOUNT_VOLUME or MOUNT_TAPE_VOLUME procedure), its DISK$ or TAPE$ name is established as a *universal* name by the File Service and uniquely identifies the volume from any local area network node.

If some other process in the application mounts a volume with the same volume name, its 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.

Note: If the volume (for example, the disk in drive DUA0) is mounted, you can also refer to it with an explicit device name. For example, in Pascal:

```
OPEN(f,FILE-NAME : = 'DUA0:[TEST]TEST.DAT')
```

The corresponding example in C is:

```
#include stdio
FILE *file-ptr
file-ptr = fopen("DUA0:[TEST]TEST.DAT","r");
```

Device names such as DUA0 are local to their network node. For example, suppose the entries on the *Edit System Characteristics* menu are:

```
"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:

```
#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.

File Access Listener

The File Access Listener is built into VAXELN systems that support file access from remote nodes. Its inclusion is controlled by an entry on the System Builder's *Network Node Characteristics* menu, described in Chapter 13, "System Development."

The function of the File Access Listener is to handle connection requests (such as file openings) that involve different network nodes, including incoming requests from VAX/VMS nodes. Accordingly, the inclusion of the File Access Listener in a VAXELN system also presumes that the Network Service is present.

Note that the inclusion of the File Access Listener does not necessarily mean that the File Service must be present. For instance, a system that included a line printer plus its device driver and also included the File Access Listener could be used as a *print server* by the network community. The File Access Listener would accept I/O connection requests directed at the printer and establish the connection with the printer driver's message port(s).

Using File Service Volumes from VMS

The File Service uses the same on-disk and tape file structure as VAX/VMS, and most VMS file-handling operations are supported, such as COPY, DIFFERENCES, DIRECTORY, EDIT, and so forth.

For example, assume that node MILDEW is a VAX with two disks, DQA0 and DQA1, and that it is running a VAXELN system with the File Service, Network Service, and File Access Listener.

You can perform COPY with the VAX/VMS command

```
$ COPY MILDEW::[directory]file.txt  *.*
```

which refers by default to [directory]file.txt on DQA1, or the command

```
$ COPY MILDEW::DISK$TEST2:[directory]file.txt  *.*
```

which refers to DQA0. You can also use the device specification directly, as in:

```
$ COPY MILDEW::DQA0:[directory]file.txt  *.*
```

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 Language Reference Manual and the *VAXELN C Run-Time Library Reference Manual*, as appropriate to the programming langage in use.

File Utility Procedures

The file utility procedures provided by the File Service are summarized in this section. To use these procedures, the $FILE_UTILITY module appropriate to the language you are using must be included in the compilation of your program. (For more information, see Chapter 12, "Program Development.")

Note: The VAXELN File Service supports all of the file utility procedures for disk and tape volumes. However, the CREATE_DIRECTORY, DELETE_FILE, RENAME_FILE, and PROTECT_FILE procedures are invalid for tapes. An error message is returned if an attempt is made to apply them to tape volumes.

COPY_FILE Procedure

The COPY_FILE procedure makes an exact duplicate of a specified file. A string of 1–255 characters gives the file specification of the source file to copy. A second string of 1–255 characters gives the file specification of the destination file to be copied to.

Optional parameters return the resultant filename 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 will be returned, indicating which file contains the error.

Note: The COPY_FILE procedure provides the only means of creating an ISAM or RELATIVE organization file in VAXELN, by copying an existing file of the organization.

CREATE_DIRECTORY Procedure

The CREATE_DIRECTORY procedure creates a directory on the specified file service disk volume. This procedure is invalid for tape volumes.

A string of 1–255 characters gives the file specification for the directory or subfile directory to be created. A file owner user identification code (UIC) can also be specified. An optional parameter returns the resultant filename string of the directory file actually created. For example, in Pascal,

```
CREATE_DIRECTORY('DISK$TEST:[DATA]');
```

creates the directory DATA.DIR in the master file directory of the volume.

Note that the directory must be created on a VAXELN disk volume; the procedure cannot create a directory on a remote non-VAXELN system's volume. Also, the procedure creates only the last directory in the specification; any intermediate directories must already exist.

DELETE_FILE Procedure

The DELETE_FILE procedure deletes a file from a mounted disk volume. This procedure is invalid for tape volumes.

A string of 1–255 characters gives the file specification, 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. An optional parameter stores the resultant filename string of the deleted file.

DIRECTORY_CLOSE Procedure

The DIRECTORY_CLOSE procedure closes an existing directory on a mounted disk volume. A variable supplies a pointer to the directory file variable.

DIRECTORY_LIST Procedure

The DIRECTORY_LIST procedure obtains the next filename from a mounted disk directory. A variable supplies a pointer to the directory file. If more than one directory is traversed by DIRECTORY_LIST, the directory name will change. An optional variable supplies a pointer to the file attributes record.

DIRECTORY_OPEN Procedure

The DIRECTORY_OPEN procedure opens an existing directory on a mounted disk volume in preparation for a 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 1–255 characters gives the file specification of an existing directory to search for. The general form of the character string is:

```
disk:[directory]filename.type;version
```

The filename, type, and version can use the "wildcard" characters, % and *, as in VAX/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 string

```
DISK$TEST:[testdata]*A%%C.*;*
```

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. Wildcards are not allowed in volume names or, for VAXELN disks, in directory specifications.

If the directory is on a non-VAXELN (for example, VAX/VMS-serviced) disk, the asterisk (*), percent (%), 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 1–64 characters receives the resultant node specification or server process port name, and an optional variable supplies a pointer to the file attributes record.

PROTECT_FILE Procedure

The PROTECT_FILE procedure changes the protection of a disk file. This procedure is invalid for tape volumes.

A string of 1–255 characters gives the file specification. The procedure sets the file ownership UIC and/or protection code for the specified file. An optional parameter returns the resultant filename string of the file.

RENAME_FILE Procedure

The RENAME_FILE procedure renames a disk file. This procedure is invalid for tape volumes.

A string of 1–255 characters gives the current file specification; no wildcard characters are permitted. (To rename several related files, use DIRECTORY_LIST to find them and RENAME_FILE to rename each one.) A second string of 1–255 characters gives the new file specification. Optional parameters return the resultant filename 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 filename.

Disk Utility Procedures

The disk utility procedures provided by the disk File Service are summarized below. To use these procedures, the $DISK_UTILITY module appropriate for the language you are using must be included in the compilation of your program (see Chapter 12, "Program Development").

DISMOUNT_VOLUME Procedure

The 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 non-file logical block I/O.

A string of 1–30 characters names the device; for example, 'DQA1' for drive 1 on disk controller DQA. Note that the user must have RWED privileges to dismount a volume.

INIT_VOLUME Procedure

The 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, but only if 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 1–30 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 1–12 characters gives the volume label for the disk.

An optional argument supplies the default extension quantity in blocks for all 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 8, "System Security," for more information on protection.)

Other optional arguments designate:

- 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 (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. These 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. A null list can be specified.

MOUNT_VOLUME Procedure

The MOUNT_VOLUME procedure mounts a disk for use as a file-structured volume. The procedure requires the device and 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 1–30 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 1–12 characters supplies the volume label. If it is omitted, the procedure mounts whichever volume is loaded in the indicated drive.

Tape Utility Procedures

The tape utility procedures provided by the tape File Service are summarized below. To use these procedures, the $TAPE_UTILITY module appropriate for the language you are using must be included in the compilation of your program (see Chapter 12, "Program Development").

DISMOUNT_TAPE_VOLUME Procedure

The 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 1–30 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.

INIT_TAPE_VOLUME Procedure

The INIT_TAPE_VOLUME procedure initializes a file service magnetic tape as a tape volume that conforms to ANSI standard X3.27-1978. Tapes must be initialized before they are used. The procedure requires the device and 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 1–30 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 1–6 characters gives the volume label for the tape. An optional argument designates the density of data recorded on the tape.

MOUNT_TAPE_VOLUME Procedure

The 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 and 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 1–30 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 1–6 characters supplies the volume label. If it 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.

Interface with Disk and Tape Drivers

This information is provided in case you are writing new disk or tape drivers that will use the File Service or in case you want to study the drivers supplied with the development toolkit. Otherwise, this information is not needed for normal use of VAXELN.

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. (See, for example, ELN$:DUDRIVER.PAS and ELN$:MUDRIVER.PAS.) The arguments are functions (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. (For example, in the case of DUDRIVER, the initializing process exits.)

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 the start-up event value (startup_event), the file context (file_context), a string (drive_name) naming the drive (typical drivers take the controller name as a program argument and concatenate a digit to it to form the drive name), and 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 module ELN$:DAP.PAS contains the Pascal language declarations of the two disk routines described above and the declarations of the function types you can use to declare action routines for ELN$FILE_INITIALIZE. The two tape routines described above 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.

The corresponding definitions for disk drivers written in C are contained in the module $DAP in ELN$:VAXELNC.TLB. This module is included in the compilation of your driver source module via a command of the 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 database should, therefore, either be statically allocated or be allocated on the heap.

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. It is used by the Pascal and C run-time 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 writing file- or record-oriented device drivers (or for studying the ones we supplied). Unless you are writing file- or record-oriented device drivers (including disk or tape drivers that will not use the File Service), you need not be concerned with this subject in normal use of the development system. A typical case for using the DAP would be 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 prespecified 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, we suggest you study the driver and definition sources supplied with your development system.

Figure 9-1 illustrates the message flow involved in a typical I/O operation.

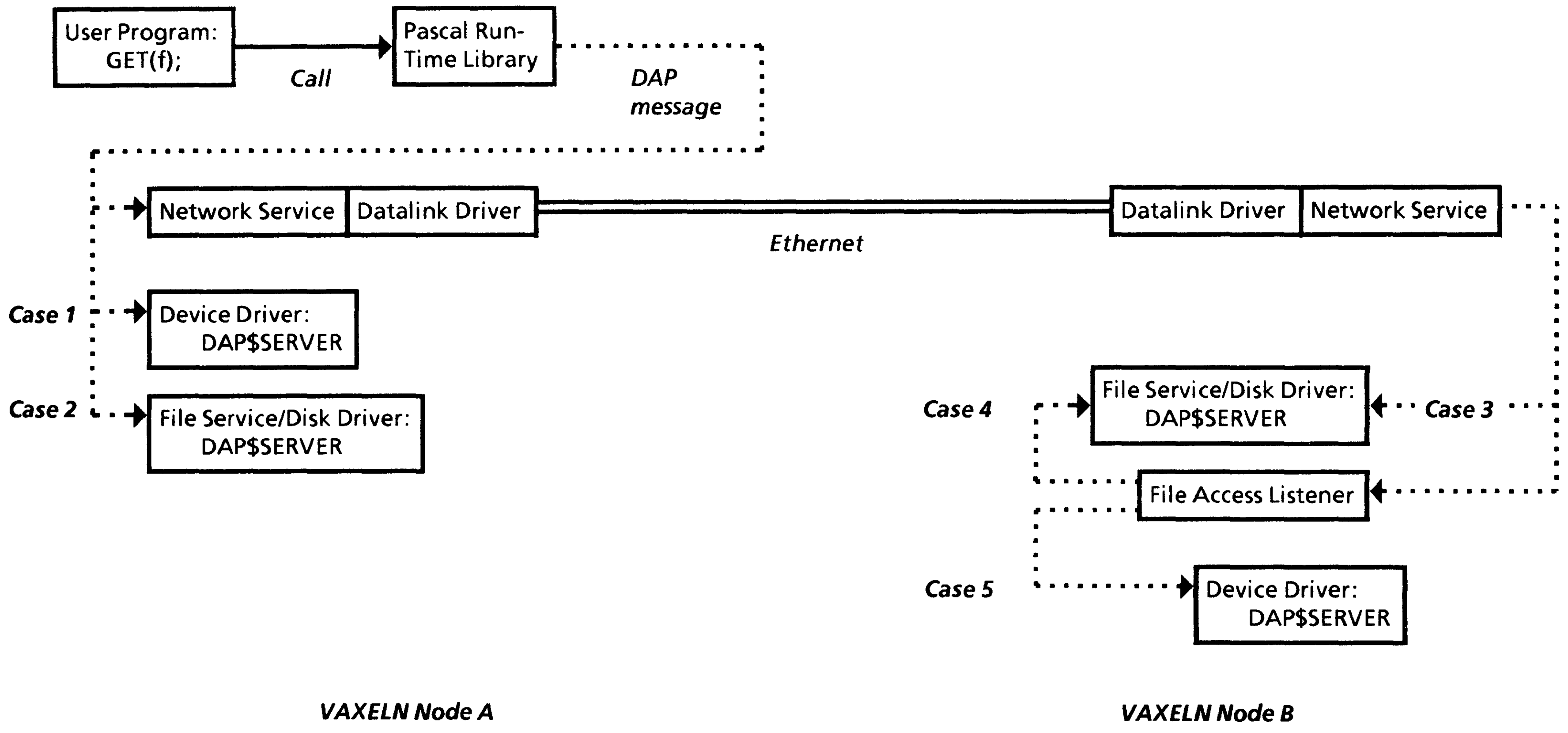


Figure 9-1. DAP Message Transmission ("Read" Request)

In the example illustrated by Figure 9-1, a user program makes a "read" request (the Pascal GET procedure). When the run-time 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 Here, the program has opened a local terminal (for instance) 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 later in this section.)

Case 2 Here, 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 Here, 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 (via the datalink drivers) over the Ethernet to the node (here, B) 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 Here, the OPEN call used an explicit node name to access a file on a mounted disk (DUA1), as in:

```
OPEN(f, FILE–NAME : =
'B::DUA1:[mydir]file.dat')
```

After transmission to node B, the message is intercepted by that node's File Access Listener and sent on to the File Service on that node. (In most respects, this case also applies if node B is a VAX/VMS node, although the node is then specified by number instead of by name; similarly, it could occur if node A is a VAX/VMS node at which a comparable OPEN call was made from a VAX/VMS program.)

Case 5 Here, the OPEN call specified a node explicitly, to open a remote terminal, as in:

```
OPEN(f,FILE–NAME : = 'B::TTA0:')
```

This is simply the network version of Case 1; the terminal TTA0 on node B 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. Note that Figure 9-1 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, it 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), plus 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.

General Principles

In data communication, a *protocol* is a definition of a set of messages and, usually, the means of exchanging the messages.

Consistent with this idea, 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 (the concept of action routines).

The DAP assumes that a communication path already exists for the messages, which, in VAXELN programming, is a circuit. (See Chapter 5, "Interjob Communication.")

The low-level operations of locating the communicating processes and formatting, interpreting, and transmitting messages are done for you by run-time 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 and supply it 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.

Action Routines and DAP$SERVER

Essentially, 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 via the $DAP include module contained in ELN$:VAXELNC.TLB.

DAP Data Types

For each kind of action routine, there is a set of data types representing the routine's parameters. In addition, the result type dap$l-status (as in the above example) represents the success/failure status of each action routine call. For the actual definitions of the types of action-routine parameters and the result type dap$l-status, see the source file DAP.PAS, supplied with your development system.

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.

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 10
Device Drivers

The VAXELN development system includes disk drivers, a tape driver, printer drivers, terminal drivers, and real-time device drivers. This chapter discusses the features of these drivers.

Disk Drivers

Included with your development system are device drivers for the following mass storage devices:

- A variety of disk devices which use the UNIBUS via a UDA50 UNIBUS disk adapter (DUDRIVER) or the RQDX disk interface on the MicroVAX; examples include the RA80, RA81, and RA60 disk drives on VAX systems and the RX50, RD51, and RD52 disk drives on MicroVAX systems. In addition, the RC25 controller is available for both the VAX QBUS and the VAX UNIBUS.
- RB02 and RB80 disks attached to the Integrated Disk Controller of the VAX–11/730 (DQDRIVER).
- TU58 (VAX–11/730 and 750) console tape cartridges, which, operationally, resemble disk devices (DDDRIVER).

The corresponding driver must be part of the system running on the same machine that has the actual disk interface and drives. If you are using the supported disk types and the 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 a case is how to include the drivers in systems. (See the instructions in Chapter 13, "System Development.")

"Logical I/O"

When a disk is not mounted, you can access it directly using Pascal or C I/O procedures. You can "open" a disk for non-file I/O (called "logical I/O") by giving its device name (for example, 'DQA1:'—the colon is required) to the Pascal OPEN procedure (or the corresponding open functions in C) instead of a file name. Operations performed on the opened file variable then 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.

In other words, logical I/O means simply that your program maintains and uses its own information about the logical structure of records in the file. It is up to your program to interpret the structure of individual records read from the disk, to record the placement of records relative to one another, and to do the other things normally done by the File Service.

Note: When you open a disk for logical I/O, no other job can access the disk.

You can write disk drivers of your own 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 used by the Pascal and C I/O procedures, see Chapter 9, "The File Service."

Disk Capacities and Other Specifications

Table 10-1 gives the basic specifications of the disk device types for which drivers are supplied.

Table 10-1. Disk Devices

Model	Device Code	Type	Bytes/Disk	Disks/Drive	Drives/Controller	Driver Image
RB02	DQ	Cartridge	10,485,760	1	4[1]	DQDRIVER.EXE
RB80	DQ	Fixed disk	124,214,270	1	1[1]	DQDRIVER.EXE
TU58	DD	Tape cartridge[2]	262,144	2	2	DDDRIVER.EXE
RX50	DU	Diskette	409,600	2	4[3]	DUDRIVER.EXE
RD51[4]	DU	Fixed disk	10,485,760	1	2[3]	DUDRIVER.EXE
RD52[4]	DU	Fixed disk	29,360,128	1	2[3]	DUDRIVER.EXE
RA60[5]	DJ	Cartridge	214,958,080	1	4[6]	DUDRIVER.EXE
RA80[5]	DU	Fixed disk	126,877,696	1	4[6]	DUDRIVER.EXE
RA81[5]	DU	Fixed disk	478,150,656	1	4[6]	DUDRIVER.EXE
RC25[5]	DA	Fixed disk/ Cartridge	27,262,976	2	1[7]	DUDRIVER.EXE

[1]The RB02 and RB80 use the same controller, the Integrated Disk Controller (RB730) on a VAX–11/730. A total of four drives can be attached to the controller, and at most one of them can be an RB80. RB02 disks are identical to RL02 disks, and cartridges can be interchanged between these two drive types.

[2]The TU58 cartridge is the console medium on VAX–11/730 and VAX–11/750 processors. It 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.

[3]The RQDX controller interfaces up to four disk drives to the MicroVAX (Q22) bus; up to two of these drives can be Winchester (RD51 or RD52) disks.

[4]RD51 and RD52 devices support controller-initiated bad block replacement; that is, the hardware automatically handles bad blocks.

[5]RA60, RA80, RA81, and RC25 devices support host-initiated bad block replacement; that is, the driver will automatically revector bad blocks as they occur on the disks.

[6]The UDA50 disk adapter interfaces RA60, RA80, and RA81 disks to the VAX UNIBUS.

[7]The RC25 controller is available for both the VAX QBUS and the VAX UNIBUS.

General Features of the Disk Drivers

All the supported disks and drivers include the disk File Service, which supports the Files–11 on-disk structure. This is the same on-disk file structure as used in VAX/VMS. This feature allows disk volumes to be moved to a VAX/VMS system and used with VMS software. It also allows disks mounted on VAXELN systems to be used by most VMS file-handling commands when the VAXELN system or systems are part of a network with VMS systems.

Interface to File Service

All drivers use the File Service to perform actions on the disk. The actions performed are:

- Open: Prepare a device and its driver for program I/O. Performed when a disk volume is mounted or when the first user program accessing the disk for logical I/O calls the Pascal OPEN procedure or the corresponding C open functions.
- Get: Read data from disk. Performed when information is retrieved from a disk volume using the Pascal input procedures or the corresponding C input functions.
- Put: Write data on disk. Performed when information is added to a disk volume using the Pascal output procedures or the corresponding C output functions.
- Close: Terminate input/output exchange with a user program. Performed when a disk volume is dismounted or when the last user program accessing the disk for logical I/O calls the Pascal CLOSE procedure or the corresponding C close functions.

Recovery from Power Failure

When disks are online and mounted, they are brought back online and remounted automatically following a power failure. The disk controller is reinitialized by the device driver. The file service operations that were in progress when the power failed are retried, and the disks are ready for use again without manual intervention.

Note that spinning down an RC25 controller and subsequently spinning it back up is equivalent to a power-failure recovery. The actions described above apply in this case.

Tape Driver

Included with your development system is a device driver for the TK50 magnetic streaming cartridge tape drive (MUDRIVER), which is applicable for all other byte-structured magnetic tape mass storage control protocol (TMSCP) tape drives as well. The driver must be part of the system running on the same machine that has the actual magnetic tape interface and drive.

If you are using the supported tape type and the driver as supplied, you can regard the driver (and the File Service) as a self-contained program that performs tape I/O for you. All you need to know in such a case is how to include the driver in systems. (See the instructions in Chapter 13, "System Development.")

"Logical I/O"

There is no logical I/O to a tape in the sense that there is to a disk, since all tape file operations are done in the context of the ANSI file structure, which the user cannot directly read or write.

Tape Specifications

Table 10-2 gives the basic specifications of the TK50 tape drive.

Table 10-2. Tape Specifications

Model	Device Code	Type	Driver Image
TK50	MU	Streaming Cartridge	MUDRIVER.EXE

General Features of the Tape Driver

The supported tape and driver includes the tape File Service, which supports the ANSI tape file structure. This is the same tape file structure as used in VAX/VMS. This feature allows tape volumes to be moved to a VAX/VMS system and used with VMS software. It also allows tapes mounted on VAXELN systems to be used by most VMS file-handling commands when the VAXELN system or systems are part of a network with VMS systems.

Interface to File Service

The tape driver provides the File Service the following actions, which the File Service uses to perform requested file operations:

- Open: Prepare a device and its driver for program I/O. Performed when a program first accesses a particular device or when a tape volume is mounted.
- Close: Terminate input/output exchange with a user program. Performed when a program is

finished accessing a particular device or when the tape volume is dismounted.

- Get: Asynchronously read the next block from the tape and return a context to the read operation.
- Put: Asynchronously write the next block to the tape and return a context to the write operation.
- Reposition: Asynchronously reposition the tape and return a context to the reposition operation. Performed when a new file is accessed.
- Tapemark: Asynchronously write a tapemark to the tape and return a context to the tapemark operation. Performed when a file is closed or the tape is closed.
- Erase: Asynchronously erase all data from the tape and return a context to the erase operation. Performed when all data on the tape must be physically removed.
- Return: Provide the status of the completed action of the context given.

Recovery from Power Failure

In case of a power failure, if tapes are online and mounted, they are automatically brought back online, remounted, rewound to the beginning, and repositioned to the last known position. The tape controller is reinitialized by the device driver. The File Service operations that were in progress when the power failed are retried, and the tapes are ready for use again without manual intervention.

Error Recovery

TMSCP devices do their own error detection and recovery. The only data errors reported are unrecoverable errors, which the driver forwards to the File Service.

Printer Drivers

The program images LCDRIVER.EXE and LPVDRIVER.EXE, supplied with your development system, are device driver programs for line printers. They support LP11-type line printers attached to the printer/parallel port of the DMF-32 board or to an LPV11 printer interface, respectively.

The parallel port on a DMF-32 can be used either for a line printer or for parallel I/O, but not both simultaneously (see "Parallel I/O Support," later in this chapter).

The line printer driver must be included in any application that uses a line printer for output. (Note that several systems in a network can use the printer on one node.) For instructions on including it, see Chapter 13, "System Development."

You can open the line printer 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.

General Features of the Printer Driver

The driver generally has one program parameter, a device controller name you supply with the System Builder. It appends '0' to the controller name to form the local name of the printer unit itself. If you load the driver with an explicit program description, you can give the program a second argument, which it will append '0' to and establish as a universal name.

For example, if the name specified to the System Builder for the printer controller is 'LPA', the name you use in place of a file specification when opening the file on the same node is 'LPA0:'. If you also supplied a universal-name argument, such as 'PRINTER', you can access the printer with the universal name 'PRINTER0' from any node.

Alternatively, you can supply an explicit node specification in the file specification if the printer is not on the local node. 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; this means that the same DMF-32 cannot be used for parallel I/O.

Characteristics of the Printer Driver

The following characteristics are defined in the printer driver source files (LCDRIVER.PAS and LPVDRIVER.PAS) by Pascal named constants. For different behavior, you can change the characteristics, recompile the source files, and relink the drivers.

Maximum Record Length

This is the maximum length of single records written to the line printer. The standard value is 512 bytes (characters).

Lines per Page

This is the number of consecutive lines written per 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 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. TRUE would be used for printers that lack a mechanical form-feed feature.

Page Width

This is the maximum number of characters per printed line. The standard value is 132 characters.

Line Wrapping

A Boolean value specifies whether lines longer than the specified page width wrap automatically. The default is FALSE.

Lowercase to Uppercase Conversion

A Boolean value specifies whether lowercase characters are converted to uppercase on output. The default is FALSE. You can change the value to TRUE to have all letters printed in uppercase.

Nonprinting Character Handling

A Boolean value specifies whether nonprinting characters are allowed in the output. The default is TRUE.

Insertion of CR before LF

A Boolean value 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 output.) The default is FALSE.

Terminal Drivers

Device drivers are supplied with your development system for performing program I/O with the console terminal or with terminals attached to asynchronous line interfaces (the DMF-32, DHV11, DZQ11, and DZV11 interfaces).

For most applications, you can regard these device drivers as self-contained programs once they are included in your system. If you would like to study the programming methods used, see the source file YCDRIVER.PAS. The remainder of this section discusses the drivers' operational characteristics as seen by a programmer or terminal user, such as the meanings of control characters, escape sequences, and so forth.

The DMF-32 device interfaces up to eight asynchronous serial communication lines to a VAX target machine, for communication with terminals or other VAXELN systems. The DZQ11 and DZV11 devices interface up to four asynchronous lines to a MicroVAX target machine. The DHV11 device interfaces up to eight asynchronous lines to a MicroVAX target machine. (These are in addition to the target machine's console terminal.) Other features of the DMF-32 are supported by separate device drivers; for example, see "Parallel I/O Support" and "Printer Drivers," elsewhere in this chapter.

All data transmissions involving terminals are full duplex transmissions with the same speed (baud rate) for sending and receiving. In addition, the DMF-32, DHV11, DZQ11, and DZV11 can be used to communicate between remote VAXELN and VAX/VMS systems, as discussed under "Point-to-Point DDCMP Communication," later in this section.

Terminal I/O

Input and output to a terminal is accomplished by sending appropriate messages to message ports created by the Console Driver or other terminal driver.

The Console Driver handles transmissions between the program and the console terminal; the asynchronous line drivers handle transmissions between the program and one or more terminals attached to asynchronous serial interfaces. For instructions on describing individual terminals in your systems, see Chapter 13, "System Development."

The run-time code for VAXELN procedures (such as the Pascal READ and WRITE procedures) formulates and transmits the necessary messages implicitly when these procedures are called with reference to a terminal.

Type-Ahead and Synchronization

Input characters typed before an actual "read" request are buffered in a "type-ahead" buffer. The type-ahead feature allows you, for example, to 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, and 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 automatically with the terminal by means of the XON and XOFF control characters. This means that, for most applications, the terminal's AUTO XON/XOFF setting should be enabled in its set-up mode.

Line Terminators

Lines of input are terminated by typing a *line terminator,* which is a RETURN, CTRL/Z, or any other character with an ASCII code less than 32 (decimal), except those that have special interpretations as control characters (see "Control Characters," later in this section). When escape recognition is enabled, an entire valid escape sequence is treated as a line terminator, is not echoed, and is returned to the inputting program; this is the only case in which a line terminator also constitutes program input.

Point-to-Point DDCMP Communication

The DMF-32, DHV11, and DZV11 can also be used for error-free, though not transparent (as Ethernet is) communication between remote VAXELN and VAX/VMS systems. The user can establish a virtual circuit between jobs on remote machines over a serial line. This is accomplished by allowing each line to act as a full-duplex asynchronous point-to-point DDCMP communication link.

The DIGITAL Data Communications Message Protocol (DDCMP) is a data link control procedure that ensures a reliable data communication path between communication devices connected by data links. This DDCMP option is specified with the System Builder, on a terminal line by terminal line basis.

An example of a VAXELN serial DDCMP link is illustrated in Figure 10-1.

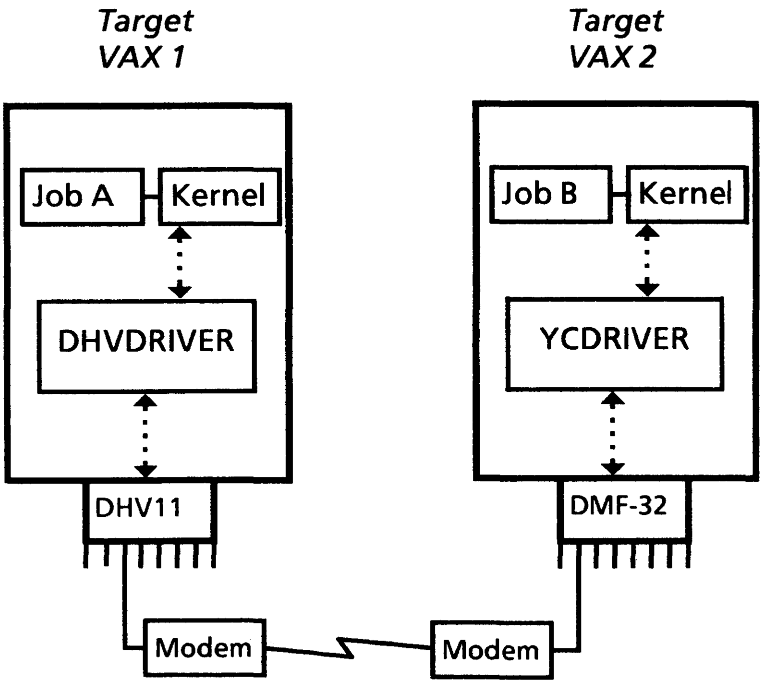


Figure 10-1. A VAXELN Serial DDCMP Link

A job starts the DDCMP protocol on a line by connecting a circuit to the driver handling the line; the job stops it by disconnecting from the circuit. In Figure 10-1, messages sent by Job A are received by Job B, and vice versa. For example, if Job A were to use line TTA2 on a DHV11, part of the Pascal program would be:

```
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 would look like:

```
  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 messages can be of any data type and any size up to a maximum of 1024 bytes, but cannot be zero bytes long. The maximum length is set as a program constant within the the $DDCMP module (see "Additional Support Routines," later in this section.) 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 will disconnect the circuit. Note that, because this is a full-duplex communication line, both jobs can be sending messages simultaneously.

The following limitations apply to DDCMP communication:

- Messages that are received are guaranteed to be received in the proper order and error-free. However, due to the nature of the DDCMP protocol, flow control is not as complete nor 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, additional messages are refused by the driver.
- If the receiver does not receive the messages within a timeout period of approximately 20 seconds (accounting for retransmissions and acknowledgements), the sending driver will stop the protocol and disconnect the circuit. To guard against 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, but prolonged uncontrolled sending of messages should be avoided.
- Only one virtual circuit is allowed per line.

Setting Terminal Characteristics with the System Builder

When a terminal driver is included in the system, the characteristics of the terminal on each line can be specified with the System Builder. You can also specify the characteristics of the console terminal on a separate menu. The characteristics you can specify, and their default settings, are as follows:

- *Terminal type* specifies, for terminals other than the console, the kind of asynchronous line interface in use. This setting is ignored on a DDCMP line.
- *Speed* specifies, for terminals other than the console, the baud rate for transmission and reception on the indicated line. Values are from a specific set in the range 50–38,400. If the rate is not specified, 1200 baud is the default for the console and hardcopy terminals (the console is assumed by default to be a hardcopy terminal), and 9600 baud is the default for other terminals. Be sure that the terminal is set to the same speed via its set-up mode.
- The *Parity* option enables (*Yes*) or disables (*No*) parity checking for the line. Parity checking is disabled by default. *Parity type* specifies, for terminals other than the console, the option *Odd* or *Even*, describing the kind of parity checking used by the connected terminal. If neither option is specified, *Even* is the default. Be sure that, in its set-up mode, the terminal's parity type and enablement are set properly.
- The type of terminal in use is specified by *Display type*. *Hardcopy* specifies that the terminal is a hardcopy device, such as an LA120 printing terminal, and is the default for the console terminal; otherwise, the default is *Scope*, meaning

a video terminal. *Scope* causes the DELETE key to backspace and rub out the deleted character; *Hardcopy* makes it rewrite the deleted characters enclosed in backslashes (\deleted characters\). This setting is ignored on a DDCMP line.

- *Escape recognition* specifies that, on input, the terminal driver checks the format of escape sequences to see whether they conform to American National Standards Institute (ANSI) format. For the correct formats, see "Escape Sequences," later in this section. In general, a terminal with escape recognition in effect should have its escape-sequence format set to ANSI (via its set-up mode). This setting is ignored on a DDCMP line.
- *Echo* controls whether the terminal displays (echoes) input lines it receives. If you do not specify this characteristic, the default is *Yes*. *No* means the terminal displays only the characters written to it by software. This setting is ignored on a DDCMP line.
- *Pass all* specifies that the driver passes all characters (including tabs, form feeds, and control characters except CTRL/X) directly, without interpretation or translation. With *Pass all* in effect, only fixed-length records can be read. *No* is the default option, meaning that special interpretations apply to certain characters (see, for example, "Control Characters," later in this section). This setting is ignored on a DDCMP line.
- *Eight-bit* specifies (*Yes*) that the high-order bit of an input character is not masked to zero by the terminal driver. If you select *No*, the high-order bits of all input characters are masked to zero. Note that the *Eight-bit* characteristic determines

the interpretation by software of input characters; the "bits per character" setting in a terminal's set-up mode governs the number of bits output by the terminal. This setting is ignored on a DDCMP line.

- *Modem* indicates, for terminals other than the console, that the line is connected to a modem or cable that supplies the standard (EIA) modem control signals. The default is *No*; in this case, any modem control signals are ignored. Modems can be used only on the DMF-32 and DHV11. With the DMF-32, only the first two of its eight lines can be used for modems. (See also "Modem Control," later in this section.)
- *DDCMP* indicates whether the line is a regular terminal line or a DDCMP line. The default is *No*; in this case, a regular terminal line.

Control Characters

Unless the *Pass all* characteristic is in effect or *DDCMP* is specified, control characters identify special actions to be performed by the driver rather than actual characters to be sent to the program. All control characters have ASCII codes in the range 0–31 or equal to 127 (DELETE). The full set is shown in Table 10-3.

The characters designated CTRL/*x*, where *x* is a letter, are generated by holding down the CTRL key on the keyboard while pressing key *x*.

In some cases, when *Echo* is in effect, the character CTRL/*x* is echoed as a caret followed by the character *x*, for example, ^U for CTRL/U.

Table 10-3. Control Characters

Terminal Key or Name	Code	Meaning
"Bell"	7	Sound bell or buzzer on terminal.
BACK SPACE	8	Back up cursor one character (note: *does not* delete the previous character from input).
TAB, CTRL/I	9	Advance to next (horizontal) tab stop (tab placement is controlled by the terminal).
LINE FEED,CTRL/J	10	Advance to next line (without carriage return).
CTRL/K	11	Advance to next (vertical) tab stop (tab placement is controlled by the terminal).
CTRL/L	12	Advance to next page or display (form feed) and terminate current input line.
NO SCROLL[1], CTRL/Q	17	Resume transmitting output from program.
CTRL/R	18	Redisplay current input line.
NO SCROLL[1], CTRL/S	19	Suspend transmitting output from program.
CTRL/U	21	Erase current input line.
CTRL/X	24	Erase type-ahead buffer and current input line.[2]
CTRL/Z	26	Designate end-of-file to program; terminate current input line.
DELETE	127	Delete previous character or (if escape recognition is in effect) partial escape sequence from input.
ESC	27	Begin escape sequence if *Escape recognition* is in effect; otherwise, echo as $, perform carriage return and line feed, and terminate current input line .
ENTER[3], RETURN	13	Perform carriage return and line feed; terminate current input line.

[1]The key NO SCROLL, on VT100-type terminals, alternates between CTRL/S (for the first and other odd-numbered keystrokes) and CTRL/Q.

[2]The function of the CTRL/X key is the same whether or not *Pass all* is in effect.

[3]The key ENTER, on the keypad of VT100 and similar terminals, is normally the same as RETURN.

Escape and Control Sequences

When escape recognition is in effect, and it is a regular terminal line, you can read escape sequences from a terminal with syntax checking performed by the terminal driver. In all cases, whether or not escape recognition is in effect, you can write out escape sequences to perform actions specific to the terminal. (For example, the VT100 and VT200 series terminals allow you to control the movement of the cursor with escape sequences.)

The syntax of escape sequences is checked only on input and only when escape recognition is in effect. Only ANSI-format escape sequences, such as used with the VT100 and VT200 series, are recognized on input. (For the set of escape sequences used with a particular terminal, see its hardware documentation, such as the *VT100 User Guide*.)

When escape recognition is in effect, any sequence of input characters beginning with the ESC character (ASCII code 27) is checked by the terminal driver to determine whether it is syntactically valid. 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 (note that there is no actual space between the syntax elements):

```
ESC character-sequence final-character
```

character-sequence. This is a sequence of zero or more characters, each of which has an ASCII code in the range 32–47. This range consists of the space character and 15 punctuation marks.

final-character. The final character is a single character which has an ASCII code in the range 48–127, which includes uppercase and lowercase letters, digits, and an assortment of punctuation marks. The following alternate forms are permitted:

ESC ; character-sequence final-character

ESC ? character-sequence final-character

ESC O character-sequence final-character

where, with ESC O, the final character can have an ASCII code in the range 64–127. The character sequence is the same in all cases. (The eight-bit character SS3 [$8F_{16}$] can be used to introduce an escape sequence, in lieu of ESC O.)

Also valid are ANSI *control sequences,* in which the character sequence and final character are preceded by a left bracket ([) and a sequence of parameter specifiers (note that there is no actual space between the syntax elements):

ESC [param-sequence char-sequence final-char

The eight-bit character CSI [$9B_{16}$] can be used to introduce an escape sequence, in lieu of ESC [.

param-sequence. A "parameter" sequence consists of zero or more parameter specifiers, each of which has an ASCII code in the range 48–63. For instance, for some control sequences on VT100 and VT200 series terminals, this is a sequence of digit characters separated by semicolons.

char-sequence. This is a sequence of zero or more characters, each of which has an ASCII code in the range 32–47.

final-char. The final character is a single character which has an ASCII code in the range 64–127.

For example, the following control sequence erases from the current cursor position to the end of the line on a VT100 terminal:

```
ESC[0K
```

where 0 is a parameter and K is the final character.

The following sequence turns on the "bold" and "reverse video" character attributes on a VT100 terminal:

```
ESC[1;7m
```

where 1 and 7 are parameters, separated by a semicolon, and m is the final character.

VT52-Type Escape Sequences

The VT52 terminal uses escape sequences that do not comply fully with the ANSI format. VT100 and VT200 terminals allow you to designate, in the terminal's set-up mode, that it will use VT52 escape sequences instead of the larger ANSI set supported on that terminal type.

We recommend that you use ANSI escape sequences whenever possible. However, most VT52 escape sequences are actually 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:

```
ESCJ
```

where, 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 48–127.

Modem Control

Modems allow you to connect telephone or other remote lines to the terminal interface, for access to the target computer from remote terminals. The DMF-32 and DHV11 terminal drivers support modem control (for example, of DEC DF03, DEC DF100, Bell 103a, Bell 113, and equivalent modems) in full-duplex, autoanswer mode. Of the eight asynchronous lines on a DMF-32, only the first two can be connected to modems.

The modem itself is controlled by a set of signals it exchanges with the target computer. All transmission and interpretation of these signals is done for you by the driver; they are discussed here to help you decide whether the modem you have is usable.

The signals are shown in Table 10-4, and all must be supported by the modem in question.

When modem control is enabled for a terminal line, the line is monitored continually (by the interface hardware) for the RING signal. If the CARRIER and DSR signals are then detected by the driver, 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 two seconds for it to reappear and, if it does not, returns an error to any current or future read request if the CARRIER signal does not come back.

Table 10-4. Modem Control Signals

Signal Name	Source	Meaning
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.
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, 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.

Additional Support Routines

The modules $TERMINAL and $DDCMP in library RTLOBJECT.OLB contain several support routines that are useful if you are writing Pascal terminal drivers or serial DDCMP line drivers. You can use these declarations by including the modules in compilations of your Pascal terminal drivers or serial DDCMP line drivers. For details, see the module's source file, TERMINAL.PAS, and any of the terminal driver source files, such as DZVDRIVER.PAS.

Parallel I/O Support

The Pascal source file DR11C.PAS, supplied with your development system, contains declarations of the DMF-32 device registers suitable for using the device's parallel port for digital input and output.

The parallel port on a DMF-32 can be used either as a line printer port or to send and receive up to 16 bits of data on 16 parallel lines. The type and variable declarations in DR11C.PAS can be used as delivered in programs you write to perform parallel I/O.

DR11C.PAS is intended as a "template" that you can modify to make the program you need. In some cases, you can simply add a PROGRAM block that uses the declarations the module provides. In addition to the register declarations, the module provides templates for the following:

- An interrupt service routine and communication region.
- An initialization procedure that creates DEVICE objects representing the devices "request A" and "request B" lines, as well as initializing the

parallel port for digital I/O (instead of for a line printer).

- Input and output procedures to read and write a 16-bit word of data from the device.

Real-Time Device Drivers

The VAXELN development system includes device drivers for the following real-time devices:

- The ADV11C or AXV11C analog-to-digital converter.
- The KWV11C programmable, real-time clock.
- The DLVJ1 asynchronous serial line controller.
- The DRV11–J parallel line interface device.

The design of these drivers prohibits accessing a given device from more than one job. However, gaining access from different processes within the same job is possible, provided the caller ensures there is no simultaneous access to the same device.

Analog-to-Digital Converter

The Pascal module $AXV_UTILITY, supplied with your development system, defines the procedures provided to interface with the ADV11C analog-to-digital converter and the AXV11C. The AXV11C provides all of the functionality of the ADV11C and two digital-to-analog outputs as well.

By means of a hardware jumper, an ADV11C device can be configured to have 8 or 16 input channels. In the former case, analog voltage is measured across two input channels; in the latter case, 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 KWV11C clock option (see the next subsection, "Real-Time Clock").

You can only access an AXV11C from one job, which must be running in kernel mode. This job can be an "AXV11C 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 real-time devices. Because the KWV11C clock can be used in conjunction with an AXV11C device, some types used in $AXV_UTILITY are defined in the module $KWV_UTILITY.

The $AXV_UTILITY module provides the following procedures:

- AXV_INITIALIZE, which causes an ADV11C or AXV11C device to be readied for input and/or output 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).

- AXV_READ, which 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 for each specified channel is performed. The process is repeated until all data has been collected. This procedure may be called for either an ADV11C or AXV11C device.
- AXV_WRITE, which 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 AXV11C device.

The procedures described above return optional status values. In the interest of good real-time response, the procedures provide limited error checking; they only report errors detected by the device. No input parameters are verified and kernel service calls made in the course of execution will raise exceptions upon failure.

Call formats and detailed argument descriptions for the AXV11C support routines are provided in the *VAXELN Pascal Language Reference Manual* and the *VAXELN C Run-Time Library Reference Manual*, as appropriate to the programming language in use.

Real-Time Clock

The Pascal module $KWV_UTILITY, supplied with your development system, defines the procedures provided to interface with the KWV11C real-time clock. The KWV11C is a programmable, real-time clock that may be used to initiate action after a specified time interval (via an interrupt or an external signal) or to time an event. In the first mode, it may be used in conjunction with an ADV11C or AXV11C device to initiate the collection of data.

The device's clock counter has a resolution of 16 bits. It 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, may be used to start the counter. (A Schmitt Trigger is a logic device that responds to voltage levels rather than voltage transitions. The *LSI-11 Analog System User's Guide* contains more information on the hardware.)

The driver interface provided for the KWV11C is of the same style as that provided for the ADV11C, described previously. In fact, the major motivation for providing the KWV11C driver is to allow you to use all of the functionality of the ADV11C.

The design of this driver precludes accessing a given KWV11C 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 KWV11C 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:

- KWV_INITIALIZE, which causes a KWV11C 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 KWV11C; 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).

- KWV_READ, which 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 KWV_WRITE.
- KWV_WRITE, which 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 subsequently stop and read it using KWV_READ.

The procedures described above return optional status values. In the interest of good real-time response, the procedures provide limited error checking; they only report errors detected by the device. No input parameters are verified and kernel service calls made in the course of execution will raise exceptions upon failure.

Call formats and detailed argument descriptions for the KWV11C support routines are provided in the *VAXELN Pascal Language Reference Manual* and the *VAXELN C Run-Time Library Reference Manual*, as appropriate to the programming language in use.

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 QBUS interface that contains four asynchronous serial line channels. The channels can be independently configured for EIA RS-422, RS-423, or RS-232C signal compatibility. Provisions are also made for configuring the channels for 20 mA current loop operation.

Four independent serial line interfaces exist with consecutive bus device address and vector assignments that can be user-configured via wire-wrap jumpers on the module. Each serial line can also be independently configured for baud rates of 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), and parity (none, even, or odd). All of these configuration parameters are also set via wire-wrap jumpers on the conroller 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 real-time applications that collect data and control real-time devices using asynchronous serial lines. This is in contrast to the support provided for the DZV11, DZQ11, and DHV11, which is intended to provide a more functional interface for reading and writing via standard Pascal and C I/O routines to terminals connected via 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 real-time devices. The $DLV_UTILITY module also exports definitions of the DLV11'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:

- DLV_INITIALIZE, which readies 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.
- DLV_READ_BLOCK, which 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 DLV_INITIALIZE.
- DLV_READ_STRING, which 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 DLV_INITIALIZE.
- DLV_WRITE_STRING, which 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 Language Reference Manual* and the *VAXELN C Run-Time Library Reference Manual*, as appropriate to the programming language in use.

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 QBUS interface that provides communication between a MicroVAX and up to four user devices in 16-bit word lengths via 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 via either polling or interrupt-driven routines. Write data is output by the DRV11–J to the I/O bus through 3-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 real-time applications that collect data and control real-time devices using parallel lines. This is in contrast to the support provided for other, non-real-time devices, which are intended to provide a more functional interface for reading and writing via standard Pascal and C I/O routines to terminals connected via the serial lines.

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 DRVUTIL.PAS and DRVBODY.PAS can also serve as a model for other drivers for real-time devices. The $DRV_UTILITY module also exports definitions of the DRV11's device registers if it is desirable to directly read and write the registers. (See the DRVUTIL.PAS 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 using a dynamically allocated buffer array. The array is a two-dimensional array: the first array index specifies the parallel port number (0..3) and the second array index specifies a data word. The procedures internally utilize a separate DEVICE object per parallel port. Therefore, a user program can have interrupt driven I/O in progress on each port simultaneously. For example, an application program may 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 may 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 that either 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:

- DRV_INITIALIZE, which readies 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.
- DRV_READ, which 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 DRV_INITIALIZE.

- DRV_WRITE, which causes data words to be written to the specified parallel port. Before calling this procedure, the data words should be stored in the buffer pointed to by the buffer parameter returned by DRV_INITIALIZE.

Call formats and detailed argument descriptions for the DRV11-J support routines are provided in the *VAXELN Pascal Language Reference Manual* and the *VAXELN C Run-Time Library Reference Manual*, as appropriate to the programming language in use.

Chapter 11
Exception Handling

This chapter discusses VAXELN exceptions and exception handling procedures, including a brief overview of the VAX stack architecture. Status codes are explained and message processing features that handle the conversion of status codes into message text are discussed.

Exceptions in VAXELN

The term *exception* is commonly used to describe programming events that occur during the execution of a particular program. Exceptions can be:

- Synchronous. In this case, the exception would occur at the same particular place in the program given a set of circumstances; for example, dividing by zero.
- Asynchronous. In this case, the exception is triggered by some 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, "divide by zero" 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 run-time 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.

In the event of an exception there are only two options: ignore it or handle it. An exception might or might not be important for a program to be aware of. In addition, an exception might or might not be expected. It is up to the programmer to decide if a particular problem or exception condition is important or "fatal" to the program execution.

The mechanism for notifying a running program of an exception is that the VAXELN kernel exception processing software temporarily stops the normal execution of the program and calls 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 reasons of transportability, the VAXELN exception mechanism is almost identical to the exception mechanism used on VAX/VMS.

VAX Stack Architecture

Before explaining the exception handler mechanism, a brief description of the VAX *stack* architecture is important.

Whenever a program is executing on a VAX, the hardware registers SP and FP 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' memory, the P1 region. (See Chapter 3, "Processes and Jobs," 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. They are helpful in performing recursive operations, but in many cases they are best used as a "trail of breadcrumbs" to record 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 like PUSHAL, which pushes an address on the stack. The action of *pushing* is a two-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. In the case of PUSHAL, a longword is placed on the stack. In other contexts, different-sized objects are pushed or popped from the stack. Note that VAX stacks grow downward in address as they expand and there is no implication on the alignment of SP on a particular memory length boundary, although some instuctions (like 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 point in 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 which called it. At the same time, the value of the SP register is equal to or "less than" (below) the FP value. The memory between the SP and FP values is referred to as the *local storage* of the procedure activation. Both together are referred to as the procedure's *stack frame,* as illustrated in Figure 11-1. Languages like Pascal, C, and FORTRAN use this space to store procedure temporaries or variables.

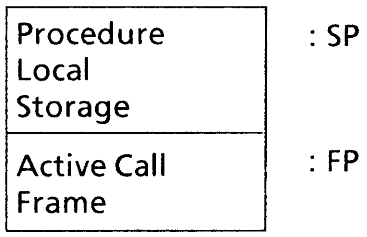


Figure 11-1. A Procedure's Stack Frame

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 11-1, if a procedure call is performed, the stack would look like Figure 11-2.

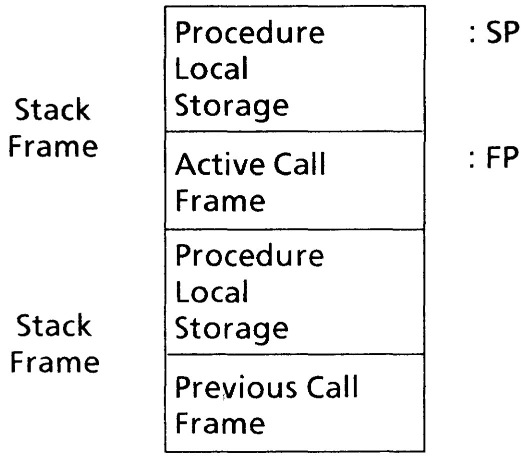


Figure 11-2. Frame Structure after Procedure Call

Internally, the call frame block looks like Figure 11-3.

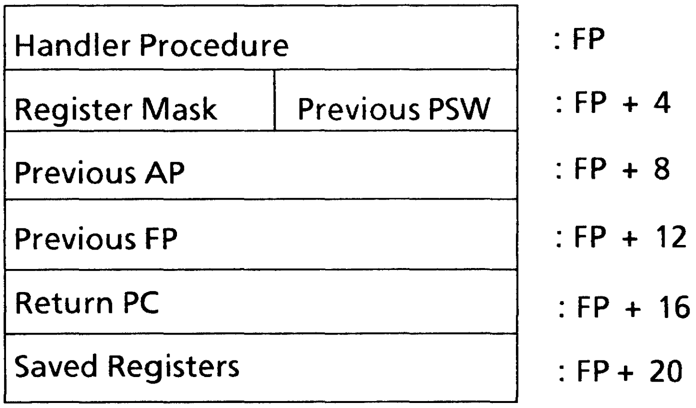


Figure 11-3. Call Frame Block

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 Procedure location is either zero or the address of an established condition handler procedure. (For a more detailed description of the frame contents, see the *VAX Architecture Handbook*.)

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.

Exception Handler Arguments

When an exception occurs, the VAXELN kernel exception logic builds an argument list that describes the exception. It then searches the current list of stack frames to find a frame that contains a non-zero condition handler address. When one is found, the handler procedure is called.

If no handler is found, the kernel takes some default action. That is, if the debugger is present in the system, a special debugger handler is called. It 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 process is deleted by the kernel.

The argument list for an exception handler routine contains two values. The first argument value is the address of another data block containing information about the exception that occured. This block is the *signal* argument block. The signal arguments are illustrated in Figure 11-4.

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	

Figure 11-4. Signal Arguments

Each different exception has a distinct argument list that provides information about the exception. Sometimes, as in the case of "divide by zero," no additional information is needed or present. Such exceptions have the same names as the corresponding status values, as described in Appendix C, "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 11-5.

Note that the frame depth value is the number of frames searched while looking for the exception handler address.

4	: Mechanism arguments
FP where the handler was found	: Mechanism arguments + 4
Frame depth	: Mechanism arguments + 8
R0 at exception	: Mechanism arguments + 12
R1 at exception	: Mechanism arguments + 16

Figure 11-5. Mechanism Arguments

"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); this is then the way to exit an exception handler. In this case, the language run-time library does an implict continue on behalf of the program.

As explained previously, if no handler is found that handles the exception, the kernel deletes the process with the exception name as the status. Each potential exception has an individual status code defined for it (see "Status Codes," later in this chapter). The exception name value can be used to associate a descriptive text message with the status code, as explained in "Message Files and Utilities," later in this chapter.

A particular exception handler may handle one or more individual exception conditions. Some programs have handlers that "catch 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 particular 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.

"Unwind" Operation

As mentioned previously, some languages provide an explicit method for exiting the condition handler. This has the effect of continuing at a different location and possibly in a different stack environment. This operation is called *unwinding*. There is a VAXELN kernel procedure to perform this operation because the stack discipline and modification is complex. Normally, the unwind is done for you as a result of exiting an exception handler.

If you use the UNWIND procedure directly, it provides several options. There are two parameters. One is a new continue PC, which specifies an alternative continuation point. If a PC is not specified, some frame's return PC is used. The other argument specifies which frame is to be the active one after the unwind. It can be specified as either a "depth" from the exception frame or a particular FP value. (Note that a depth of zero is treated as a special case in VAXELN, unwinding to the caller of the establisher; this is in contrast to VMS, where a depth of zero means "unwind zero pre-signal frames.")

The UNWIND procedure has the effect of returning back through some number of subroutines without actually executing any code in the subroutines that are skipped.

Unwind 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 the 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, specifying "continue." It can, however, perform its own unwind operation, overriding the unwind operation in progress.

One final feature can be used when an unwind is performed. Most procedures, if they return a simple value, return that value in R0 and R1. Most VAX languages adhere to this standard. It is possible, therefore, to 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. Care is required to understand the semantics of function values in a particular language.

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 already 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.

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.

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.

Asynchronous Exceptions

Asynchronous exceptions do not occur as a result of some 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 "continue" is specified. Note, however, that 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.

There are several types of asynchronous exceptions generated by VAXELN:

- KER$_POWER_SIGNAL. If a job is specified during system build as desiring power recovery signals, the kernel will generate an exception when the power recovery takes place.
- KER$_QUIT_SIGNAL. Signal of a process object causes the target process to receive this exception.
- KER$_PROCESS_ATTENTION. This exception is caused by a process calling the kernel procedure RAISE_PROCESS_EXCEPTION.

Exception Handling Procedures

The kernel procedures relating to exception handling are summarized below.

DISABLE_ASYNCH_EXCEPTION Procedure

DISABLE_ASYNCH_EXCEPTION prevents the delivery of asynchronous exceptions to the calling process.

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.

RAISE_EXCEPTION Procedure

RAISE_EXCEPTION causes a particular software exception in the calling process. A list of zero or more additional exception arguments can be specified, 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); take care not to raise one of these exceptions.

RAISE_PROCESS_EXCEPTION Procedure

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

UNWIND Procedure

The 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.

Status Codes

All status codes returned by VAXELN routines follow the usual VAX convention in which odd-numbered integers signify success and even values mean failure of some kind (not necessarily fatal). Specifically:

- Bits 0–2 define the severity: 0 means warning, 1 means success, 2 means error, 3 means "informational," and 4 means "severe" or "fatal" error.
- Bits 3–31 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" implies that it is not.

Note: For the exit status of a process, you can return any integer you like, although we suggest following the above convention.

The creator of a job has the option of receiving a special "termination" message when the created job completes. This message simply contains an integer making up the completion (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 (success).

Caution: The successful completion of a process may be represented by more than one exit status (for example, status code 1 or 3). Therefore, to check for success in your programs, we suggest that you always check for the specific status value KER$_SUCCESS, rather than for a status code of 1. A process will always return KER$_SUCCESS upon successful completion, even if the status code returned is not 1.

Message Files and Utilities

The VAX/VMS system contains message processing features to help programs handle the conversion of status codes into meaningful message text. The feature is supported by two components on VMS:

- A message-processing utility program called MESSAGE
- A message system service, $GETMSG

The MESSAGE utility acts much like a compiler. It transforms a source file containing the definitions of status codes and text into various forms, including object files. These object files can then be referenced from program images.

The VMS system service, $GETMSG, given the status code, extracts the text information from a database created from the MESSAGE utility object files.

VAXELN provides a run-time library routine that duplicates the action of the $GETMSG system service. By using this routine, a VAXELN application can also access the message text information.

Including the Message Text in a VAXELN Application

Use the VMS MESSAGE utility to create a message object file, and simply include that message object file when you link your program. For instance:

```
$ MESSAGE /OBJ msgdef   ! create the message file
$ !
$ ! Link the program
$ !
$ LINK/NOSYSSHR app + msgdef + -
ELN$:RTLSHARE/LIB + RTL/LIB
```

The VMS MESSAGE utility is fully documented in the VAX/VMS documentation.

Accessing the Message Database During the Execution of a VAXELN Application

Two message processing routines are provided by the VAXELN run-time libraries. One is an exact duplicate of the VMS $GETMSG routine. It is called SYS$GETMSG, just as is the VMS system service. The parameters for SYS$GETMSG are identical to the description outlined in the system service documentation for VAX/VMS.

The other routine, GET_STATUS_TEXT, is provided for easier use with higher-level languages. This procedure returns the text associated with a status code that you provide as input to the routine. A format-control parameter can be provided so that the returned string contains only a part of the information available.

The VAXELN Pascal definitions for this routine are provided in the $GET_MESSAGE_TEXT module of RTLOBJECT.OLB. The functionally equivalent definitions for C programs are included in the #include module named $GET_MESSAGE_TEXT contained in ELN$:VAXELNC.TLB.

Message Files Provided with the VAXELN Kit

The VAXELN run-time library RTL.OLB contains the following message object modules that can be linked with your program to provide VAXELN-specific messages for use with the GET_STATUS_TEXT procedure:

- ELN$MSGDEF_TEXT. Contains messages generated by the VAXELN Pascal compiler and other utilities (such as the File Service).
- KER$MSGDEF_TEXT. Contains messages generated by the VAXELN kernel.
- PAS$MSGDEF_TEXT. Contains messages generated by the VAXELN Pascal run-time routines.
- C$MSGDEF_TEXT. Contains messages generated by the VAXELN C run-time routines.

The corresponding source files (for the VAX/VMS MESSAGE utility) also are provided: ELNMSG.MSG, KERNELMSG.MSG, PASCALMSG.MSG, and CMSG.MSG, respectively. These sources show the text of a message and its corresponding ID (name).

Included in RTL.OLB are ELN$MSGDEF, KER$MSGDEF, PAS$MSGDEF, and C$MSGDEF; these modules define the message names as linker global values for use from non-VAXELN Pascal programs.

The image ELNMSG.EXE is placed by the VAXELN installation procedure in the VAX/VMS directory SYS$MESSAGE. It is provided for use when you are executing VAXELN programs under VMS.

Note that the Pascal messages are not included here since they are provided by the VMS environment, and the kernel messages do not occur since programs cannot call kernel procedures when executing under VMS.

In addition, the image ELNCMSG.EXE is placed by the VAXELN installation procedure in the VAXELN directory ELN$. It provides the message text for the VAXELN C run-time routines.

You can use these images with the VMS command SET MESSAGE to find out the VAXELN message corresponding to the hexadecimal value (a "reason mask" or "reason value") that is reported in some contexts for exceptions (for instance, the local debugger). For example:

```
$ SET MESSAGE SYS$MESSAGE:ELNMSG
$ EXIT %xHHHHHHHH
```

or

```
$ SET MESSAGE ELN$:ELNCMSG
$ EXIT %xHHHHHHHH
```

where the Hs are the hexadecimal digits of interest.

During a debug session, you can use the VAXELN debugger command SHOW MESSAGE to display the text associated with a value's exit status (see Chapter 15, "Debugging"). For example, the command

```
Edebug 4,5> SHOW MESSAGE %x7C3C
```

displays the text associated with the status %x7C3C, that is:

```
KER$_BAD_VALUE, Bad parameter value
```

Chapter 12
Program Development

This chapter summarizes the use of the VAXELN Pascal and VAX C compilers and the VAX/VMS librarian and linker to prepare programs for inclusion in a VAXELN system. Program images are prepared with the VAX/VMS commands EPASCAL, CC, LIBRARY, and LINK. The program images are then combined into a system by the System Builder, as explained in Chapter 13, "System Development."

Preparing a VAXELN Program

Programs written in VAXELN Pascal are compiled with a command of the form

```
$ EPASCAL myfile
```

which produces the object module myfile.obj from the VAXELN Pascal source file myfile.pas. Assuming that myfile.pas specifies the complete program, the following LINK command prepares its image:

```
$ LINK myfile + ELN$:RTLSHARE/LIB + RTL/LIB
```

Programs written in VAX C are compiled with a command of the form

```
$ CC myfile + ELN$:VAXELNC/LIB
```

which produces an object module from the C source file. C programs are then linked with the command:

```
$ LINK myfile + ELN$:CRTLSHARE/LIB + RTL/LIB
```

Note that programs written in either language are linked with RTL/LIB.

In both Pascal and C, if myfile specifies only part of the program, you can create a library and insert the object module in it for input to a later compilation or use in linking:

```
$ LIBRARY/CREATE objectlib myfile
```

The formats of the EPASCAL command and the CC command, and the corresponding command arguments, are discussed in detail in the *VAXELN Pascal Language Reference Manual* or the *VAXELN C Run-Time Library Reference Manual,* as appropriate to the programming language in use.

The formats of the LIBRARY and LINK commands are discussed in the remainder of this chapter.

LIBRARY Command

The VAX/VMS librarian maintains libraries of modules, including object modules. It is used to maintain libraries of object modules for use as VAXELN Pascal compiler input or linker input and to maintain libraries of text modules for use in C program compilations.

For information about the run-time object libraries supplied with the system, see the description of the LINK command, later in this chapter.

The following examples illustrate use of the librarian for common operations in VAXELN development. For more information about the LIBRARY command, see the VAX/VMS documentation or on-line HELP for your VAX/VMS system.

Creating a New Library

$ LIBRARY/CREATE library-spec ***object-file-spec-list***

Here, a new object library is created; if you specify a list of object files, their modules are inserted in the new library.

Inserting or Replacing Modules in an Existing Library

```
$ LIBRARY library-spec object-file-spec-list
```

The library-spec is a VAX/VMS file specification that designates an object-module library. Such libraries have the default file type OLB and are the default library type for the librarian. Here, a list of object file specifications, separated by commas, designates the modules to be added. If any modules of the same names are already in the library, they are replaced with the new ones.

Listing a Library's Contents

$ LIBRARY/LIST = ***listing-file-spec*** **library-spec**

Here, a listing of the library's contents (module names) is written to the specified listing file; if the listing file specification is omitted, the listing is written to SYS$OUTPUT (usually, your terminal).

Extracting Modules from a Library

```
$ LIBRARY/EXTRACT = (module-list)-
_/OUTPUT = (file-spec-list)  library-spec
```

Here, the listed modules (whose names are separated by commas in the module-list) are extracted from the library; new files are created to receive them, with the default file type OBJ and the same names as the modules'. (The qualifier OUTPUT = (file-spec-list) is used to name the output files explicitly).

Deleting Modules from a Library

```
$ LIBRARY/DELETE = (module-list) library-spec
```

Here, the listed modules are deleted from the library.

Compressing a Library

```
$ LIBRARY/COMPRESS library-spec
```

Here, the specified library is compressed; unused space resulting from module deletions is recovered. By default, a new library (OLB) file is created with the same specification you supplied, but a version number one higher; you can use the qualifier OUTPUT = file-spec to specify the output file explicitly.

LINK Command

The VAX/VMS linker produces program images by combining object modules. The program images are then ready for inclusion in VAXELN systems (see Chapter 13, "System Builder").

For example, the command:

```
$ LINK myfile,myfile2,-
_ELN$:RTLSHARE/LIBRARY,-
_RTL/LIBRARY
```

produces the image file myfile.exe, ready for inclusion in a VAXELN system, from the object files myfile.obj and myfile2.obj, with references from the object modules resolved to the object module library RTL.OLB and the shareable image library RTLSHARE.OLB. The command format, libraries, and qualifiers are discussed below.

Format

$ LINK file-specification-list

The LINK command itself, and individual file specifications, accept a large set of qualifiers, each of which is a word preceded by a slash (/). For a discussion of qualifiers that are commonly useful with VAXELN, see "Qualifiers," later in this section.

File Specifications

Each file specification specifies either an object module created by the compiler, an object library, or a shareable image library. The file specifications are separated by commas or plus signs. In either case, all specified input files are used to create a single program image.

The first file specification must not be a library unless the INCLUDE qualifier is used to specify which of its modules to use (see "Qualifiers," later in this section).

Object module files have the default type OBJ. Object and shareable image libraries have the default file type OLB and must be written with the LIBRARY or INCLUDE file qualifier.

VAXELN Libraries

The following run-time libraries are supplied for linking with VAXELN Pascal and C object modules. All are placed in the directory ELN$ by the installation procedure (see the *VAXELN Installation Manual*). You can obtain a list of any library's contents with the LIBRARY command described previously.

RTLSHARE.OLB

This is a shareable image library containing the following shareable images:

DAP.EXE. This is the shareable image for the Data Access Protocol (DAP) routines used by device drivers.

DDCMP.EXE. This is the shareable image for the DIGITAL Data Communications Message Protocol (DDCMP) routines used by device drivers.

DISK.EXE. This is the shareable image for the disk utility procedures.

DMATH.EXE. This is the shareable image for the Pascal math routines using D_floating instructions.

DPASCALIO.EXE. This is the shareable image for the run-time support for all Pascal I/O routines using D_floating instructions, with the exception of the DAP interface.

FILE.EXE. This is the shareable image for the disk File Service.

FILEUTIL.EXE. This is the shareable image for the file utility procedures.

GMATH.EXE. This is the shareable image for the Pascal math routines using G_floating instructions.

GPASCALIO.EXE. This is the shareable image for the run-time support for all Pascal I/O routines using G_floating instructions, with the exception of the DAP interface.

NETWORK.EXE. This is the shareable image for the Network Service.

PASCALMSC.EXE. This is the shareable image for the miscellaneous Pascal run-time library routines.

PRGLOADER.EXE. This is the shareable image for the dynamic program loader utility procedures.

TAPE.EXE. This is the shareable image for the tape File Service.

TERMINAL.EXE. This is the shareable image for routines used in writing terminal drivers.

CRTLSHARE.OLB

This is a shareable image library containing the following shareable images:

CMSC.EXE. This is the shareable image for miscellaneous (floating-point independent) C run-time library routines.

DCIO.EXE. This is the shareable image for the C run-time library I/O routines using D_floating instructions.

DCMATH.EXE. This is the shareable image for the C run-time library math routines using D_floating instructions.

GCIO.EXE. This is the shareable image for the C run-time library I/O routines using G_floating instructions.

GCMATH.EXE. This is the shareable image for the C run-time library math routines using G_floating instructions.

RTLOBJECT.OLB

This is an object library containing object-module versions of everything in RTLSHARE.OLB. A number of the RTLOBJECT modules are available for inclusion in the compilation of your Pascal programs. In several cases, the corresponding source files are also supplied for your use and inspection. Inclusion of modules from this library is required to use several of the VAXELN features and procedures that are not predeclared.

$AUTHORIZE_UTILITY. This contains the definitions of the AUTH_ADD_USER, AUTH_MODIFY_USER, AUTH_REMOVE_USER, and AUTH_SHOW_USER routines.

$AXV_UTILITY. This contains the definitions of the AXV device driver utility routines AXV_INITIALIZE, AXV_READ, and AXV_WRITE.

$CMSG. This contains the Pascal-compatible status values returned by the C run-time code.

$DAP. This contains definitions used in writing device drivers using the Data Access Protocol.

$DATALINK. This contains definitions used in writing datalink drivers.

$DDCMP. This contains various routines useful for writing serial DDCMP line drivers (and used in the drivers supplied with your development system).

$DISK_UTILITY. This contains the definitions of the routines DISMOUNT_VOLUME, INIT_VOLUME, and MOUNT_VOLUME.

$DLV_UTILITY. This contains the definitions of the DLV device driver utility routines DLV_INITIALIZE, DLV_READ_BLOCK, DLV_READ_STRING, and DLV_WRITE_STRING.

$DRV_UTILITY. This contains the definitions of the DRV device driver utility routines DRV_INITIALIZE, DRV_READ, and DRV_WRITE.

$ELNMSG. This contains the definitions of VAXELN-specific messages.

$EXIT_UTILITY. This contains the definitions of the exit utility routines DECLARE_EXIT_HANDLER and CANCEL_EXIT_HANDLER.

$FILE_UTILITY. This contains the definitions of the file utility routines CREATE_DIRECTORY, COPY_FILE, DELETE_FILE, DIRECTORY_OPEN, DIRECTORY_CLOSE, DIRECTORY_LIST, PROTECT_FILE, and RENAME_FILE.

$GET_MESSAGE_TEXT. This contains the definitions of types and routines used to process messages.

$KERNEL. This contains the definitions of kernel procedures that are not predeclared in VAXELN.

$KERNELMSG. This contains the definitions of kernel procedure status values.

$KWV_UTILITY. This contains the definitions of the KWV device driver utility routines KWV_INITIALIZE, KWV_READ, and KWV_WRITE.

$LOADER_UTILITY. This contains the definitions of the progam loader routines ELN$LOAD_PROGRAM and ELN$UNLOAD_PROGRAM.

$MUTEX. This contains the definitions of the MUTEX data type and related procedures CREATE_MUTEX, INITIALIZE_AREA_MUTEX, DELETE_MUTEX, LOCK_MUTEX, and UNLOCK_MUTEX.

$PASCALMSG. This contains the status values returned by the Pascal I/O run-time code.

$PHYSICAL_ADDRESS. This contains the definition of the PHYSICAL_ADDRESS function.

$STACK_UTILITY. This contains the definitions of the stack utility routines ALLOCATE_STACK and DEALLOCATE_STACK.

$TAPE_UTILITY. This contains the definitions of the MOUNT_TAPE_VOLUME, INIT_TAPE_VOLUME, and DISMOUNT_TAPE_VOLUME routines.

$TERMINAL. This contains various routines useful for writing Pascal terminal drivers (and used in the drivers supplied with your development system).

$UNIBUS. This contains the definitions of routines useful in writing drivers for UNIBUS or QBUS DMA devices: UNIBUS_MAP, LOAD_UNIBUS_MAP, and UNIBUS_UNMAP.

$VAX_DEFINITIONS. This contains the definitions of the VAX$PSL and VAX$VA routines.

CRTLOBJECT.OLB

This is an object library containing object-module versions of everything in CRTLSHARE.OLB.

RTL.OLB

This is an object library that contains modules for run-time routines that have local read/write data and a module defining entry points for the kernel procedures. This library is normally included in every LINK operation. It must follow RTLOBJECT.OLB in the LINK command if RTLOBJECT.OLB is used.

FILE.OLB

This is the object-module library containing the File Service modules and support routines. It is needed *only* when you are writing a driver that uses the File Service and you want to link the driver with local copies of these modules instead of shareable images (for debugging purposes, perhaps). The LINK command must list this library first, as shown here:

```
$ LINK/MAP/DEBUG driver_object,-
_ELN$:FILE/LIBRARY,-
_RTLOBJECT/LIBRARY,-
_RTL/LIBRARY
```

Selection of Default Double-Precision Type

When the VAXELN toolkit is installed on your system, libraries are configured so that the VAX D_floating format is used by run-time routines for double-precision operations. (Note that some mathematical run-time routines generate temporary double-precision values even when your program has not declared any DOUBLE data.)

Two command procedures are placed in ELN$ to allow you to specify the double-precision format for the machine for which you are developing a VAXELN system:

- GFLOATRTL.COM makes G_floating the default representation for the run-time routines. Type @ELN$:GFLOATRTL to make G_floating the default.
- DFLOATRTL.COM restores D_floating as the default representation.

Note also that D_floating is the VAXELN PASCAL and VAX C compilers' default double-precision floating-point representation. To explicitly generate G_floating instructions instead, use the compiler's G_FLOATING command qualifier.

For example, to prepare a program for a MicroVAX computer that has the F_ and G_floating formats, compile with the G_FLOATING qualifier and use GFLOATRTL.COM to specify the run-time representation; you can then omit the emulation software (described in Chapter 13, "System Builder") for floating-point instructions.

General Information on Linking

The organization of run-time support into object and shareable image libraries allows you the flexibility to specify an image's run-time support several ways:

- All program images can share a single copy of those routines that are shareable. Here, their object modules are linked with RTLSHARE or CRTLSHARE and RTL.
- All program images in the final system can use their own local copies of all run-time routines. For this purpose, each program's object modules are linked with RTLOBJECT.OLB or CRTLOBJECT.OLB and RTL.OLB. (RTLOBJECT or CRTLOBJECT should be listed first, because it refers to symbols defined in RTL.)
- A system can be built with both types of images; that is, some program images will keep local copies and some will share the same copies.

In most cases, it is preferable to link with RTLSHARE or CRTLSHARE instead of with RTLOBJECT or CRTLOBJECT. When a program image has references to a shareable image, the System Builder ensures that the necessary shareable images are built into the system.

Because RTLSHARE.OLB and RTL.OLB are included in virtually every linker operation, you will probably find it convenient to define the following logical names:

```
$ DEF LNK$LIBRARY ELN$:RTLSHARE.OLB
$ DEF LNK$LIBRARY_1 ELN$:RTL.OLB
```

LNK$LIBRARY and LNK$LIBRARY_1 are default libraries for the linker, so when these definitions are in effect, you need only specify the object module or modules for your program, for example:

```
$ LINK myobject
```

Qualifiers

Qualifiers are appended either to the LINK command itself or to individual file specifications. Each qualifier is preceded by a slash (/). The file qualifier LIBRARY or INCLUDE must be appended to any file specification for a library.

DEBUG. DEBUG causes the linker to copy the debugger symbol table information, if any, from object modules to the image. It must be specified for any program whose object modules were compiled with debugging information included. (See Chapter 15 for more information about debugging.)

LIBRARY. LIBRARY means that the associated file is either a shareable image library or object library and, if the linker needs to resolve references from the VAXELN object module(s) you supplied, it will be searched for modules containing the necessary definitions. (Note that, in this discussion, the "modules" referred to are either actual object modules from an object library or shareable images from a shareable image library.)

INCLUDE = (module-list). The use of INCLUDE implies that the qualified file is a library even if the LIBRARY qualifier is not used. The module (or image) names in the list are separated by commas, and at least one must be present. Used by itself, INCLUDE means that the listed modules, and only those modules, are included in the linker operation and searched for unresolved symbols, such as routine names; this is a useful optimization when you know exactly which modules are needed by the program. If you specify LIBRARY along with INCLUDE, the library's other modules are searched for any references that remain unresolved after examining the explicitly included ones.

SHAREABLE. The command qualifier SHAREABLE can be used to create a shareable image (for inclusion in your own shareable image library) instead of an executable program image. Shareable image files, like program image files, have the default type EXE, but they are not executable; they are used only to link to object modules. All shareable images, when specified directly in the LINK argument list, must be in shareable image libraries.

To operate on shareable image libraries, you use the LIBRARY command's SHARE qualifier and the shareable image files, instead of object files, as librarian input. As with the images in RTLSHARE.OLB or CRTLSHARE.OLB, the System Builder will ensure that if a program refers to one of your shareable images, that image will be included in the finished system.

Notes

The command qualifier NOSYSSHR is recommended, to prevent the linker from searching the VMS default shareable image library for unresolved references; strictly speaking, there should be no such reference in a correct VAXELN program image.

Shareable images cannot be used as VAXELN Pascal compiler input. This makes no difference in the case of the RTLSHARE library, since its modules are for predeclared routines. It does mean, however, that you cannot create shareable images of your own routines and use them as input for a separate Pascal compilation.

For more information on linker qualifiers and capabilities, including shareable images and image libraries, see the VAX/VMS documentation or on-line HELP for your VAX/VMS system.

Chapter 13
System Development

After you have developed the programs needed by a VAXELN system, you create the system with the System Builder, described here.

This chapter also describes the procedures for including images supplied by DIGITAL (drivers and services), and for booting the finished system image on a *target machine*; that is, a VAX computer with no operating system present.

If you select debugging options for its programs, the system can also be debugged interactively, from the target machine's console terminal or over the Ethernet (see Chapter 15, "Debugging").

EBUILD Command

The EBUILD command invokes the System Builder to combine one or more program images into a bootable system image.

Format

$ EBUILD ***qualifier-list*** **data-file-specification**

Note that optional items in this chapter are shown in *italics*.

Qualifiers

The following qualifiers can be applied to the EBUILD command; each is preceded by a slash (/).

BRIEF, NOBRIEF. These qualifiers, along with MAP, control the contents of the system map. A brief map lists all the images included in the system, all devices and terminals specified, and all the system characteristics. For a description of a full map, see the description of the FULL qualifier, below. (NOBRIEF is the same as FULL.)

EDIT, NOEDIT. EDIT causes the System Builder to enter an interactive screen-editing mode, compatible with VT100- and VT200-series terminals. In EDIT mode, all characteristics and programs of the system can be altered interactively. NOEDIT causes the System Builder to build a system image immediately, from the current contents of the specified data file. EDIT is the default.

FULL, NOFULL. These qualifiers, along with MAP, control the contents of the system map. A full map lists all the images in the system, including their program descriptions, all devices with their device descriptions, all terminal descriptions, and all the system characteristics.

KERNEL = file-specification. KERNEL specifies the name of an alternate kernel image as input. This feature is used only for special applications in which the kernel is being debugged. The default kernel image included in the system is ELN$:KERNEL.EXE. Note that a kernel image is always included implicitly and does not need a program description.

LOG, NOLOG. These qualifiers specify whether or not the System Builder displays the size of the finished system image. The default is LOG.

MAP = *file specification*, NOMAP. These qualifiers enable or inhibit the production of a system map listing. NOMAP is the default. The file specification is optional, and the specified file receives the listing. By default, the listing has the same name as the data file specification and type MAP. The contents of the map are controlled by the BRIEF and FULL qualifiers, which are mutually exclusive. BRIEF is the default.

Note: If you describe the same program more than once in a system, the map shows the program name for duplicate descriptions suffixed with a semicolon and a number; this name is the one used in CREATE_JOB procedure calls or the debugger's CREATE JOB command to distinguish one program's description from another.

SYSTEM = file-specification. SYSTEM specifies a file to which the system image is written. By default, the System Builder creates a system image with the same name as the data file and file type SYS.

File Specification

The data file is used by the System Builder to store information entered or changed in EDIT mode. If the file does not exist and EDIT is in effect, the System Builder creates the specified file. If the file does exist and you edit it in EDIT mode, the System Builder creates a new version of the file incorporating your edits. A new file, or a new version of an existing file, is not made if you are not in EDIT mode, do not change existing information, or leave the editing session with the command QUIT.

If you do not specify a file type, EBUILD uses the default file type DAT.

If you select *Build System* in EDIT mode, or if you use NOEDIT, the System Builder creates a new system image file (default type SYS) from the data file you supplied or from the data file of the editing session.

System Builder Menus (EDIT Mode)

When you use EDIT mode, EBUILD displays a series of menus on the screens of VT100- and VT200-series terminals. With this interface, you can tell the utility what programs, devices, system characteristics, network node characteristics, terminal characteristics, and console characteristics you want.

We suggest that to guide your reading, you invoke EBUILD now, with the name of a nonexistent file, for example:

```
$ EBUILD sample
```

Note: To ensure that VAX/VMS has the correct characteristics for your terminal, type the following command before using EBUILD:

```
$ SET TERMINAL/INQUIRE
```

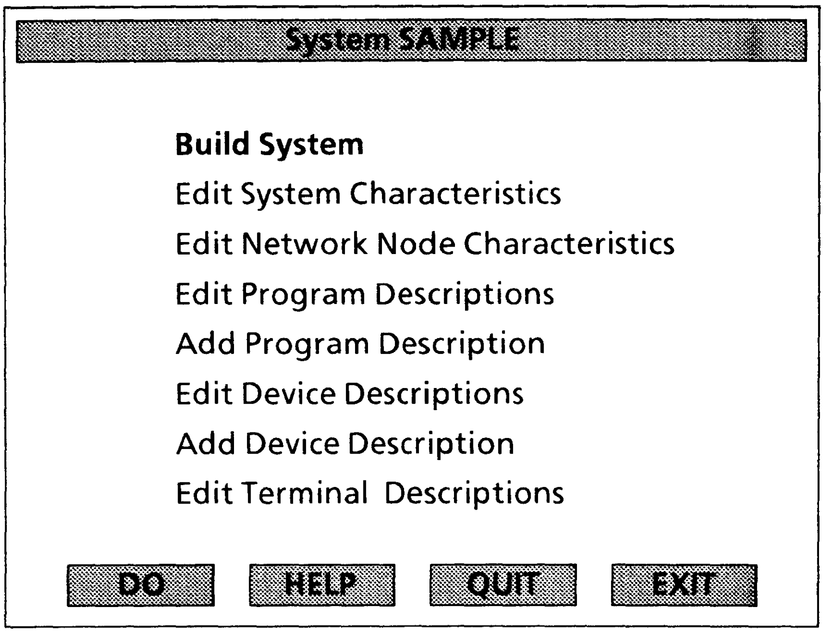
Main Menu

The main menu is displayed when you invoke the EBUILD command unless you use the NOEDIT qualifier. The menu items allow you to build a system from the current data-file information, edit the characteristics of the entire system and the network node, add or edit descriptions of programs, devices, and terminals, and edit console characteristics.

The menu header shows the name of the system (the data or system file name) that you're working on.

The legends DO, HELP, QUIT, and EXIT correspond to the terminal keys PF1, PF2, PF3, and PF4, respectively. (In any menu, these legends define current actions of those keys, if any.)

The *Main* menu looks like this:



The arrow keys (↑, ↓, ←, →) are used on all menus to select menu entries or, in some cases, options within entries. (Some menu entries are filled in just by typing words.) The currently selected entry is highlighted on the screen. (With VT100 terminals, the highlighting is not seen unless the terminal has the Advanced Video Option.)

The diamond symbol appears at the top, bottom, or edge of a menu to show that there is additional text off the screen; you can "scroll" with the arrow keys to show the information. You can also use the CTRL/E and CTRL/H keys to move around in an argument list; CTRL/E moves to the end of the list and CTRL/H (or Back Space) moves to the beginning. CTRL/R refreshes the screen at any time; for example, to remove broadcast notices from the display. CTRL/U deletes text from the cursor back to the beginning of the current line.

DO activates the currently selected entry (for example, incorporating edits you have made with the *Edit Program Descriptions* entry). QUIT aborts the System Builder without altering the input file (it requires confirmation with DO). EXIT ends the System Builder session but incorporates your changes. If you activate *Build System*, the System Builder creates the new system image file (type SYS) and exits. If DO is pressed and an integer menu entry is not in its allowable range, you are given a chance to change it to a valid value.

HELP supplies brief descriptions of the menu entries or the general System Builder interface, depending on your current context.

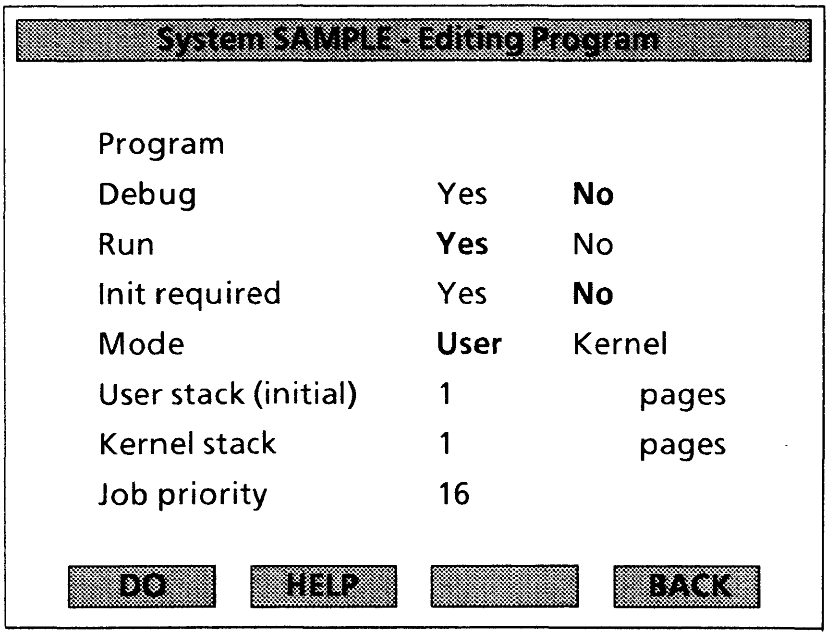
Program Descriptions

If you are running EBUILD, select the entry *Add Program Description* on the main menu and press DO (PF1), to guide your reading.

Each image in the system (except the kernel and the shareable run-time library images) has a program description, which you can add or edit by selecting the main menu entries. (As a shortcut, when there are no program descriptions yet in the system, *Edit Program Descriptions* displays an empty menu; DO will then switch you to *Add Program Description*, so you don't have to return to the main menu first.)

Some program descriptions are added for you automatically by the System Builder. For example, if you select the *File access listener* entry on the *Edit Network Node Characteristics* menu, the File Access Listener's image and program description are added automatically. So are most device drivers, as described later in "Device Descriptions."

The *Program Description* menu looks like this:



Program		
Debug	Yes	**No**
Run	**Yes**	No
Init required	Yes	**No**
Mode	**User**	Kernel
User stack (initial)	1	pages
Kernel stack	1	pages
Job priority	16	

The default settings for program descriptions are highlighted on the screen. The defaults have been chosen as good starting points for most values, and it is advisable to use them before changing them.

The first entry on the menu is the name of the program image you are describing. For example, if the image file is named MYDRIVER.EXE, MYDRIVER is the name you supply. You can describe the same image any number of times if you want the system to associate different characteristics with different jobs running that program. For example, you might want to include more than one description of a program and be able to debug one of them but not the others.

To include several descriptions, you use the same image name in each, and the actual image is loaded only once. The different descriptions, however, apply to the different jobs created to run the images.

To distinguish one description from others, the System Builder's map file shows the program name for duplicate descriptions suffixed with ";*n*" where *n* is an integer and 1 means the first duplicate description. This name, with the suffix, is used as the program name argument in the CREATE_JOB procedure and in the debugger's CREATE JOB command.

Program descriptions, in addition to the image name, consist of the following information. If you press the DO key, the description entered in the system file is what you have specified or what is supplied (and displayed) by default; if you press BACK, you are returned to the previous menu, and your edits are not incorporated.

Debug. If you select *Yes*, any job that runs this program gives control to the debugger instead of executing immediately. (See Chapter 15 for more details about debugging.) The default is *No*.

Run. *Yes* means that a job running the program image is started automatically when the system itself is started. *Yes* is the default.

Init required. If you say that initialization is required, it means that the program is run automatically and will run to completion before any other program starts unless it calls the INITIALIZATION_DONE procedure. If several programs have this property, they are started in order of job priority. (The System Builder assures, however, that debuggers and device drivers are started in the necessary order.) The default is *No*.

Mode. The processor mode in which a program runs is either *User* or *Kernel*. Kernel mode is required for device drivers (programs calling CREATE_DEVICE), programs using the ALLOCATE_MAP, MFPR, MTPR, and FREE_MAP routines. User mode, the default, is recommended unless a program definitely requires kernel mode.

User stack (initial). The user stack is used for user-mode calls to your own procedures and most predeclared procedures. It is extended automatically as needed during the execution of a job. Any stack size you specify is thus the initial size; the default is 1 page. The specified size must be in the range 1–32,767 pages.

Kernel stack. The kernel stack is used by all programs for kernel procedure calls, and by kernel-mode programs for all execution. The kernel stack is not automatically extended. If the program attempts to use too much kernel stack, the process receives an exception. Most kernel mode programs require a larger kernel stack than the default 1 page. The specified size must be in the range 1–32,767 pages.

Job priority. Note that 0 is the highest priority, and 31 is the lowest. For your own programs, you might want to run for awhile with the default priority (16), until you get an idea about the system's overall performance.

Process priority. This is the initial priority of the master process and any subprocesses it creates. Here, 0 is the highest priority, the lowest priority is 15, and the default is 8. In nearly every case, you should use the default value initially.

Job port message limit. This is the maximum number of messages that can reside in the job port (that is, before being removed by the RECEIVE procedure) at one time. The specified number must be in the range 0–16,384; the maximum value is also the default.

Powerfailure exception. If you select *Yes*, the program receives an exception (KER$_POWER_SIGNAL) when the processor restarts after a power failure. Programs should get this exception only if they have established an exception handler for it; such a handler allows you to take general, system-wide action when power has failed, resetting the system time perhaps. Device drivers generally need to handle power recovery in a special way, with interrupt service routines; this use of interrupt service routines is not affected by this program description choice. The default is *No*.

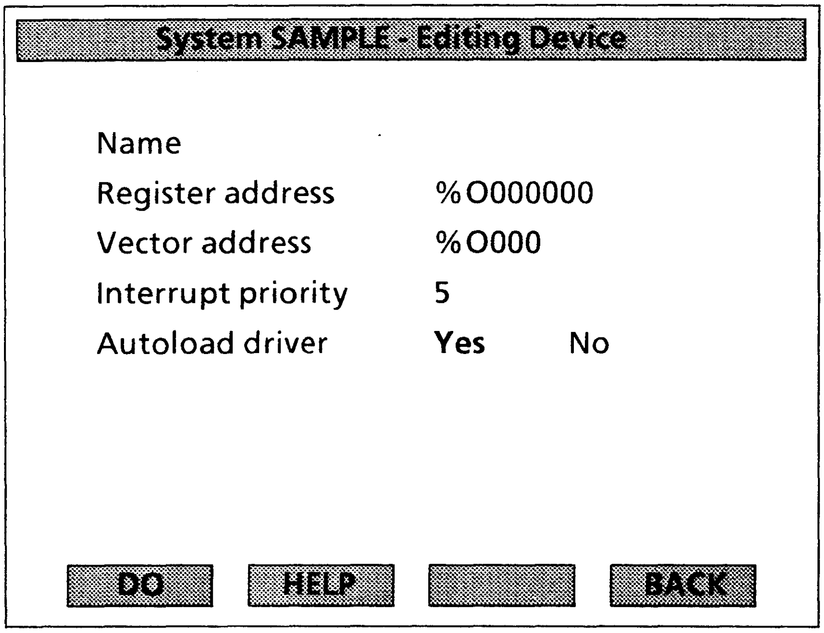
Argument(s). Program arguments are strings and, if they have embedded spaces, must be enclosed in quotation marks (","); note: *not* apostrophes (','). Multiple arguments are separated by commas. Naturally, the requirement for and meaning of program arguments depends on the program. Device driver programs generally take arguments to supply names for the devices they are to control. The most frequent use of program arguments in your programs will probably be to supply file specifications to a Pascal program, to go along with program parameters declared as files.

The argument display scrolls to the left if you enter more arguments than will fit on the screen; you can scroll back and forth with the left- and right-arrow keys (←, →). You can also use the CTRL/E and CTRL/H keys to move around in the argument list; CTRL/E moves to the end of the list and CTRL/H (or Back Space) moves to the beginning. CTRL/R refreshes the screen at any time (for example, to remove broadcast notices from the display), and CTRL/U deletes text from the cursor back to the beginning of the current line.

Device Descriptions

Device descriptions consist of a device name, register address, vector address, and interrupt (bus-request) priority, and in most cases, cause the associated device driver to be built into the system automatically. Each description specifies a single device that is part of the target machine's hardware configuration.

The *Device Description* menu looks like this:


System SAMPLE - Editing Device

Name		
Register address	%0000000	
Vector address	%0000	
Interrupt priority	5	
Autoload driver	**Yes**	No

No device description is necessary for the target machine's console terminal or Ethernet adapter; these descriptions are provided for you when you select the corresponding entries on the *Edit System Characteristics* menu. (See Chapter 14, "Booting and Down-Line Loading," for the default Ethernet adapter addresses assumed by the down-line bootstrap loader.)

The control/status register addresses, interrupt vector addresses, and priorities for bus devices are specified exactly as described in a device's hardware manual. For devices supported by DIGITAL-supplied drivers, see Table 13-1. (The device names suggested in Table 13-1 are the conventional names for the first device controller of the specified type and cause the appropriate driver to be loaded, as explained below.)

In cases where more than one device controller is permitted on the same backplane (such as the DZV11 multiplexer in MicroVAX systems), the addresses shown in Table 13-1 are for the first such controller. For further information about additional controllers, see the hardware manual for the device or, for MicroVAX devices, the *MicroVAX I Owner's Manual.*

Devices with multiple interrupt vectors require only one device description; the other vectors are obtained with arguments of the procedure CREATE_DEVICE.

Name. The name describes a device controller. It is used in programs as an argument to the CREATE_DEVICE procedure and, usually, as a program argument to the corresponding driver program. For example, if a terminal controller is named TTA, its driver program is given the argument 'TTA'. The driver program uses this name as a CREATE_DEVICE argument. For terminals, individual lines are described by terminal descriptions and named with the controller name and a line number (for example, TTA1); see the discussion of terminal descriptions later in this chapter.

Table 13-1. Device Information

Device	Register Address	Vector Address	Priority	Name	Remarks
VAX–11/730 integrated controller (RB02/RB80 disk)	%o775606	%o250	5	DQA	RB02 disk cartridges are the same as RL02 cartridges.
TU58 console tape (VAX–11/730 or VAX–11/750)	—	%o360	4	DDA	Leave the register address blank, since this device is controlled by internal processor registers.
UDA50 adapter (VAX–11/730 or VAX–11/750)	%o772150	%o154	5	DUA	Typical devices are the RA80, RA81, and RA60 disks.
DMF-32 devices (VAX–11/730 or VAX–11/750 line printer, terminals, DR11C parallel I/O)	%o760340	%o320	5	LCA, TXA	The same addresses must be specified for every device on the same DMF-32. LC is the conventional name for the printer controller, and TX is for terminals.
DZV11 (MicroVAX multiplexer)	%o760100	%o300	4	TTA	Modems are not supported by DZVDRIVER. (Note that the DZV11 is shipped with a register address of %o760010, which must be changed to %o760100 for a MicroVax.)
DHV11 (MicroVAX multiplexer)	%o760440	%o300	4	TTA	Modems are supported on all eight lines.
LPV11 (MicroVAX line printer controller)	%o777514	%o200	4	LPA	
RQDX (MicroVAX disk controller)	%o772150	%o154	4	DUA	The RQDX1 controller supports both RX50 diskettes and RD51/RD52 Winchester disks.

Table 13-1. Device Informatic

Device	Register Address	Vector Address	Priority	Name
RC25 (VAX-11/725)	%o772150	%o154	4	DUA
TK50 cartridge tape system	%o774500	%o260	5	MUA
TU81 reel tape system	%o774500	%o260	5	MUA
AXV11-C analog input/ output board	%o170400	%o400	4	AXV
KWV11-C programmable real-time clock	%o170420	%o440	4	KWV
DLVJ1 asynchronous serial interface	%o176500	%o300	4	DLV
DRV11-J high-density parallel interface	%o764160	%o340	4	DRV

Note that the device name must be used consistently in various contexts: here, in CREATE_DEVICE calls, in OPEN calls, and so on; however, the actual choice of a device name is entirely up to you. In the VAXELN documentation, the conventional names for devices are used, such as 'DQA1' for a disk controlled by the VAX–11/730 Integrated Disk Controller. Any name for this or any other device is perfectly all right, as long as it is used consistently. (See also the discussion below about the effect the device name has on automatic device driver loading.)

Register address. This entry gives the physical, 18-bit address of the device's first device control register (18-bit values are used for QBUS devices as well). A driver program can then obtain the address with an output argument of CREATE_DEVICE. Valid values range from %o000000 (the default) to %o777777 and can be specified in decimal, octal (%o), or hexadecimal (%x). For the correct value, see the device's hardware manual or Table 13-1.

Vector address. This is the address of the device's first interrupt vector. A driver program can then obtain the address (and the others, for multiple-vector devices) with an output argument of CREATE_DEVICE. Valid values range from %o000 (the default) to %o776. For UNIBUS and Q22 bus devices, this is the vector that the device asserts on the bus when its interrupt request is acknowledged. It is actually used by the VAX processor as an index into the second page of the System Control Block (SCB). The SCB vector is not actually filled in until CREATE_DEVICE is called. For the correct value, see the device's hardware manual or Table 13-1.

Interrupt priority. This value is the device's bus-request priority, from 4 to 7; the default is 5. These values correspond to the VAX interrupt priority levels 14 (hex) to 17 (hex), respectively. The resulting interrupt priority level can then be obtained with an output argument of CREATE_DEVICE. For the correct value, see the device's hardware manual or Table 13-1.

Autoload driver. If you select *Yes* (the default), the appropriate device driver image is included in the system automatically:

1. The current list of program descriptions is searched for a program that has the *Kernel* and *Init required* characteristics and that has at least one program argument matching the device name. This method may be preferable while you are developing a new device driver, so you can give it the *Debug* characteristic.
2. Otherwise, if one exists, a driver is obtained from ELN$:*cc*DRIVER.EXE, where *cc* is the first two characters in the device name. The name specified by the device description is passed to the driver as a program argument. In this case, a program description is also provided for you: Automatically loaded drivers are given high job priorities (such as four), a kernel stack of four pages, the *Kernel* and *Init required* characteristics, and other appropriate program characteristics. (Use the MAP command qualifier to examine the results.) This is the easiest way to include drivers, including ones you write yourself.

Notes: The selection and loading of terminal drivers other than the Console Driver is controlled by the *Terminal type* entry on the *Terminal Description* menus; it *is*, however, necessary to supply a device description for the terminal controller itself, such as the DMF-32 asynchronous controller. You then use the controller's device name with an appended digit as a

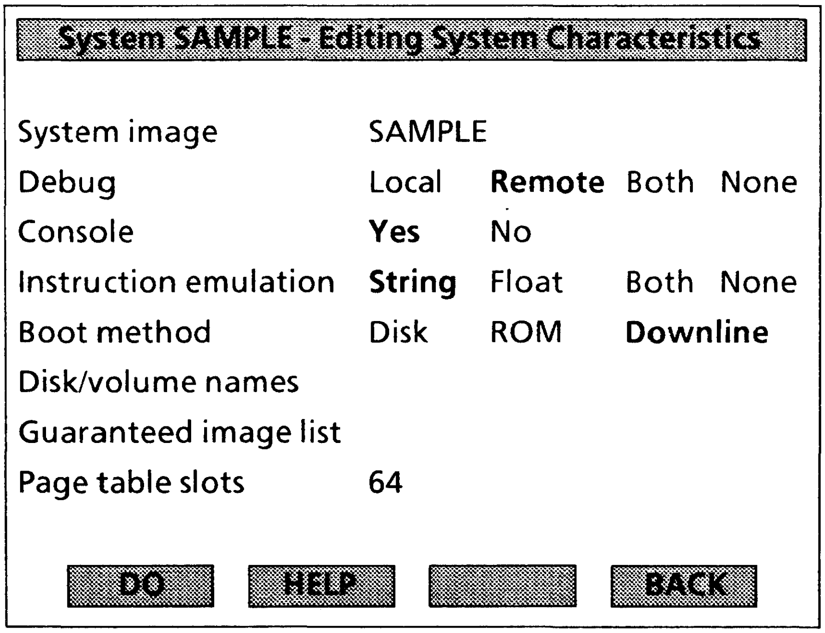
Terminal name on the *Terminal Description* menus, to designate the characteristics of the particular terminal attached to that line.

When autoloading any terminal driver, at least one terminal name must be specified on the *Terminal Description* menu for each terminal controller.

You may want to load a line printer driver with an explicit program description; doing so allows you to specify, as a second program argument, a universal name, so that the printer can be accessed from remote nodes.

System Characteristics

System characteristics are overall properties of the system. The *System Characteristics* menu looks like this:



System SAMPLE - Editing System Characteristics

System image	SAMPLE			
Debug	Local	**Remote**	Both	None
Console	**Yes**	No		
Instruction emulation	**String**	Float	Both	None
Boot method	Disk	ROM	**Downline**	
Disk/volume names				
Guaranteed image list				
Page table slots	64			

System image. This is the name of the (output) system image file. The default name you see on the characteristic menu is either the name of your data file (which will be suffixed with type SYS to form the output file specification) or the name you supplied in the SYSTEM qualifier on the EBUILD command.

Debug. Here (as opposed to program descriptions), the options specify which debuggers are built into the system, if any. *Local* debugging means that the debugger image EDEBUGLCL is included; *Remote* designates the remote debugger (EDEBUGREM), for use with the EDEBUG utility; this is the default. If you select *Both*, both are included. In this case, EDEBUGREM will be the primary means of debugging; EDEBUGLCL will get control only in the event of a system error. (For more information on the debuggers, see Chapter 15).

Note: You may want to include a debugger during the development of a system, even if no program has the Debug option; the debugger will get control in the event of exceptions that are not handled by any program.

Console. Selecting *Yes* (the default) means that a Console Driver and the device description for the console terminal are included automatically, to allow communication with the console terminal on the target machine. The driver and device description are included implicitly if the Debug option is *Local* or *Both*. The name for the system's console terminal (for use in the OPEN procedure, for example) is 'CONSOLE:'. If you select *No* with the debugging option *Remote*, the remote VAX/VMS terminal behaves as the console terminal.

Instruction emulation. This entry selects emulation software for instructions present in the "full" VAX architecture but not included in the MicroVAX architecture. Select *None* unless you are building a system for a MicroVAX I target machine. Selecting *Float* includes emulation software for the extended precision floating-point instructions. Selecting *String* (the default) includes emulation for the other instructions in the subset. Selecting *Both* includes emulation for both groups.

Note that you can choose the double-precision format in VAXELN Pascal and C programs by using the compiler qualifier [NO]G_FLOATING, and you can choose the default double-precision format of the run-time library with the command procedures DFLOATRTL.COM and GFLOATRTL.COM. This way, you can ensure that only the microcoded floating-point instructions are generated and can omit the floating-point emulation software, which is decidedly slower.

Boot method. This selects the method by which the finished system will be booted on the target machine. Systems can be loaded and booted from *Disk*, from read-only memory (*ROM*), or *Downline* (the default). If you say a system with the Network Service will be booted from disk or ROM, but have not specified a node address, a warning message is issued. This characteristic specifies the type of image header used in the system:

Disk	No header
ROM	MicroVAX ROM header
Downline	VAX/VMS image header

Disk/volume names. These supply device specifications and volume names for disks present on the target machine, in the following format:

"device-specification□*volume-name*"

where □ is a space.

For example:

```
"DQA1 TEST1"
```

Programs then can refer to a volume by prefixing the given name with DISK$. For example, in Pascal:

```
OPEN(file1,FILE-NAME : =
'DISK$TEST1:[rt.src]rxdriver.pas')
```

Multiple quoted arguments are separated by commas; the first such specification identifies the default disk volume. The File Service will automatically mount the indicated volumes when the system is bootstrapped. The volume name is optional; if it is omitted, the File Service will attempt to mount whichever disk is present in the indicated drive.

Guaranteed image list. This is a list of shareable images, separated by commas, that are referenced by programs loaded by the dynamic program loader. Shareable images that are referenced by programs in the system are automatically included.

Page table slots. This is the maximum number of page tables that the system can use at one time. Each job requires two process page tables (one for mapping the P0 region and one for the P1 region). Each additional subprocess in the job requires one more, for mapping its P1 region. The default value, 64, thus accommodates a system with 32 simultaneous jobs, if they do no multitasking, and so forth. The minimum number of slots is 2; the maximum is 32,767.

Ports. This is the maximum number of message ports the system can use at one time. The minimum number of ports is 2; the maximum is 32,767. The default is 256.

Pool size. This specifies the approximate number of system objects that can be in simultaneous use. One "block" (the units of the menu) is needed for each system object in use, processes require a total of three, and a few additional blocks are needed for each job. Essentially, you can use one block per system object plus 3 times the number of jobs and processes in simultaneous use. Pool blocks themselves are 128 bytes in size. The number specified must be in the range 16–32,764; the default is 384 blocks.

Virtual size. This is the maximum size, in 512-byte pages, of each P0 and P1 region in the system. The value is used by the kernel to allocate process page tables for each job and process. By default, then, each job can use 0.5 million bytes of virtual memory for its P0 region and an equal amount for each process's P1 region. The size must be in the range 128–32,640 pages; the default is 1,024 pages.

Interrupt stack. This is the maximum number of pages required for the system interrupt stack. It must not be lower than the default, 2 pages. The size must be in the range 2–8,192 pages.

I/O region size. This specifies the maximum number of pages required by all interrupt service communication regions. The value is used by the kernel to allocate system page table entries during the start-up of the system. The value must be in the range 0–32,767; the default is 128 pages.

Dynamic program space. This is the number of memory pages that can be allocated for dynamically loading programs into the running system. Note that the pages are not actually allocated until needed. The value must be in the range 0–32,767; the default is 0 pages.

Time interval. This is the interval, in microseconds, between interval timer interrupts. The specified value is the minimum time that can be used for time-dependent operations. Each interrupt increments the system time and starts time-dependent scheduling in the system. The value must be in the range 1–120,000,000 microseconds (2 minutes). Note that, on some processors (including the MicroVAX), the default of 10,000 microseconds cannot be altered.

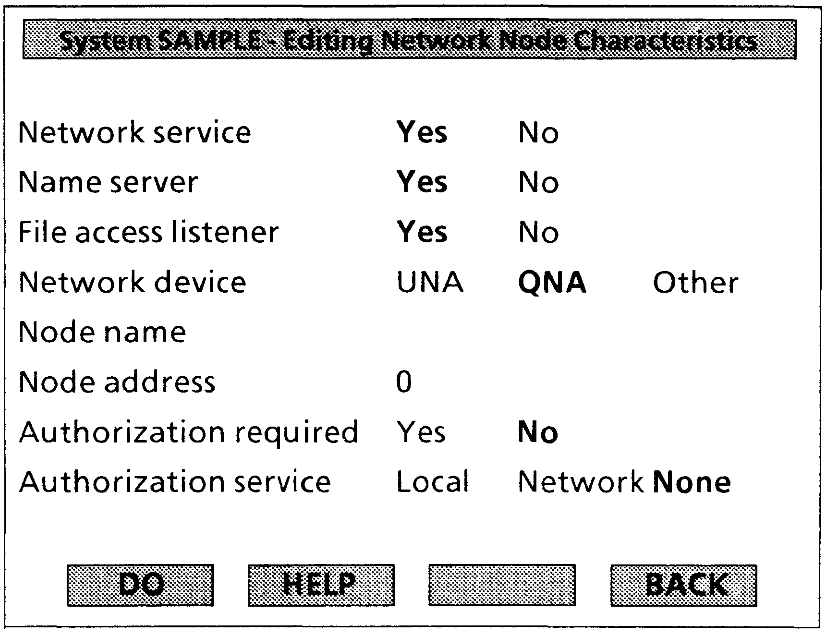
Connect time. This is the time, in seconds, that is allowed to elapse before a circuit connection must be accepted. The value must be in the range 1–3,599 seconds (59 minutes, 59 seconds); the default is 45 seconds.

Memory limit. This specifies the maximum amount of physical memory, in pages, that is available for use by the system. The default limit of 0 means the system should use all the memory available on the target configuration. A limit only needs to be specified in special applications; for example, for a system that contains a multi-ported memory. The minimum is 0 (or no limit); the maximum is 65,535 pages.

Network Node Characteristics

The items on this menu define the network node characteristics for the Network Service and the Authorization Service.

The *Network Node Characteristics* menu looks like this:


System SAMPLE - Editing Network Node Characteristics

Network service	[✓ Yes]	No	
Name server	[✓ Yes]	No	
File access listener	[✓ Yes]	No	
Network device	UNA	[✓ QNA]	Other
Node name			
Node address	0		
Authorization required	Yes	[✓ No]	
Authorization service	Local	Network	[✓ None]

Network service. Selecting *Yes* (the default) includes the Network Service automatically. (For general information on setting up a network, see Chapter 7, "The Network Service.")

Name server. Selecting *Yes* (the default) means that the Network Service running on this machine can "volunteer" to be the name service in network applications. In such applications, at least one system (and usually more) should have this characteristic, to ensure the integrity of universal names in case one or more processors shut down. The *Network service* entry must also be selected.

File access listener. Selecting *Yes* (the default) includes the File Access Listener automatically. Briefly, the File Access Listener allows access to files on this node from a remote node, using a device name or a null volume name. An informational message is issued by EBUILD if you include the File Access Listener without also selecting *Network service.*

Network device. This selects the type of interface that connects a VAXELN machine to the Ethernet, in network applications. *UNA* is the Digital UNIBUS-to-Ethernet adapter (DEUNA). *QNA* specifies the QBUS-to-Ethernet adapter (DEQNA) and is the default. The necessary device driver program (for example, XEDRIVER) and device description are included automatically if you also specify *Network service.*

Node name. This is the node name by which a VAXELN node is identified in a network. It has a maximum of six characters and must be unique in the network. You do not have to specify the node name if the system will be loaded down line from your development system. (For more information about adding node names to a network and down-line loading, see Chapter 14, "Booting and Down-Line Loading.")

Node address. This is the address for a VAXELN node in a network. You do not have to specify it if the system will be loaded down line. The address has one of three forms:

nnn	DECnet node number
aaa.nnn	DECnet area and node number
nn-nn-nn-nn-nn-nn	48-bit Ethernet address

where a and n are digits. The default address is 0. (For more information about node addresses and node numbers, see Chapter 14.)

Authorization required. Selecting *Yes* means that the Network Service will not allow inbound circuit connections unless it can authorize the user via communication with the Authorization Service. When set to *No* (the default), the Network Service will not authorize inbound connections via the Authorization Service.

Authorization service. This specifies whether an Authorization Server should be included in the system and if so, whether the service should serve only the local node (*Local*) or the entire local area network (*Network*). Some nodes can have local services of their own, but there should be only one network Authorization Server. The default is *None*. (For general information on system security, see Chapter 8, "The Authorization Service.")

Authorization file. This specifies the name of the data file that the Authorization Service should use. The data file must exist either on the same node as the Authorization Service, or on a node that the service is authorized to access (for example, one with its own local service). The default file is AUTHORIZE.DAT.

Default UIC. This specifies the default user identification code (UIC) for users not explicitly authorized. The default is [1,1].

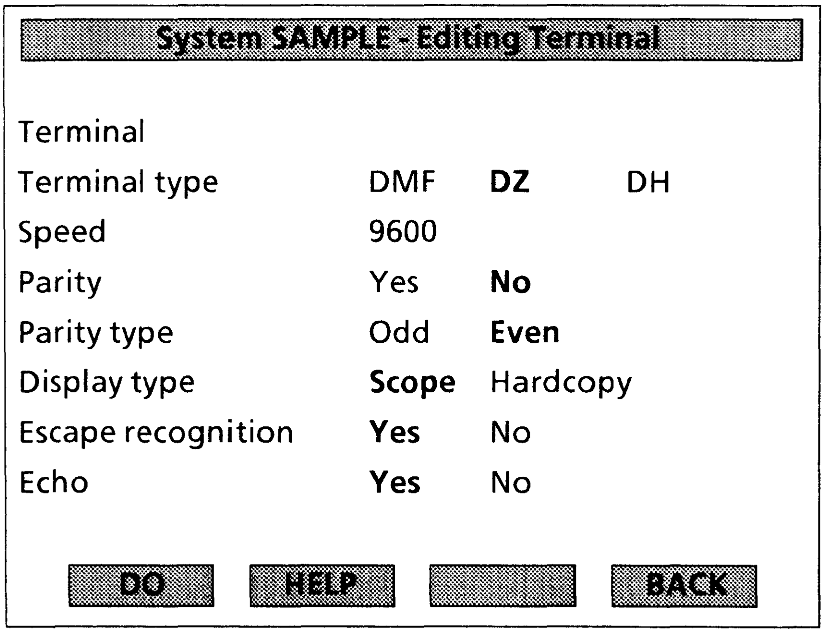
Node triggerable. This specifies whether down-line load triggers are enabled. Selecting *Yes* (the default) means that the system will allow itself to be remotely triggered; this should be the setting during development, so that developers can remotely load the system.

Network segment size. This is the size, in bytes, of the largest segment that will be sent over the network. Note that this maximum applies to any intermediate routing nodes between the source and destination of a message. The segment size includes a 32-byte header prefixed to remote datagrams by the Network Service; consequently, the largest message data buffer that can be sent to a remote node as a datagram has a byte size of the segment size minus 32. The number must be in the range 192–1,470; the default is 576 bytes.

Terminal Descriptions

Each terminal description supplies information about a terminal connected to an asynchronous serial controller line. Be sure to include a device description for the asynchronous controller (for example, the DMF-32 or DZV11). The console terminal is described on a separate menu, *Edit Console Characteristics*.

The *Terminal Description* menu looks like this:



System SAMPLE - Editing Terminal

Terminal			
Terminal type	DMF	**DZ**	DH
Speed	9600		
Parity	Yes	**No**	
Parity type	Odd	**Even**	
Display type	**Scope**	Hardcopy	
Escape recognition	**Yes**	No	
Echo	**Yes**	No	

Terminal. The terminal is designated by the controller device name suffixed with a unit number; for example, TTA0 is the first line on controller TTA.

Terminal type. This specifies the type of controller used for terminals. *DMF* means the asynchronous lines on a DMF-32 controller. *DZ* (the default) designates the DZV11 interface and *DH*, the DHV11 interface, for the MicroVAX. Note that this designation is used only to select and load the terminal driver for the controller type; it is the terminal *name* (for instance, TTA1) that designates a terminal in programs.

Speed. This is the baud rate that applies to the individual line and specifies the speed for both input and output. Possible values are:

50	134	600	2000	4800	19200
75	150	1200	2400	7200	38400
110	300	1800	3600	9600	

The default baud rate is 9600.

Parity. Selecting *Yes* enables parity checking on this line. The default is *No*.

Parity type. The choices *Odd* and *Even* specify the parity type, or "sense," if parity checking is enabled. The default is *Even*.

Display type. *Scope* means that the attached terminal is a cathode ray tube (CRT) terminal, such as a VT100 or VT200. *Hardcopy* means that the attached terminal is a hard-copy terminal; that is, a terminal that prints on paper rather than displaying on a screen. The default is *Scope*. This setting is ignored on a DDCMP specified line.

Escape recognition. *Yes* (the default) means that, on input, the terminal driver program checks the format of escape sequences to see whether they conform to ANSI format. This setting is ignored on a DDCMP specified line.

Echo. *Yes* (the default) means that input characters are echoed on the terminal. This setting is ignored on a DDCMP specified line.

Pass all. *Yes* means that all control characters are passed to the user's program as ordinary input, instead of being interpreted by the driver program. The default is *No*. This setting is ignored on a DDCMP specified line.

Eight-bit. *Yes* means that the attached terminal uses eight-bit ASCII characters. The default is *No*. This setting is ignored on a DDCMP specified line.

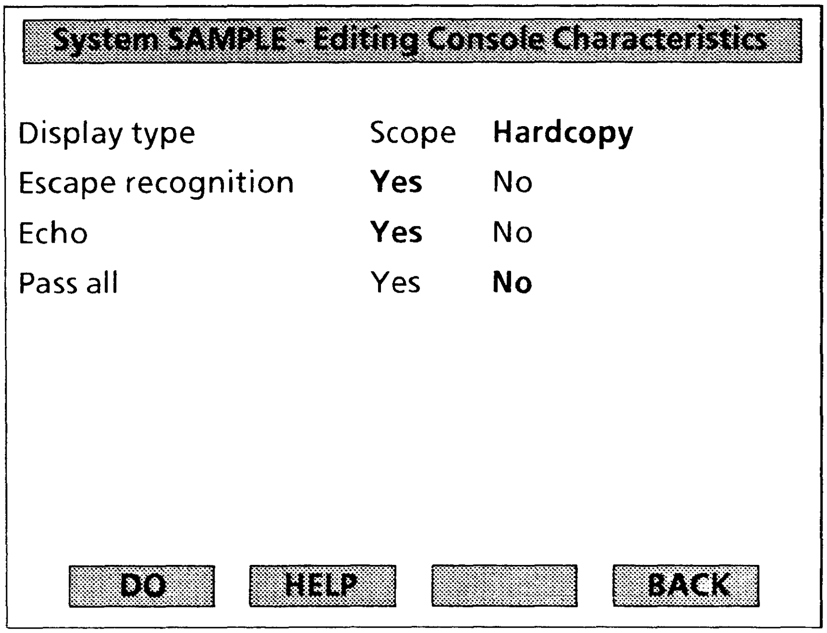
Modem. *Yes* means that a modem is attached to this line. Modems are supported only on the DHV11 and DMF-32 controllers. The default is *No*.

DDCMP. This specifies whether the terminal line should use the DIGITAL Data Communications Message Protocol (DDCMP) for asynchronous DECnet communication with another system. *Yes* means that the line behaves as a point-to-point full-duplex DDCMP link. *No* (the default) means that it is a regular terminal line.

Console Characteristics

The items on this menu have the same meanings as for terminal descriptions, but the settings apply only to the console terminal on the target machine.

The *Console Characteristics* menu looks like this:


System SAMPLE - Editing Console Characteristics

Display type	Scope	**Hardcopy**
Escape recognition	**Yes**	No
Echo	**Yes**	No
Pass all	Yes	**No**

Display type. *Scope* means that the attached terminal is a CRT terminal; *Hardcopy* means that it is a hard-copy terminal. The default is *Hardcopy.*

Escape recognition. *Yes* (the default) means that, on input, the terminal driver program checks the format of escape sequences to see whether they conform to ANSI format.

Echo. *Yes* (the default) means that input characters are echoed on the terminal.

Pass all. *Yes* means that all control characters are passed to the user's program as ordinary input, instead of being interpreted by the driver program. The default is *No.*

Chapter 14
Booting and Down-Line Loading

The system image prepared by the System Builder (file type SYS) is ready to be booted on a target machine.

One such file (ICP.SYS) is created by the VAXELN installation procedure (described in the *VAXELN Installation Manual*) and should be booted once to ensure that the VAXELN installation was successful. This or any other system image can be written onto a Files–11 disk or other medium, carried to the target machine, and booted, following the instructions in the first part of this chapter.

If your host and target machines are connected by the Ethernet, you can use the information in the second part of this chapter for down-line loading and booting.

Booting Systems from Disks

This section shows the procedure for booting ICP.SYS, which adapts to booting any system of your own. It involves copying the system image to a Files–11 volume with the copying procedure that follows, transferring the finished volume to the target machine, and booting from it.

Note: System images using the following procedure must be built with the EBUILD system characteristic *Boot method* set to *Disk*. (See Chapter 13, "System Development," for more information on the System Builder and the EBUILD command.)

The COPYSYS Command Procedure

The copying procedure is the command procedure COPYSYS.COM, placed by the VAXELN installation procedure in ELN$. COPYSYS initializes, loads, and makes bootable a Files–11 disk or TU58 cartridge. Use it as follows to prepare a bootable copy of ICP.SYS on a TU58 cartridge in the console TU58 drive of your VAX–11/750 host development system:

```
$ @ELN$:copysys
System image file: ELN$:icp
Output disk: csa1
Initialize the disk? (Y/N) [N]: y
$
```

If you receive the error message "No such device" after entering the information specified above, the console device is not connected. In this case, have your system manager connect it with the command:

```
$ MCR SYSGEN CONNECT CONSOLE
```

You have now created a TU58 cartridge containing a VAXELN system image (ICP.SYS). You must answer "Y" to the initialization question the first time you use the cartridge for this purpose; if you reuse it for another VAXELN system, you can say "N" (the default is "N").

You can also enter the entire command on one line:

```
$ @ELN$:copysys ELN$:icp csa1
```

Here, the default "no initialization" is chosen.

The TU58 cartridge containing ICP.SYS can be transferred to an 11/750 target machine and booted with the following console boot command:

```
>>>B DD0
```

The system responds by displaying the following:

```
%%


            VAXELN  V2.00
                      .
                      .
                      .
SUCCESSFUL COMPLETION OF ICP
```

For an 11/730, the console boot command takes the name of the device to boot; for example:

```
>>>B DQ1
```

Here, presumably, DQ1 contains an RL02 cartridge that you have prepared with COPYSYS.COM.

For a TU58 cartridge in the external 11/730 drive, the console command is:

```
>>>B DD0
```

The command for booting from an RX50 diskette (DUA1) in the first floppy disk drive on a MicroVAX is:

```
>>>B DUA1
```

Note that the execution of ICP.SYS takes approximately 5 minutes to complete.

Down-Line Loading

This section gives the procedure and preparatory steps for using the Ethernet, instead of portable disks or other media, to load systems onto target machines.

During the debugging cycle, you can load machines with the network, as mentioned here and in Chapter 15, "Debugging."

Down-line loading of a VAXELN system uses a down-line load bootstrap loader, residing on the target machine, and the DECnet network facilities on the host development system. These two software components use the network communication hardware to copy a VAXELN system image file from the host development system to the main memory of the target machine. Once the VAXELN system is in the target memory, it gets control of the processor and begins execution.

The VAXELN system need not contain the Network Service to be loaded down-line. The Network Service must, however, be included to allow network communication between the VAXELN system and other systems on the same network. (For more information, see Chapter 7, "The Network Service").

Preliminary Steps

There are a number of preliminary steps necessary to set up your host and target machines before down-line loading:

1. Install communication hardware on the host and target machines.
2. Install and configure DECnet–VAX software on the host system.
3. Test communication between the host and target machines.
4. Add the target machine's description to the host system's network node data base.
5. Configure or install the down-line load bootstrap loader on the target machine.

Before continuing with the set-up procedures, we recommend that you become familiar with the Network Control Program (NCP). This utility is the principal tool used to control the network software and hardware and is described fully in the *DECnet–VAX System Manager's Guide.*

The following sections describe in more detail some of these requirements.

Installing Communication Hardware on the Target Machine

The communication hardware should be installed at the default I/O bus address on the target processor. Table 14-1 lists the address assumed by the down-line load bootstrap loader for each particular hardware device.

Table 14-1. Datalink Device Default Addresses

Device	Address (Octal)
DEUNA	774510
DEQNA	774440

Configuring a Host for Down-Line Loading

The following commands must be issued to configure your VAX/VMS host for down-line loading, to enable the host's recognition of boot-request messages from the target system:

```
$ RUN SYS$SYSTEM:NCP
NCP> DEFINE LINE UNA-0 SERVICE ENABLED
NCP> DEFINE CIRCUIT UNA-0 SERVICE ENABLED
NCP> SET LINE UNA-0 STATE OFF
NCP> SET LINE UNA-0 ALL
NCP> SET CIRCUIT UNA-0 STATE OFF
NCP> SET CIRCUIT UNA-0 ALL
```

Adding the Target Machine to the Host Node Data Base

The target VAXELN machine needs to be described in the host system's network node data base. To enter the machine in the data base, use the NCP utility to store the target machine's node address, node name, Ethernet hardware address, and host load device name. This information is typically stored in the permanent data base using the DEFINE command. For example (for a node named "FRED"):

```
$ RUN SYS$SYSTEM:NCP
NCP> DEFINE NODE FRED ADDRESS 5 SERVICE -
_CIRCUIT UNA-0
NCP> DEFINE NODE FRED HARDWARE -
_ADDRESS AA-00-03-00-00-E1
```

The node address and name may have already been specified when your network was installed, but always be sure each node in your network has a unique address and name. The service circuit is the name of the host system's hardware device controller, connecting the host system to the target machine.

The hardware address is required for down-line loading via the Ethernet and is the Ethernet address contained in read-only memory on the target machine's Ethernet hardware controller. This address is normally printed on the controller board, but if it is not, contact your field service representative, who can provide the address by running the controller's diagnostic package.

Once the target machine has been added to the host system's permanent data base, the information should be copied to the current "volatile" data base using the SET command. For example:

```
NCP> SET NODE FRED ALL
```

After the DEFINE and SET ALL commands have been used, the target machine's description will remain permanently in both data bases, including across rebootstraps of the host system.

Note that the DEFINE command requires a system user identification code (UIC) or SYSPRV privilege, and the SET command requires OPER privilege.

Note: The commands described above are valid for a VMS system. The commands to configure a MicroVMS system are the same, except that the service line and the service circuit are QNA-0 instead of UNA-0.

Configuring the Bootstrap Loader

The down-line load bootstrap loader must be either configured or installed on the target machine. VAX–11/730 and VAX–11/750-family processors use the console storage medium (TU58) to store the bootstrap loaders. On the MicroVAX processors, the down-line loader is contained in the boot ROM.

To install the down-line load bootstrap loader on a TU58 console tape, use the VAXELN NEWBOOT command procedure. This procedure copies the bootstrap image file and, for VAX–11/730s, a bootstrap command procedure, to the console medium. This command procedure prompts for the bootstrap load device (XE=DEUNA), the device containing the console medium on which the loader is to be installed, and the processor type of the target machine. For example:

```
$ SET DEFAULT ELN$
$ @NEWBOOT
Bootstrap device [XE]:
Console media device [CSA1]:
Processor type [730]:
Set default bootstrap? (Y/N) [Y]:
```

The command procedure copies the loader file(s) to the console medium, and the loader installation is complete.

Note that writing to the console storage device requires that the storage device's driver be loaded, an operation that requires CMKRNL privilege. We recommend that you use the NEWBOOT procedure from the fully privileged system manager account.

Since the MicroVAX down-line loader is contained in its boot ROM, there is, strictly speaking, no configuration necessary. However, it may be useful to set the MicroVAX I CPU's configuration DIP switches to skip disk booting, enabling unattended down-line loading of the target machine. (See the system configuration section of the *MicroVAX I Owner's Manual* for details.)

Down-Line Loading Procedure

To load a target machine down-line, the VAXELN system image file must be available to the network software on the host development system, and the down-line load bootstrap loader must be running on the target machine.

When you build the system with the System Builder, be sure to specify *Downline* as the *Boot method* entry of the *Edit System Characteristics* menu.

The VAXELN system image file is made known to the network software by storing its file name in the host system's network node data base using NCP. For example:

```
NCP> SET NODE FRED LOAD FILE -
_DISK$WORK:[ROBOT]TEST.SYS
```

The same operation can be performed by EDEBUG as follows:

```
$ EDEBUG/LOAD = DISK$WORK:[ROBOT]TEST.SYS -
_FRED
```

The down-line load bootstrap loader can be started using the console boot command ("B") on the target machine. For example, to start the DEUNA loader on an 11/730 enter:

```
>>> B XE0
```

For an 11/750 enter:

```
>>> B CSA0
```

For a MicroVAX enter:

```
>>> B XQA0
```

When the loader starts, it sends a load request message to the host system. In response to the load request, the network software on the host system creates a Maintenance Operation Monitor (MOM) process which reads the specified VAXELN system image file and sends it to the target bootstrap loader.

When a machine is loaded down line (in contrast to being bootstrapped from a disk or read-only memory) it is not necessary to set the node name or node address with the System Builder; as part of the load procedure, the target machine receives its proper node name and address. This also means that if you have a system that needs to be run on multiple processors in a network, the same system image can be used for each machine.

Reloading a Machine that has the Network Service

Once a VAXELN system is initialized and is running the Network Service, it is usually not necessary to enter a new boot command on the target machine's console. Instead, the remote bootstrap "trigger" function can be used.

To use this feature, you must set the default bootstrap loader to the down-line load bootstrap loader by setting the default bootstrap selection switches to the correct read-only loader. On the 11/730, this setting is performed by the NEWBOOT command procedure. On an 11/750, set the Default Boot Device switch to "A"; on the MicroVAX, set the CPU configuration DIP switch number 1 to "on".

To trigger a target machine, use the NCP TRIGGER command. For example:

```
NCP> TRIGGER NODE FRED
```

The trigger function sends a "boot-request" message to the target machine, which causes the VAXELN datalink device driver to halt execution of VAXELN and begin execution of the default bootstrap, the down-line load bootstrap loader.

Note: If desired, the DEUNA controller on an 11/750 target machine can be configured to process the boot-request message and cause the machine to halt by causing a power-failure sequence.

Therefore, to assure that the 11/750 restarts, you must put the Auto Restart switch in the Boot position. Note that this implies that a machine that requires unattended triggering cannot also restart using memory with battery backup (that is, it will always rebootstrap when the power is restored).

If you encounter problems loading your target machine, the network event-logging facility on the host system can often be used to locate the problem. To enable event logging on your host system, use the NCP SET LOGGING commands.

For example, to enable network event logging to your host's console terminal:

```
NCP> SET LOGGING MONITOR KNOWN EVENTS
NCP> SET LOGGING MONITOR STATE ON
```

The resulting messages on the console display the maintenance messages and network state changes observed by the MOM network process. Any problems opening the VAXELN system image file or communicating with the target machine are displayed.

Down-Line Loading during Debugging

During the VAXELN programming and development cycle, the target machine is likely to be loaded down-line and remotely debugged many times. To facilitate this operation, the VAXELN debugger can load machines down-line.

The VAXELN debugger LOAD qualifier stores the specified VAXELN system image file name in the network node data base and triggers the target machine's down-line load bootstrap loader. For example, the system TEST.SYS is loaded as follows during an EDEBUG session:

```
$ EDEBUG/LOAD = TEST FRED
```

See Chapter 15 for a complete description of the debugger.

Reloading Production Machines Down-Line

Once a VAXELN application has been debugged and is installed in production use, you can continue to use the down-line load facilities to load the target machines. The host's node data base needs to contain a description of each VAXELN machine and system in the network. The description should contain all the information described in the previous sections, including the file name of the production VAXELN system image file.

The default bootstrap loader on the target machines should be set to the down-line load bootstrap loader, as described previously. Whenever a target machine is rebootstrapped (for example, after a power failure or a hardware or software crash), it will be reloaded by the host system.

Down-Line Loading from Multiple Hosts

When down-line loading target systems from multiple host systems, it is important that only one host system has down-line load parameters set for the target system. If two or more hosts are capable of responding to a target system's request for down-line loading, the first to respond actually performs the load, independent of which host initiated the load via an NCP TRIGGER command or an EDEBUG/LOAD command.

The load parameters are LOAD FILE, SERVICE CIRCUIT, and HARDWARE ADDRESS. If the last two parameters are set, the host will attempt to load the target, even though it may not have the LOAD FILE name needed to complete the load.

For example, if SERVICE CIRCUIT and HARDWARE ADDRESS are set, but LOAD FILE is not set (the typical case), the VMS MOM program will still attempt to load the target system ("load volunteer"). This blocks other hosts from loading, but later in the process it will discover that it does not have the LOAD FILE name.

Unfortunately, this configuration error is difficult to diagnose. If you encounter "%SYSTEM-F-TIMEOUT, Device timeout" errors logged by DECnet-VAX event logging, this may be the problem. If so, check all the node databases on the hosts and ensure that only one database has the load parameters specified for the target.

For example, if you had been loading target system "ABC" from host "XYZ" and now decide to load it from host "XXX", execute the following NCP commands.

On node "XYZ":

```
NCP> CLEAR NODE ABC SERVICE CIRCUIT -
_ HARDWARE ADDRESS
```

On node "XXX":

```
NCP> SET NODE ABC SERVICE CIRCUIT UNA-0 -
_ HARDWARE ADDRESS AA-00-03-01-2B-0D
```

Chapter 15
Debugging

Introduction

VAXELN provides two methods of debugging a target system. The method is selected from the *Edit System Characteristics* menu of the System Builder.

The first method is *remote debugging*. With this method, a remote debugger nucleus is placed in your system by the VAXELN System Builder. When the host development system and the target machine running your system are connected by the Ethernet, you can use the VAX/VMS command EDEBUG to access the target system and debug processes which are running on it.

Remote debugging with EDEBUG also provides these features:

- The ability to access one or more VAXELN target system nodes at the same time for debugging purposes.
- The ability to make use of the debug symbol table information provided by VAX/VMS compilers so that variable names, labels, and source-line information can be used during your debug session.
- The ability to allow your VAX/VMS terminal to behave as the console device on the target system.

The second method is *local debugging*. With this method, the entire VAXELN debugger utility is built into your system image. This method provides a completely self-contained debugger; therefore, a network connection to a VAX/VMS host is not required. Debugging commands are entered at the system's hardware console terminal.

In this mode, the special remote debugging features listed above are not available. However, the local debugger does provide the special ability to debug the VAXELN kernel in addition to processes on the running system.

This chapter explains how to select a debugger mode with the System Builder, how to make a program "debuggable," how to make the VAXELN kernel debuggable, the format of the EDEBUG command, general concepts for using the debuggers, and debugger syntax rules. The remainder of the chapter is a summary of the debugger commands used in debugging VAXELN systems.

Selecting a Debugger Mode with the System Builder

The VAXELN System Builder provides several alternatives for specifying how you want to debug your target system. The following options are provided on the *Edit System Characteristics* menu:

- **None.** The target system has no debugging capability. You would use this option when your system is fully debugged.
- **Local.** Local debugging is allowed. The debugger is completely contained in the system image.
- **Remote.** Remote debugging is allowed. The EDEBUG utility is used to remotely access the target system over the Ethernet.

- **Both.** Both remote debugging and local kernel-level debugging are allowed. You would use this option when you want to use the features of the EDEBUG command, but still debug the VAXELN kernel from the system hardware console.

From the same *Edit System Characteristics* menu, you select whether or not you want a console terminal capability in the system. If you omit the console device from the system, VAXELN's remote debugger will automatically make your VMS terminal the system's console device when you establish a connection to the node with the EDEBUG command.

Making a Program Debuggable

On the *Edit Program Descriptions* and *Add Program Description* menus of the System Builder, for each program in your system, you can specify that the debugger is to get control when a job that runs the program is started. Each process that is created inside the job is also stopped before it begins execution.

If a debugger is included in the system, it will also gain control of a process if an unhandled condition occurs or if you explicitly ask for control by halting the process with the HALT command. (Chapter 11, "Exception Handling," contains more information on VAXELN's exception handling mechanism.)

Making the Kernel Debuggable

In some rare instances you may need to set breakpoints and examine locations within the VAXELN kernel image. The kernel can only be debugged by the local debugger from the VAX hardware console terminal.

There are three ways to gain the attention of the kernel debugger:

- Type the following:

  ```
  SET SESSION /KERNEL
  ```

  This instructs the local debugger to attach the session associated with the kernel.

- After placing the VAX in hardware console mode (see the appropriate hardware manual for the correct procedure), type the following on the VAX hardware console:

  ```
  >>>D/I 14 5
  >>>C
  ```

  This activates the kernel debugger session if the system is not executing above interrupt priority level 5 (hex). If the system is executing at or above that interrupt priority level, there is no way to gain control at the console.

- Type the following on the VAX hardware console to boot the system:

  ```
  >>>B/4 <device>
  ```

  This activates the kernel debugger during the system initialization sequence.

You leave the kernel debugging session with the GO command.

The EDEBUG Command

The VAX/VMS command EDEBUG debugs the VAXELN system on the given node, where the VAX/VMS system (the "host") is connected to the target machine by the Ethernet. It can also load new systems from the host to the target machine, can start them with or without the debugger in control, and can reconnect to nodes that were loaded previously. (Chapter 14, "Booting and Down-Line Loading," contains information on configuring and managing the Ethernet connection.)

The EDEBUG command DEBUG allows the same operations from EDEBUG itself; this allows you to debug systems on several nodes during the same interaction with EDEBUG on your development system. (See the description of the DEBUG command, later in this chapter.)

Formats

The EDEBUG command has the following formats:

$ EDEBUG nodename

This version of the command is used when the node is already running a system. For example, the machine may have been bootstrapped with a new system, or you may be reconnecting to it.

$ EDEBUG/LOAD = system nodename

Here, system is the file specification for a system image (type SYS file). It must include the remote debugger. The system is loaded across the Ethernet (replacing any system already there) and started.

$ EDEBUG/NODEBUG/LOAD = system nodename

Here, the system is loaded across the Ethernet and started. After loading, the debugger exits.

After you have exited the EDEBUG command, the target system on that node is left in a suspended state. All processes that are not controlled by the debugger continue to run normally. Processes that are awaiting debugger commands, however, remain suspended until you reconnect to the node and return control to the debugger.

EDEBUG uses the logical names DBG$INPUT and DBG$OUTPUT for I/O, which are usually assigned to SYS$INPUT and SYS$OUTPUT, respectively. This allows EDEBUG commands to be entered with VAX/VMS command procedures.

General Concepts for Using the VAXELN Debuggers

The VAXELN debugger facility allows you to perform the common debugging operations of examining or depositing memory, evaluating expressions, setting breakpoints, and controlling the execution of your program and the system. The debuggers also provide a means of performing system-wide operations, such as displaying all the jobs currently running on the system.

VAXELN applications can be made up of multiple jobs and processes executing on several nodes in a network. Therefore, when you are debugging, you may need to control more than one process and more than one target system at the same time. The VAXELN debuggers have special features to handle this requirement.

For each process that you wish to debug, the debugger establishes a *command session*. Commands that you enter are directed to the process associated with the command session. You can change the command session with the command SET SESSION.

A command session is in one of the following states:

Running. The session's process is not waiting for a debug command.

Awaiting Commands. The session is suspended, waiting for debug commands.

If anything happens to change the state of a Running session (for example, a breakpoint is encountered), you are immediately notified, even if that session is not the current command session. All sessions waiting for debug commands are "frozen" while you work with the command session.

Process Identifiers

Sessions uniquely identify a particular process on a particular VAXELN node. The debugger assigns identifiers to each process in a running system. These *process identifiers* also identify the debugger's session for that process. This means that you can use a process identifier to specify a process that does not have an active debug command session.

There are three parts to a process identifier:

- **Job.** The process' job is identified either by name or by a unique identification number (ID) assigned by the debugger. The name is usually the name of the job's program. The ID is necessary if you have more than one instance of a particular program running on the system. In that case, you use the job ID to identify the instance.
- **Process.** The process in the identified job is specified by an identification number.
- **Node.** The node which the process is running.

These three parts are specified as:

job,*process node*

If *process* is omitted, the job's master process is assumed. If *node* is omitted, the current session's node is assumed. For example:

```
4,3
console,3
console,3 node10
```

In some instances the job may contain non-alphanumeric characters, which are not allowed by the syntax of the debugger. To avoid this problem, enclose the job name in single quotes to isolate it. For example:

```
'myjob;2',3
```

The Command Session

The command session to which you are entering commands is shown to you in the command prompt string. For example,

```
Edebug 4,5>
```

means that commands which you type are directed to process 5 in job 4. If you are debugging more than one node, the node name is also included in the prompt.

The command session is set when the debugger begins to debug the first process that requests attention. After that, the command session remains constant until you explicitly change it with a SET SESSION command or until the command session's process exits. When the command session exits, the session is set to its job's master process or, if the existing process *is* the master process, the debugger "picks" a session that is awaiting command input.

The debugger will not prompt for commands while the command session's process is in a Running state.

The "Control-C" Session

A special session is provided for use when there are no debugging sessions or when the active session is running. This session allows you to enter commands that are not directed at any particular process; for instance, SHOW SYSTEM, which displays the current jobs in the system.

This session is activated by typing:

```
CTRL/C
EDEBUG CONTROL-C >
```

The special prompt string is to remind you that the special session is active. You can enter commands until you enter a null command line. At that point, you return to the state that was active when CTRL/C was typed.

The "Control-C" session has some restrictions on which commands it can execute. For example, you cannot EXAMINE or DEPOSIT memory because there is no process context associated with it. You can, however, EVALUATE expressions, perform system-wide commands, and enter HALT commands.

Note that entering CTRL/C at your terminal will abort the command currently executing and return to a session prompt.

Symbolic Debugging

When you are remotely debugging a system using the EDEBUG command, you can refer to program locations, variables, or constants by name; you can also view source lines. The information used by EDEBUG to allow this is provided by the VAX compilers and linker as part of the .OBJ and .EXE files. Typically, you request this by using a DEBUG qualifier at compile and link time (see Chapter 12, "Program Development," for more information).

The debugger symbol table contains the following information for each module that contributed to the program image:

- Information that relates addresses to the line numbers in a module

- Information that relates addresses to the source line that produced the code at those addresses
- Symbolic information (names and types) about the module's variables

The symbol table contains the information about variables only if you requested it when you compiled and linked the module.

Note: The complete debugger symbol tables are quite large; that is, they make the program image much larger than would otherwise be the case. Generally speaking, you will want to recompile programs after they are debugged, without requesting a symbol table. The source-line and traceback information is comparatively small and can be left in the image; both kinds are included in compilations by default. In no case is the symbol table loaded into the system on the target machine.

The symbol table is associated with all the processes in the job running the program. The association is either implicit (when a job starts under debugger control) or explicit (by use of the debugger's SET PROGRAM command). For more information about the relation between the symbol table and references in the debugger, see "Identifiers," later in this chapter.

Breakpoints

Many debug operations cannot be done unless the program is stopped. At the beginning of any session, the associated program is stopped and does not execute until you type the GO command. The STEP command allows you to execute the program one step at a time.

The notion of a *breakpoint* is that of a "stop sign" that you can plant at a particular location in the program; subsequently, the program will stop whenever it reaches the breakpoint. Breakpoints are set with the SET BREAK command, described later in this chapter. For example, if you set a breakpoint at the beginning of a session and then type GO, the program executes until it reaches the breakpoint, where it stops; you are then able to examine variables and perform other operations in the current context. You can continue execution from the breakpoint with the GO command.

The SET BREAK command also allows you to specify a command that is executed when the program stops at the breakpoint. (See the SET BREAK command description, later in this chapter.)

Debugger Syntax Rules

The VAXELN debugger command language is designed to be simple to use for programmers who are familiar with several programming languages; the command language is "generic" to most programming languages. This contrasts with the VAX/VMS debugger, DEBUG-32, which provides a fully functional command language that is dynamically tailored to the language in use at the point where the program is stopped.

The VAXELN debugger command syntax and semantics are modeled after the DEBUG-32 command language and, wherever possible, a command does exactly the same thing in both debuggers.

All commands are expressed in the same general format, as follows:

VERB *qualifiers parameters*

In this chapter, optional items are shown in *italics*. All keywords used as verbs or qualifiers can be abbreviated to their shortest unique form.

An example of the simplest command is

```
Edebug 4,5> EXIT
```

which terminates the debugger. This command has no qualifiers or parameters.

An example of a more elaborate command is

```
Edebug 4,5> EXAMINE /HEX my-variable
```

which examines the variable my-variable and displays the value in hexadecimal form.

If the command does not conform to the general format, or if the verb is not recognized as a known verb name, the debugger treats the command as an expression. For example, the command

```
Edebug 4,5> 123 + 456
```

is perfectly correct and displays the value 579. This feature provides a way to examine the value of a variable by simply typing its name. For example, the command

```
Edebug 4,5> x
```

examines the variable x.

Command Files

The command

```
@file-specification
```

causes EDEBUG to fetch commands from the specified file. The file inclusion can be nested to a depth of 8.

Expressions

Many of the parameters used in debug commands are arithmetic expressions which provide values for the individual commands. Expressions are algebraic sequences consisting of constants, parentheses, operators, and variable references.

The expression syntax is a useful combination or hybrid and is not identical to any language in particular. The semantic rules for interpretation are straightforward, following the normal rules for algebraic expressions.

The valid operators are:

arithmetic:	+
	–
	*
	/
	DIV
	MOD
	@

where + and – can be either monadic (prefix) or dyadic (infix) operators.

Boolean:	=
	<>
	<
	<=
	>
	>=
	AND
	OR
	NOT
address-related:	ADDRESS(variable)
	@

The monadic @ operator is used to take the value of a location represented by an address expression. In addition, the ADDRESS function can be used to take the address of an addressable variable. Note that some variables, such as those assigned to registers, are not addressable.

The dyadic @ operator specifies an arithmetic left-shift if the count is positive, or a right-shift if the count is negative. For example,

```
ref @ 2
```

shifts the contents of ref two bits to the left.

The operators = and := are used interchangeably to express assignments.

String Expressions

The + operator can be used to concatenate string values to form longer string values. For example:

```
'header' + str1
```

Address Expressions

Address expressions are distinct in that they produce an address value for a location instead of the value at the location. To get an idea of the difference, consider this Pascal assignment statement, in which A, B, and C are variables:

```
A := B + C;
```

Here, B, C, and B + C produce values, for assignment to A. A, in effect, is an address expression that designates the address to receive the value. If A were on the right-hand side of the assignment operator, it too would represent a value, not an address.

In the debuggers, you need to be able to express the left-hand side as an expression, meaning that it represents a target address for some operation. For example,

```
EXAMINE 1234
```

means "examine the integer at address 1234."

```
EXAMINE myvar
```

means "examine the variable myvar."

```
EXAMINE myvar + 2
```

means "examine the integer at address-of-myvar-plus-2."

Address expressions consist of variable references, integer constants, and the dyadic operators −, +, *, DIV, /, and @. Parentheses are allowed.

The monadic operators in address expressions are + and −, with the usual arithmetic meanings, and @, which means "contents of." For example,

```
EXAMINE PC
```

displays the contents of the program counter (an address), but

```
EXAMINE/INSTRUCTION @PC
```

displays the instruction at the current PC address. Since the PC always contains addresses, not instructions, the command

```
EXAMINE/INSTRUCTION PC
```

is an error. Similarly,

```
SET BREAK PC
```

is an error. The correct command is:

```
SET BREAK @PC
```

The "address versus contents" distinction also applies to symbolic references to pointer variables. For example,

```
EXAMINE ptr
```

where ptr is a pointer variable, means "examine the contents (an address) of ptr." In contrast,

```
EXAMINE @ptr
```

means "examine the memory pointed to by ptr."

Note: In addition to @, the forms ptr^, *ptr, and ptr→ are equivalent.

Symbols defined using the DEFINE command are treated specially when they appear in address expressions. These symbol references yield values much like a constant does, which is logical since these internal variables are similar to registers (which always express a value).

Identifiers

During your debug session, you may want to refer to items by their text name or identifier. There are three kinds of identifiers:

- Identifiers defined using the DEFINE command
- Special predefined identifiers
- Program locations and variable names available to EDEBUG which are defined by a program's debug symbol table entries

In general, identifiers are ASCII names consisting of up to 31 alphabetic or numeric characters and the special characters dollar sign ($) and underline (_). Identifiers do not begin with numbers, and no distinction is made between upper and lower case.

Identifiers Defined Using the DEFINE Command

Both the local and remote debuggers allow you to define identifiers for use in expressions and commands with the DEFINE command. These identifiers represent variables or constants. For example,

```
Edebug 4,5> DEFINE an_integer :: INTEGER = 10
```

defines a debugger variable named an–integer with an INTEGER data type and which is given a value of 10. After this command is executed, the identifier an–integer will yield the value 10 whenever it is used by itself or in an expression.

You can change the value of a defined identifier by using the DEPOSIT command. For example,

```
Edebug 4,5> DEPOSIT an–integer = 20
```

changes the value of an–integer to 20.

The DEFINE command allows the following data types:

INTEGER	simple integer
BOOLEAN	simple TRUE or FALSE
BYTE_DATA(*n*)	*n* bytes of unspecified data
REAL or FLOAT	single-precision floating-point
DOUBLE or GRAND	double-precision floating-point
HUGE	double double-precision floating-point
STRING(*n*)	character string with fixed size of *n* characters
CHAR	STRING(1)
VARYING_STRING(*n*)	character string with varying size of up to *n* characters
RELOCATION	relocation constant

For the data types BYTE_DATA(*n*), STRING(*n*), and VARYING_STRING(*n*), *n* can be given as an expression. For example:

```
Edebug 4,5> DEFINE my–string :: STRING(X + 3)
```

Note that if *n* is not given, it is assumed to be 1.

RELOCATION is a special type similar to INTEGER, but whose values represent program locations. It can be used any time, but is usually used when symbolic debugging is not available. A display of a program location (as with SHOW SESSION) shows the location relative to the RELOCATION data item whose value is closest to the location, if the distance does not exceed 2048 bytes.

For example, the following command defines a RELOCATION constant named Mbase:

```
DEFINE Mbase :: RELOCATION = 2000
```

When the address to be displayed is within 2048 bytes of Mbase, it is displayed as "Mbase+byte-offset." For instance, if the address is 2020, the debugger displays it as "Mbase+20."

When using EDEBUG you may encounter debugger-defined variables wth the same name as a variable in the program you are debugging. To distinguish the two, the defined variable can be preceded with the percent character (%) to tell the debugger to use its internal identifier and not the program variable. For example,

```
my_integer and %my_integer
```

refer to the same defined identifier.

The complete syntax of the DEFINE command is described in the "Command Summary" section, later in this chapter.

Special Predefined Identifiers

The debugger has several identifiers that are predefined. These identifiers allow you to access fixed hardware entities.

For example, the following general purpose registers are predefined:

```
R0, R1, R2, ..., R11, AP, FP, SP, PC
```

In addition, the following identifiers are predefined:

PSL	processor status longword
$	the constant 80000000 (hex)
NIL	the constant 0

As with identifiers defined with the DEFINE command, these predefined identifiers can be prefixed with the percent character (%) if you encounter a conflict with a program's symbol name.

When debugging a program that runs in Kernel mode or within the kernel debug session, the machine processor registers are named P*n*, where *n* is a decimal integer. For example, P1 is the Executive mode stack pointer.

Program Locations and Variable Names

When you are remotely debugging using the EDEBUG command, your program image can provide the debugger with a symbol table, so that you can refer to locations and variables by the names you used within the program.

The debugger provides a generic syntax for expressing program locations and variables. All names consist of two parts:

```
path–name    variable–reference
```

The variable–reference part is your program's name for a variable or location; path–name provides a way to qualify variable–reference to a particular module or routine in the program.

In most cases, path–name is not needed, because the debugger makes some assumptions about the path name. The default path name is referred to as the *reference scope* or *view scope*. This scope is established automatically by the debugger, based on where your session is stopped.

For example, if your program is stopped in a module called driver inside a routine named initialize, the view scope is set to duplicate the compiler's view of the variable reference scope for the routine initialize. Whatever you can refer to inside the routine can be referred to without a path name.

It *is* necessary, however, to specify a path name when you want to refer to a name that is not visible inside the routine where the session is stopped; for instance, another module.

The general form of path–name is

```
module–name \ routine–name \ ...
```

where module–name is the module in which the variable is defined, and the optional list of routine names can be used to further qualify the path name if the variable is internal to some routine.

For example, the path name for routine initialize in module driver, referred to previously, is:

```
driver\ initialize \
```

In some cases, it is enough to specify only the module name, and in other cases, it is necessary to specify the entire sequence of routine names that identify the area of the program you are interested in. The following is an example of a fully expressed path name for the variable file–type:

```
file–io\ open–routine \ file–type
```

Path names can be used to identify locations within a program. For example,

```
declare\%LINE 10
```

refers to line 10 in module declare. The module name and backslash can be omitted if the debugger is currently stopped at a point in module declare.

Note: In highly optimized programs, the compiler may have eliminated all the code at a particular location, even though the source module has explicit statements there.

You can also refer to a program location by a label, as follows:

```
outmodule\%LABEL errormessage
```

Again, the module name and backslash are unnecessary if the label is defined in the current view scope.

The debugger assumes that the view scope is associated with where your session is stopped. Since this may not always be a convenient place, the debugger provides a means to change the view scope. The scope can be moved back and forth along the call history stack by using the commands PREDECESSOR and SUCCESSOR. PREDECESSOR moves backward in history and SUCCESSOR moves forward. That is, PREDECESSOR moves the view scope back to the caller of the routine where your session is stopped and SUCCESSOR moves the view scope forward again.

When the actual stopping point differs from the view scope, the debugger displays both the view scope and the place where your session is stopped. When your program stops in a location that is not part of the known program, the debugger automatically searches back on the call history stack until it finds the last active place in the program.

Variable References

The debugger provides a generic syntax for specifying variable names that is useful to all programmers. Depending on the source language, it may or may not be identical to the way you referred to the name in your program. Simple names are:

```
i
my–variable
his–variable
```

Structure members must always be fully specified, as in:

```
rec1.item2
rec1.item3.value1
```

Array elements can be expressed in two notations:

```
array1(1,2,3)
array1[1,2,3]
```

Pointer qualified references can be made in any of these forms:

```
ptr–variable^
ptr–variable→
*ptr–variable
```

Types and Typecasting

Each expression, value, or variable reference in a debugger command line has an intrinsic data type, which affects the item's memory size, display format, and expression semantics. In addition to their usage in the DEFINE command (explained previously under "Identifiers"), types are explicitly used in several other places within the command language.

In the EXAMINE and DEPOSIT commands, types are used as qualifiers to specify a type to use in the display or deposit of a value. For example,

```
Edebug 4,5> EXAMINE/REAL counter
```

causes counter to be examined and displayed as a floating-point value regardless of its true type. Here, the item's intrinsic memory size is also being specified; therefore, the debugger examines 4 bytes regardless of the actual size of counter. (For more information, see the descriptions of the EXAMINE and DEPOSIT commands, later in this chapter.)

Type names can be specified as part of a variable reference to "cast" the reference's intrinsic type to another. This is useful inside expressions to force particular semantic interpretations. For example:

```
Edebug 4,5> EVALUATE gas_v >= counter :: REAL
```

This command asks to evaluate the Boolean expression that compares the variable gas_v with counter. However, to make sense, it is necessary to interpret counter as a real number rather than its implied integer type. This syntax is referred to as *typecasting*. It is always specified as double colons (::) followed by a type specifier.

The following shows the relationship of typecasting to types as EXAMINE command qualifiers:

```
Edebug 4,5> EXAMINE/REAL counter
```

has the same effect as

```
Edebug 4,5> EXAMINE counter :: REAL
```

The debugger does not attempt to understand all data types available in all programming languages. Rather, it understands and operates on several basic computational types. When interpreting a program's symbol table information, the debugger translates the language-specific data type into one of its generic data types. You can determine the debugger's interpretation of a variable by using the SHOW SYMBOL command.

The types used by EDEBUG as defined variable types, qualifiers for EXAMINE or DEPOSIT, and typecasting include: INTEGER, BOOLEAN, BYTE_DATA(*n*), REAL or FLOAT, DOUBLE or GRAND, HUGE, STRING(*n*), CHAR, VARYING_STRING(*n*), and RELOCATION. These types are defined under "Identifiers," earlier in this section.

Some of these type names (STRING, for example) do not have an associated static size. In these cases, it may be necessary to specify a size value. For example:

```
STRING(10)
```

or

```
BYTE_DATA(100)
```

Note that the size value can be given as an expression, which is interpreted when the type specifier is used. For example:

```
Edebug 4,5> EXAMINE d_block :: STRING(50 DIV ctr)
```

If no size value is given, it is assumed to be 1.

Computational Constants

This subsection defines the syntax rules for the computational constants supported by the VAXELN debugger.

Boolean Constants

Boolean constants are the reserved identifiers TRUE and FALSE, denoting the numeric value 1 and 0, respectively.

Integer Constants

Integer constants are strings of characters beginning with one of the digits 0 to 9. Numbers are interpreted in the current radix. For instance, 1234 or 1FEF4 are examples of integer-valued constants. The radix in use for a particular debug session is dictated by the place in the program where the session is stopped and any explicit setting of the radix done with the SET MODE command.

A radix can be specified explicitly by prefixing the numeral string with a % construct. The explicit radix specifiers are:

%X or %HEX	for hexadecimal
%D or %DEC	for decimal
%O or %OCT	for octal
%B or %BIN	for binary

For example:

```
%O177440
```

The numeral string after the % construct can be enclosed in quotes to allow "white space" inside the string. For example:

```
%X'ff ff ff ff'
```

When the default radix is hexadecimal, the debugger will interpret any unrecognized identifier as a hexadecimal number. This feature helps avoid the need to specify %X. For example, the following are all valid hexadecimal constants:

```
FE00
0FE00
%Xfe00
```

Note that integer constants must always be in the range -2^{31} to $2^{31}-1$, or in the range 0 to $2^{32}-1$.

Floating-Point Constants

Floating-point constants are given in FORTRAN format if the default radix is decimal. For example:

```
1.24
1.2e10
1.234e - 10
- 12.4
```

If the default radix is not decimal, the floating-point constants must be given using the % construct. For example:

```
%F'1.24'
%F'1.2e10'
%F'1.234e - 10'
%F' - 12.4'
```

In this case, the numerals within the quotes are always interpreted as decimal numbers.

Floating-point constants are always converted to the VAX double-precision format. Depending on the machine model you are using, the range of floating-point numbers can vary. Normally, the debugger uses the VAX DOUBLE floating-point format, providing for approximately 16 digits in the range .29E−38 to 1.7E38.

On some VAX machine models, another double-precision floating-point data type is available; this type is referred to as GRAND. It has a smaller precision, but a larger exponent range, providing for approximately 15 digits in the range .56E−308 to .9E308.

You can change the default floating constant type with the SET MODE command.

String Constants

String and character constants are sequences of characters enclosed in apostrophes; a pair of consecutive apostrophes ('') inside a string constant represents a single apostrophe character. String constants in the debugger can have up to 132 characters. Note that a string constant must not span multiple lines.

Special Constants

In address expressions, a period character (.) is shorthand for the address of the source for the last EXAMINE operation or of the target of the last DEPOSIT operation.

Several things that can be examined in some way are not actually addressable, in which case the debugger will issue an error message saying that the period character is meaningless. For example, variables that are not stored on byte boundaries are not addressable.

The backslash character (\) represents the value of the most recent expression.

RELOCATION is a special type whose values represent program locations, as explained under "Identifiers," earlier in this section.

Comments

In command lines, anything after the exclamation point character (!) is ignored and can be used for comments.

Command Summary

Strictly speaking, the debugger command set is the same whether you are debugging remotely (with EDEBUG, at the host) or locally (with the local debugger and console terminal). Only EDEBUG, however, can do symbolic debugging and source-line displays (that is, only EDEBUG can access the debugger symbol table and program source text).

The debuggers will inform you if a particular command is not possible in your current mode (for example, EXAMINE/SOURCE in local debugging).

The remainder of this section contains descriptions of the commands used in debugging VAXELN systems. In the command descriptions, optional parts of the command are noted in *italics*.

CALL

Call the specified routine in the context of the current program and return to command mode when the routine completes.

CALL target *(argument-list)*

The target of a call can be:

pathname identifier

integer

or

(address–expression)

The address expression *must* be parenthesized and is used when the name to be specified is not a simple identifier.

The arguments to the routine, if any, are separated by commas, and the list must be parenthesized. There can be up to 16 arguments. After their evaluation, each argument must fit in a longword (32 bits).

Note: If the routine does anything to affect the state of the running program, the debugger may lose control of the session. In addition, the debugger will not field breakpoints or exceptions while the routine is executing.

The CALL command is useful when the debugger is unable to perform some particular or complex function for your debugging. A typical example might be in debugging a command interpreter, where the CALL command could be used to call a routine you have in your application that dumps the symbol table.

The value returned in R0 by the called routine is automatically displayed after the routine executes.

For example:

```
Edebug 4,5> CALL (dump-symbol) (@R0)
Symbol table entry at: 43450  Name: test1
     Type: integer  Value: 2
 16 (00000010)
```

Other examples to show the special features of the CALL command syntax:

```
Edebug 4,5> CALL %x200
 16 (00000010)
Edebug 4,5> CALL external-call (1,0,false)
Parameters in HEX are  00000001  00000000  00
 1 (00000001)
Edebug 4,5> CALL (rtn-ptr + 0) (1,0,false)
 2 (00000002)
```

CANCEL BREAK

Cancel the breakpoint at the given address, or cancel all breakpoints. The KERNEL qualifier causes the command to cancel breakpoints set by the SET BREAK /KERNEL command.

```
CANCEL BREAK  address–expression
CANCEL BREAK  /ALL
CANCEL BREAK  /KERNEL    address–expression
```

See the SET BREAK command description for examples of CANCEL BREAK.

CANCEL CONTROL

This command causes processes that start in this session's job to begin independently of the debugger.

```
CANCEL CONTROL
```

CANCEL EXCEPTION BREAK

This command reestablishes the default exception handler search for the associated session. It reverses the action of the SET EXCEPTION BREAK command.

```
CANCEL EXCEPTION BREAK
```

CREATE JOB

Create a job in the target system, running the designated program. program–name is specified as a string expression (that is, it is the name that appears on the System Builder's map listing, enclosed in apostrophes).

CREATE JOB program–name *(argument-list)*

The arguments to the program, if any, are separated by commas, and the list must be parenthesized. There can be up to 16 arguments, all of which are strings. The arguments can be constants, DEFINE values, or variables of string types.

For example:

```
Edebug 4,5> CREATE JOB tstjob ('1','2','3')
```

CREATE PROCESS

Call the specified routine in the context of the current program and start the procedure as a process. The response to the command is either nothing (implying success) or else the error message for the failure status is displayed.

CREATE PROCESS   target *(argument-list)*

The target can be:

pathname identifier

integer

or

(address_expression)

The address expression *must* be parenthesized.

The arguments to the routine, if any, are separated by commas, and the list must be parenthesized. There can be up to 16 arguments. After their evaluation, each argument must fit in a longword (32 bits).

CTRL/C

Typing CTRL/C aborts the operation in progress and, except when debugging the kernel, gains the attention of the debugger's command interpreter.

CTRL/Z

Typing CTRL/Z is equivalent to EXIT.

DEBUG

Debug, connect to, or load and debug a system on another node.

```
DEBUG nodename
```

Here, the system on the indicated node is triggered to boot the system specified in the network database.

```
DEBUG/LOAD = system nodename
```

Here, system is set into the network database for nodename and the node is triggered to boot.

DEFINE

Create or redefine a session variable with a specified *type* and an initial value supplied by *expression*. If *type* is omitted, INTEGER is the default. If *expression* is omitted, 0 is the default initial value. You can redefine session variables at any point.

DEFINE identifier *:: type: = expression*

DEFINE identifier *:: type = expression*

Variables defined this way are available in all sessions. If a program is associated with a session, a reference to one of these identifiers is resolved to the session variable only after looking in the current reference scope. If the name is prefixed with %, however, it refers unambiguously to the session variable.

The permanent symbols (R*n*, P*n*, PSL, SP, AP, FP, and PC) cannot be redefined. When used in expressions, these symbols always yield values, like register names, and are never addressable.

Examples of using DEFINE variables:

```
Edebug 4,5> DEFINE iii
Edebug 4,5> iii
 iii:  0      (00000000)
Edebug 4,5> DEPOSIT iii = 2
Edebug 4,5> iii
 iii:  2      (00000002)

Edebug 4,5> DEFINE ptr1 :: ^integer : = nil

Edebug 4,5> DEFINE s1 :: string(10) = ''

Edebug 4,5> DEFINE bd1 :: byte_data(30) = s1

Edebug 4,5> DEFINE rtn_ptr :: relocation
Edebug 4,5> deposit rtn_ptr = address(external_call)
Edebug 4,5> EXAMINE/INSTRUCTION rtn_ptr + 2
 RTN_PTR + 2: MOVAB  -40(FP),SP

Edebug 4,5> DEFINE relo :: relocation = %x200
Edebug 4,5> EXAMINE/BYTE:40 relo
 18000300 00000010 00000004 00003039 | RELO + 0000 | 90..............
 00000000 00000000 00000000 00000014 | RELO + 0010 | ................
                   00000000 00000061 | RELO + 0020 | a.......
```

DELETE PROCESS

Delete a process associated with a session, where the process is specified by job name, job ID, and process ID. If *process_specifier* is not supplied, the process associated with the current session is deleted. The optional *nodename* is used to identify a process on a different node, if you are debugging several at once.

DELETE PROCESS *process_specifier nodename*

You can only use DELETE PROCESS to delete processes associated with sessions that are stopped. If necessary, use the HALT command to stop the session.

DEPOSIT

Deposit the value of an expression in a location described by a variable reference or address expression.

```
DEPOSIT /qualifier address–expression : = expression
```

The address expression can be given in parentheses if it is complex. The = symbol can be used identically with := symbol.

One of the following qualifiers can be used to specify the target's representation. If a qualifier is not given, the target is assumed to be an integer or, if the target is a simple variable reference by itself, the type of that variable or any typecast given.

Specifying that the target is a string:

/ASCII:size

specifies that the target is a string of length *size*. The *size* specifier can be given as:

```
:integer
 =integer
:(expression)
 =(expression)
```

If */ASCII* is used without a length, it is assumed to be 1.

Specifying that the target is an arbitrary sequence of bytes:

/BYTE:size

specifies that the target is a sequence of *size* bytes.

The *size* specifier can be given as:

```
:integer
=integer
:(expression)
=(expression)
```

If */BYTE* is used without a size, it is assumed to be 1.

Specifying that the target is an integer:

/BYTE	*/WORD*	*/LONGWORD*	*/QUADWORD*

Specifying that the target is a floating value:

/REAL	*/FLOAT*		
/DOUBLE	*/GRAND*	*/D–FLOAT*	*/G–FLOAT*
/HUGE	*/H–FLOAT*		

Some implicit conversion is performed for you at deposit time and if the conversion cannot be done, an error is given. The rules for the conversion are:

1. If the target is an integer or boolean, then the source must be integer or boolean.
2. If the target is floating, then the source must be integer or floating. The integer value is converted to floating.
3. If the target is string, then the source must be string or byte data. The source is padded with blanks or truncated to fit the target.
4. If the target is byte data, then the source must be string or byte data. The source is padded with zeros or truncated to fit the target.

Examples:

```
Edebug 4,5> DEPOSIT 123+456 = 10
Edebug 4,5> DEPOSIT my–variable = 15
Edebug 4,5> DEPOSIT/ASCII:10 %x200 = 'abcdef'
Edebug 4,5> DEPOSIT data–block–item.packet[i] : = i
```

EVALUATE

Evaluate the given expression. The qualifiers BINARY, OCTAL, DECIMAL, and HEX can be used to specify the display radix if the expression result is an integer. The ADDRESS qualifier displays the effective address of a specified address expression. This address would be the one examined if the same address expression were specified with an EXAMINE command.

```
EVALUATE expression
EVALUATE/BINARY expression
EVALUATE/HEX expression
EVALUATE/DECIMAL expression
EVALUATE/OCTAL expression
EVALUATE/ADDRESS addresss–expression
```

The following examples exemplify the fact that the EVALUATE keyword is not always necessary:

```
Edebug 4,5> (%x1f <> 31) or (%B'11111' <> %o37)
 FALSE

Edebug 4,5> b1 <> (b2 and false)
 TRUE
```

Other examples of EVALUATE expressions:

```
Edebug 4,5> EVALUATE 100 + 200
  300 (0000012C)

Edebug 4,5> EVALUATE /BINARY 100 + 200
  100101100 (0000012C)

Edebug 4,5> EVALUATE/HEX 100 + 200
  0000012C
```

```
Edebug 4,5> EVALUATE/DECIMAL 100 + 200
  300 (0000012C)

Edebug 4,5> EVALUATE/OCTAL 100 + 200
  454 (0000012C)

Edebug 4,5> EVALUATE/ADDRESS var–10
  7FFFD2D4

Edebug 4,5> EVALUATE/BINARY/ADDRESS var–10
  1111111111111111101001011010100 (7FFFD2D4)

Edebug 4,5> EVALUATE/HEX/ADDRESS var–10
  7FFFD2D4
```

Examples using \ (last value):

```
Edebug 4,5> counter + 50 * 100
 100600
Edebug 4,5> 2 + \
 100602
Edebug 4,5> 2 + \
 100604
```

EXAMINE

Examine the value in a location in the target system's memory. The location is specified by an address expression, or, if the debugger symbol table is present, a variable reference.

EXAMINE commands have the general form:

EXAMINE *qualifier-list*   *address–expression*

ASCII, BYTE, WORD, QUAD, LONG, and the floating-point qualifiers can be used to specify the amount of storage examined, overriding the size associated with the item's type and, in the case of ASCII and the floating-point qualifiers, determining the format of the display. If there is no associated type, INTEGER (longword) is the default.

With BYTE and ASCII, the integer expression *size* can be specified to override the basic size of the data item (if the expression is not just an integer constant, it must be enclosed in parentheses). The *size* specifier can be given as:

```
:integer
=integer
:(expression)
=(expression)
```

Examining a location with `BYTE:size` specified causes the debugger to display the data in a "dump" format consisting of a hexadecimal display on the left and an ASCII display on the right.

The qualifiers BINARY, OCTAL, DECIMAL, and HEX can be used to specify the radix in which the result is displayed if the result is an integer. If the address expression is a variable reference, the variable's type determines the default display.

If no expression is given after the verb, the action is to examine the next value after the one most recently examined. If the address expression is the symbol ˆ, the action is to examine the value before the one most recently examined.

Examples of EXAMINE commands:

```
Edebug 4,5> EXAMINE/ASCII s1
 S1: a

Edebug 4,5> EXAMINE/ASCII:10 s1
 S1: abc

Edebug 4,5> EXAMINE/BY s1
 S1:  97 (00000061)
```

```
Edebug 4,5> EXAMINE/BY:1 s1
                                                 61 | S1 | a

Edebug 4,5> EXAMINE/BY:3 s1
                                             636261 | S1 | abc

Edebug 4,5> EXAMINE/BY:10 s1
                              2020 20202020 20636261 | S1 | abc

Edebug 4,5> EXAMINE/BY:20 s1
 00064C50 00002020 20202020 20636261 | S1 | abc          ..PL..
                                           00064CD0 | S1 + 0010 | .L..

Edebug 4,5> EXAMINE/WORD s1
 S1:  25185 (00006261)

Edebug 4,5> EXAMINE/QUAD s1
                                  20202020 20636261 | S1 | abc

Edebug 4,5> EXAMINE/LONG s1
 S1:  543384161 (20636261)

Edebug 4,5> EXAMINE/DOUBLE r0
 R0:     1.0000000000000000E+00005

Edebug 4,5> EXAMINE/G-FLOAT r0
 R0:     8.41178011028559882E+00041

Edebug 4,5> EXAMINE/HUGE r0
 R0:                 1.0649995818682317E+00675

Edebug 4,5> EXAMINE/OCTAL i1
 I1:  1 (00000001)

Edebug 4,5> EXAMINE/BINARY i1
 I1:  1 (00000001)
```

```
Edebug 4,5> EXAMINE/HEX i1
 I1: 00000001

Edebug 4,5> EXAMINE/BY:3/OCTAL i1
          00000001 | I1 |...

Edebug 4,5> EXAMINE/BY:4/BINARY i1
                     1 | I1 |....

Edebug 4,5> EXAMINE data-block-item
 NEXT-DATA-BLOCK:  7FFFD3AC
 PACKET(1):  33 (00000021)
 PACKET(2):  2 (00000002)
 PACKET(3):  0 (00000000)
 PACKET(4):  0 (00000000)
 BIT-ITEM: True

Edebug 4,5> EXAMINE arr-item-10
 ARR-ITEM-10(1):  33 (00000021)
 ARR-ITEM-10(2):  2 (00000002)
 ARR-ITEM-10(3):  0 (00000000)
 ARR-ITEM-10(4):  0 (00000000)
 ARR-ITEM-10(5):  0 (00000000)
 ARR-ITEM-10(6):  0 (00000000)
 ARR-ITEM-10(7):  0 (00000000)
 ARR-ITEM-10(8):  0 (00000000)
 ARR-ITEM-10(9):  0 (00000000)
 ARR-ITEM-10(10):  805306368 (30000000)
```

Examples of EXAMINE next, . (dot) and ^ (previous):

```
Edebug 4,5> EXAMINE r1
 R1:    300    (0000012C)
Edebug 4,5> EXAMINE .
 R1:    304    (00000130)

Edebug 4,5> EXAMINE %x4000
 4000:  300    (0000012C)
```

```
Edebug 4,5> EXAMINE . + %x10
 4010: 34    (00000130)

Edebug 4,5> EXAMINE r0
 R0:    10    (0000000A)
Edebug 4,5> EXAMINE .
 R0:    10    (0000000A)
Edebug 4,5> EXAMINE
 R1:    20    (00000014)
Edebug 4,5> EXAMINE ^
 R0:    10    (0000000A)
```

EXAMINE/INSTRUCTION

Examine machine-instruction sequence beginning at the first address specified. If the second address is supplied, the sequence of instructions in that range is displayed; if omitted, one instruction is displayed. If neither address is specified, the instruction at the location after the last-examined location is displayed.

EXAMINE/INSTRUCTION *address : address*

Examples of EXAMINE/INSTRUCTION:

```
Edebug 4,5> EXAMINE/INSTRUCTION %x4000
 4000: MOVAB    - 10(fp), - 14(fp)
Edebug 4,5> SET BREAK .

Edebug 4,5> EXAMINE/INSTR %label first–label :
More?> %label first–label + 40
 %Line 345 + 0000: PUSHL #0B
 %Line 345 + 0002: PUSHL #00
 %Line 345 + 0004: PUSHAB $CODE + 00AE
 %Line 345 + 0008: PUSHL #0B
 %Line 345 + 000A: PUSHAB PAS$OUTPUT\$DATA + 0010
 %Line 345 + 0010: CALLS #05,@0000166C
 %Line 345 + 0017: PUSHAB PAS$OUTPUT\$DATA + 0010
 %Line 345 + 001D: CALLS #01,@00001664
```

```
Edebug 4,5> EXAMINE/i
 %Line 349 + 0005: MOVC3 #1B,$CODE + 0093, - 16E8(FP)
```

EXAMINE/PSL

Examine the value at the location specified by the address expression and expand the value into an ASCII display of a processor status longword (PSL).

EXAMINE/PSL *address–expression*

For example:

```
EXAMINE/PSL @SP
```

The following is another example of EXAMINE/PSL, with the corresponding ASCII display:

```
Edebug 4,5> EXAMINE/PSL PSL
CM TP FPD IS Current Mode Previous Mode IPL DV FU IV T N Z V C
--- -- --- -- ------------ ------------- --- -- -- -- - - - - -
 0  0   0  0            3             3   0  1  0  1 0 0 0 0 0
```

EXAMINE/SOURCE

Examine source-line sequence beginning at the first address specified. If the second address is supplied, the sequence of lines in that range is displayed. If the second address is omitted, one line is displayed. If neither address expression is specified, the next source line is displayed.

EXAMINE/SOURCE *address : address*

This command is available only in EDEBUG.

Examples of EXAMINE/SOURCE:

```
Edebug 4,5> EXAMINE/SOURCE %label first–label
 Module  TSTEDEBUG
  345:  writeln('first label');
```

```
Edebug 4,5> EXAMINE/SOURCE %label first_label :
More?> %label first_label + 40
 Module TSTEDEBUG
 345: writeln('first label');
 346:
 347: call_successor_label:
 348:
 349: p_vs1 := 'this is the main body value';

Edebug 4,5> EXAMINE/SOURCE
 Module TSTEDEBUG
 350: successor;
```

EXIT

Exit the debugging session.

```
EXIT
```

When you are debugging a system remotely, the EXIT command disconnects EDEBUG from the system but sets up a means for you to reconnect later, with any processes under debugger control in the same states as when you disconnected. Breakpoints, for example, are preserved.

CTRL/Z is also interpreted as EXIT.

GO

Proceed with the execution of the session. If an address is specified, execution continues at that address, with no guarantees about the integrity of the program state.

GO *address_expression*

HALT

Stop the current or specified process by raising an asynchronous exception. If the process is currently under the debugger's control or does not handle the exception, it enters the debugger command state; otherwise, it may abort. The optional *nodename* is used to identify a process on a different node, if you are debugging several at once.

```
HALT process–specifier nodename
```

The debugger performs the HALT operation with a VAXELN kernel service that signals the target process. Once the signal takes effect, the debugger gains control and the process enters the debug command wait state.

Note: There are several states in which a process cannot be halted:

- The process is not runnable because of the scheduling state of the system; the process must execute to halt.
- The process is waiting for an ACCEPT_CIRCUIT or CONNECT-CIRCUIT to complete.
- The process is in an implicit wait state during a SEND to a circuit that is full.
- The process is a kernel mode process running at an elevated Interrupt Priority Level (IPL).

In each of these cases, the debugger gains control when the process leaves the blocking state.

HELP

Display the command and syntax summary.

```
HELP
```

In EDEBUG, the HELP items are the same as in the VMS HELP command.

IF

Conditionally execute a one-line command based on the value of a Boolean expression. (Commands created with SET COMMAND can be used.)

IF boolean–expression *THEN* one–line–command

Examples of the IF command:

```
Edebug 4,5> IF a = (123 + r0) THEN DEPOSIT C : = 25

Edebug 4,5> IF a = (123 + r0) THEN dump–symbols
```

See the LEAVE command description for other examples of the IF command.

LEAVE

Leave the execution of a substituted command. LEAVE is useful as the one-line command of IF.

LEAVE

Examples of the IF command using LEAVE:

```
Edebug 4,5> IF a = (123 + r0) THEN LEAVE

Edebug 4,5> IF a = (123 + r0) LEAVE
```

LOAD

Install a new program image into the target system. The new program image can then be executed using the CREATE JOB command. This command is a simplified interface to the program load facilities and is not available in the local debugger.

LOAD */debug /kernel[= n] /job–priority = n*
  program–name file–name *node*

Note: The program image file is opened in the context of the target system and not the context of the

EDEBUG debugger. Be careful to specify an appropriate file name for that context.

PREDECESSOR

Move the current session's reference scope a given number of call frames "back" in the calling order. If the expression is omitted, 1 is the default. For example, PREDECESSOR 1 allows you to use variable names or other names declared in the routine that called the current routine. The context in which the session is currently stopped is not affected.

PREDECESSOR *expression*

SEARCH

Search the current program source for the specified string or identifier. The command argument **target** is either a string or an identifier. The debugger displays the source line or lines containing occurrences of the search target.

If the ALL qualifier is used, all occurrences in the range are displayed. NEXT is the default and displays the first occurrence in the specified range.

SEARCH */NEXT range target*

SEARCH /ALL *range target*

If either *range* or *target* is specified, the other must be. If they are omitted, the search applies to the module most recently searched, if any, beginning at the first line after the one most recently displayed and continuing to the end of that module. Otherwise, the range is specified any of the following ways:

module	Search the named module beginning at its first line and continuing to its end.

module\line	Begin the search at the specified line number in the named module.
module\line:line	Search the specified range of line numbers in the specified module.
line	Begin the search at the given line number in the current module.
line:line	Search the specified range in the current module.

Examples of the SEARCH command:

```
Edebug 4,5> SEARCH tstedebug\ 1 : 100 'a'
 Module  TSTEDEBUG
 5: procedure macsub; separate;
Edebug 4,5> SEARCH 1 : 100 var
 Module  TSTEDEBUG
 6: function foraddf(var ii,jj : integer);
Edebug 4,5> SEARCH
 Module  TSTEDEBUG
 7: procedure foradds(var ii,jj : integer);
Edebug 4,5> SEARCH /ALL
 Module  TSTEDEBUG
 20:  var in-out-str : string(<k>))
 100:  var param-string : string(<k>))
```

SET BREAK

Set a breakpoint at the specified address. Optionally specify a single command to be executed when the break occurs. Commands defined by SET COMMAND are allowed here.

The JOB and ALL qualifiers both specify that the breakpoint is valid for the entire job; otherwise, the breakpoint affects only the current session's process.

The KERNEL qualifier (not available from EDEBUG) sets a breakpoint in the kernel's breakpoint data base;

it can be used only with kernel mode programs, and no command can be specified. The breakpoint is set in the process memory and in the kernel's copy of the program running in that process.

KERNEL can be used to set breakpoints in interrupt service routines. Because the breakpoint is set in the kernel's copy of the program, you can set breakpoints in interrupt service routines merely by referring to their program address space; the command takes care of the rest. Note that the breakpoint occurs when your program is in the kernel, not necessarily in the context of the program (see the SET SESSION command).

```
SET BREAK /JOB      address–expression DO
                    (one–line–command)
SET BREAK /ALL      address–expression DO
                    (one–line–command)
SET BREAK /KERNEL address–expression
```

Note: When a debugger symbol table is present, EDEBUG takes note if you set a breakpoint at a VAX procedure entry point and places the breakpoint at that procedure's first instruction. For example,

```
SET BREAK writeroutine
```

automatically sets the breakpoint at the address of writeroutine+2, the first instruction. If there is no symbol table, you must do this yourself.

Examples of using the SET BREAK and CANCEL BREAK commands:

```
Edebug 4,5> SET BREAK 500 + 512
Edebug 4,5> SET BREAK sym_tbl\ %line 125
Edebug 4,5> SET BREAK sym_tbl\ %label not_found
Edebug 4,5> SET BREAK sym_tbl\srch_rtn\srch_struct
Edebug 4,5> SET BREAK %line 25
Edebug 4,5> EXAMINE/INSTRUCTION %x4000
 4000: MOVAB    - 10(fp), - 14(fp)
Edebug 4,5> SET BREAK .

Edebug 4,5> CANCEL BREAK symbol_not_found
Edebug 4,5> CANCEL BREAK %line 25
```

SET COMMAND

Create a command for use during a session. The optional text *one–line–command* is substituted for every subsequent occurrence of identifier in a command context. If the one-line command is omitted, you are prompted for a sequence of debugger commands, which you terminate by typing an empty line. Then, the entire sequence is performed when the command identifier appears in a command context.

```
SET COMMAND  identifier DO (one–line–command)
```

You can redefine commands at any point. For example:

```
Edebug 4,5> SET COMMAND r0plus10 do(r0 + 10)
Edebug 4,5> SET COMMAND r1plus10
Command> r1 + 10
Command>
Edebug 4,5> r0plus10
 100    (00000064)
Edebug 4,5> SHOW COMMAND r1plus10
 R1PLUS10 - r1 + 10

Edebug 4,5> SET COMMAND r0plus10 do()   ! erase
   r0plus10 command
```

SET CONTROL

This command reverses the action of the CANCEL CONTROL command.

```
SET CONTROL
```

SET EXCEPTION BREAK

This command causes the associated session in the debugger to stop when any exception occurs. The debugger then gains control before the kernel searches for a programmed exception handler. This differs from the default action, which is to give the debugger control only after no programmed exception handlers have been found. When the session stops at an exception break, a GO command will continue the search for an exception handler.

```
SET EXCEPTION BREAK
```

This state can be cancelled with the CANCEL EXCEPTION BREAK command.

SET LOG

Enable or disable logging of this session in the specified file.

SET LOG *file–specification*

If the file specification is omitted, logging is canceled. If you specify a new file during a session, the old one is closed and the new one receives further logging information.

This command is available only in EDEBUG.

SET MODE

This command is used to change several debugger command modes. Any number of the modes can be changed in a single SET MODE command. The complementary SHOW MODE command can be used to see exactly what the state of the settable debugger modes is.

Default display radix:

```
SET MODE DECIMAL
SET MODE HEXIDECIMAL
SET MODE OCTAL
```

Default Floating-Point Constant Precision

Either of the following commands sets the default floating-point precision to the VAX DOUBLE floating data type. This data type has a precision of about 16 decimal digits in the range: .29E−38 to 1.7E38.

```
SET MODE D_FLOAT
SET MODE DOUBLE
```

Either of the following commands sets the default floating-point precision to the VAX GRAND floating data type. This data type has a precision of about 15 decimal digits in the range: .56E−308 to .9E308.

```
SET MODE G_FLOAT
SET MODE GRAND
```

Note that the GRAND data type is not present on some VAX models. For this reason, the default floating-point mode is DOUBLE.

Substituted Command Verification

The command verify mode controls whether or not substituted commands and commands from command files are displayed as they are executed.

```
SET MODE VERIFY
SET MODE NOVERIFY
```

Step Actions

The step modes control how far the debugger steps for a single step command and what happens when a step is done at the site of a subroutine call. The SET MODE command can be used to firmly set the modes, and each of these modes can also be specified temporarily as a qualifier to the STEP command. These modes can also be changed by the SET STEP command.

Either of the following commands set the step unit to a program source line. This mode is only valid if a program's symbol table is available. Normally, the debugger sets the step unit to LINE if the session is stopped in a section of the program written in a higher-level language; otherwise, the unit is set to INSTRUCTION.

```
SET MODE LINE
SET MODE SOURCE
```

To set the step unit to instructions use:

```
SET MODE INSTRUCTION
```

When a STEP command is given and a subroutine call is encountered, you have the option to step "into" or "over" the subroutine. In both cases, the subroutine is executed. "Over" simply means that you do not get a chance to look at the inner workings of the subroutine. The default is "over."

```
SET MODE INTO
```

specifies that you want to step "into" a subroutine.

```
SET MODE OVER
```

specifies that you want to step "over" subroutines.

Prompting

Because the debuggers use the same terminal as the console I/O, it is sometimes necessary to get the debugger prompts out of the way. The following command tells the debugger to not prompt until a CTRL/C is typed, freeing the terminal for use by another program.

```
SET MODE NOPROMPT
```

Examples

Examples of SET MODE and SHOW MODE:

```
Edebug 4,5> SET MODE OCTAL
Edebug 4,5> 1234
 1234 (0000029C)
Edebug 4,5> SET MODE HEX
Edebug 4,5> 1234
  00001234
Edebug 4,5> SHOW MODE
 Radix is decimal.
 Floating conversion mode is double.
 Step over routine calls by line.
 Automatic commands are not verified.
```

SET PROGRAM

Inform the debugger that this session's job is running a copy of the specified program image. This causes the debugger to copy the image's symbol-table information into its memory and forget the current program, if any. The debugging mode is changed automatically based on the module in which the session is stopped; the radix is set appropriately for the source language.

SET PROGRAM *image–file–specification*

SET PROGRAM is done automatically when a session starts with a debuggable program and EDEBUG is in use. An error message is displayed if the indicated file does not exist. If no file is specified, the current symbol-table information is discarded.

This command affects all sessions associated with the current job. The SET PROGRAM command is seldom necessary, except when the debugger failed to access the program's image fle because it has been moved or because the file's protection prevents access. If the image file is not specified, the program information for the current session is not retained.

SET RETURN BREAK

Stop the session when the current routine returns. This command enables a temporary breakpoint that is encountered when the current routine is about to return.

```
SET RETURN BREAK
```

This state cannot be cancelled with the CANCEL BREAK command.

SET SESSION

Change the session to another known debugging session. The state of the abandoned session remains the same. The optional *nodename* is used to identify a process on a different node, if you are debugging several at once.

The GO qualifier, in effect, gives the GO command to the current session and then changes the session. The KERNEL qualifier starts the kernel debugger.

```
SET SESSION /GO    process–specifier nodename
SET SESSION /KERNEL
```

SET STEP

Change the default action of steps with respect to procedures and functions. INTO and OVER determine whether STEP steps "into" or "over" a routine. SOURCE, LINE, and INSTRUCTION determine the size of the step. (SOURCE and LINE have the same meaning.)

The default behavior of the STEP command, unless modified by SET STEP, is to step "over" one line.

SET STEP *INTO LINE*

SET STEP *OVER LINE*

SET STEP *INTO INSTRUCTION*

SET STEP *OVER INSTRUCTION*

SET STEP *INTO SOURCE*

SET STEP *OVER SOURCE*

See the SET MODE command description for more information on step actions.

SET TIME

Set the system time on the specified node. The time string must be in the standard format for absolute times ('dd-mmm-yyyy□hh:mm:ss.cc').

SET TIME time–string node–name

SHOW BREAK

Display information about all breakpoints.

SHOW BREAK

For example:

```
Edebug 4,5> SHOW BREAK
 Module name   Routine or Psect name       Line   Rel PC     Abs PC
 TSTEDEBUG     TSTEDEBUG                   345    000002F1 00000FBF
 TSTEDEBUG     TSTEDEBUG                   353    00000327 00000FF5
 TSTEDEBUG     TSTEDEBUG                   356    0000034B 00001019
 TSTEDEBUG     TSTEDEBUG                   359    0000036F 0000103D
```

SHOW CALLS

Display the call history for the current variable reference scope.

```
SHOW CALLS
```

For example:

```
Edebug 4,5> SHOW CALLS
 Module name   Routine or Psect name       Line   Rel PC     Abs PC
 TSTEDEBUG     TSTEDEBUG                   502    00000202 00000ED0
 TSTEDEBUG     TSTEDEBUG                   217    00000002 00000CD0
                                                  00000000 800024F5
```

SHOW COMMAND

Display one or all commands defined by SET COMMAND.

```
SHOW COMMAND identifier
SHOW COMMAND /ALL
```

See the SET COMMAND command description for an example of SHOW COMMAND.

SHOW JOB

Display brief information about all processes in a job. The optional *nodename* is used to identify a different node, if you are debugging several at once. The job can be specified by name or number; if no job is specified, the processes in the current job are displayed.

SHOW JOB *string nodename*

SHOW JOB *identifier nodename*

SHOW JOB *integer nodename*

Examples of SHOW JOB:

```
Edebug 4,5> SH JOB 6
 Job 6, program TESTLOAD, priority 16 is waiting.
  Shared read/write size: 1024.
   Process 1, priority 8, in debug command wait.
     Stack size: 5632. CPU time:   0 00:00:00.02
  Accumulated CPU time for this job:   0 00:00:00.02

Edebug 4,5> SH JOB testload
 Job 6, program TESTLOAD, priority 16 is waiting.
  Shared read/write size: 1024.
   Process 1, priority 8, in debug command wait.
     Stack size: 5632. CPU time:   0 00:00:00.02
  Accumulated CPU time for this job:   0 00:00:00.02
```

SHOW MESSAGE

Display the text associated with the expression value's exit status.

SHOW MESSAGE expression

For example:

```
Edebug 4,5> SHOW MESSAGE %x7c3c
 Bad parameter value
```

SHOW MODE

Display current operating modes of the debugger (stepping defaults, radix, and floating-point conversion type).

```
SHOW MODE
```

See the SET MODE command description for an example of SHOW MODE.

SHOW MODULE

Display information about the program associated with the current session.

```
SHOW MODULE
```

For example:

```
Edebug 4,5> SHOW MODULES
 Program- SEA$:[DEBUG.TEST]TSTEDEBUG.EXE;62

 Module name   Symbols   Language   Source
 TSTEDEBUG     Yes       Pascal     Yes
 FORADDF       Yes       FORTRAN    No
 FORADDS       Yes       FORTRAN    No
 PASSUBS       Yes       Pascal     Yes
 MAIN0         No        Macro      No
 PLISUB        Yes       PL/I       Yes
 BINARY        Some      Bliss      Yes
 PAS$INPUT     No        Macro      No
 PAS$OUTPUT    No        Macro      No
```

SHOW PROCESS

Display the system state of a job or a particular process in a job. The ALL qualifier displays all jobs. If neither is specified, the state of the process associated with the current debugging session is displayed. The optional *nodename* is used to identify a different node, if you are debugging several at once.

SHOW PROCESS *process–specifier nodename*

SHOW PROCESS */ALL nodename*

For example:

```
Edebug 4,5> SH PROCESS
 Job 6, program TESTLOAD, priority 16 is waiting.
  Shared read/write size: 1024.
   Process 1, priority 8, in debug command wait.
     Stack size: 5632. CPU time:   0 00:00:00.02

Edebug 4,5> SH PROCESS 2,4
 Job 2, program XQDRIVER, priority 1 is waiting.
  Shared read/write size: 30720.
   Process 4, priority 8, is waiting.
     Stack size: 2048. CPU time:   0 00:00:00.07
```

SHOW PROGRAM

Display the system's information about an installed program.

SHOW PROGRAM name

SHOW PROGRAM /ALL

For example:

```
Edebug 4,5> SH PROGRAM testload
 Program: TESTLOAD  Debug  User mode
   Default job priority: 16  Default process priority: 8
   Kernel stack size: 8     User stack size: 2
    Filename:  "SEA$:[DEBUG.TEST]TESTLOAD.EXE;75"
```

SHOW SESSION

Display the debug state of one or ALL debugging sessions. If neither is specified, the state of the current session is displayed. The optional *nodename* is used to identify a different node, if you are debugging several at once.

SHOW SESSION *process–specifier nodename*

SHOW SESSION */ALL nodename*

For example:

```
Edebug 4,5> SHOW SESSION 5,1
 Job 5, process 1, program TSTEDEBUG needs attention.
  Module TSTEDEBUG
  216:
 >>217: BEGIN
  218: gbldef_i : = 4;
  219: gbldef_j : = 12345;
```

SHOW SYMBOL

This command provides a way to see what EDEBUG knows about a particular name in the current variable reference scope. It displays the debugger's symbol-table information for the specified symbol. If a path name is supplied without an identifier, all the symbols defined at the end of that path are shown. If an identifier is given without a path name, the name from the current scope is shown.

The DEFINE qualifier displays the information about a debugger-defined symbol. If no identifier is supplied with the qualifier, all of the defined session variables are shown.

SHOW SYMBOL *pathname identifier*

SHOW SYMBOL */DEFINE identifier*

Examples:

```
Edebug 4,5> SHOW SYMBOLS
Outer Scope:
1 GBLDEF–I
    Type:  Integer
    Size: 1 Longword
    Located at address 00000204

1 GBLDEF–J
    Type:  Integer
    Size: 1 Longword
    Located at address 00000200

Routine: TSTEDEBUG
1 APTR
    Type:  Pointer to anytype
    Size: 1 Longword
    Located at  –0000175C(FP)

1 ARR–ITEM–10
    Type:  Array of Integer
    Size:  Not determined at compile time
    Located at  –000015D8(FP)

Edebug 4,5> SHOW SYMBOLS gbldef–i
Outer Scope:
1 GBLDEF–I
    Type:  Integer
    Size: 1 Longword
    Located at address 00000204
```

```
Edebug 4,5> SHOW SYMBOLS /DEFINE
 $: Relocation
 NIL: Relocation
 III: Integer
 PTR1: pointer to Integer
 S1: String(10)
 RELO: Relocation
```

SHOW SYSTEM

Display the memory, CPU time, and jobs of the system. The optional *nodename* is used to identify a different node, if you are debugging several at once.

SHOW SYSTEM *nodename*

For example:

```
Edebug 4,5> SH SYSTEM
 Available: Pages: 1335, Page table slots: 47, Pool blocks: 216
 Time since SET TIME: Idle: 0 00:00:23.10 Total: 0 00:00:23.73
 Time used by past jobs: 0 00:00:00.03

 Job 2, program XQDRIVER, priority 1 is waiting.
 Job 3, program EDEBUGREM, priority 3 is running.
 Job 4, program DUDRIVER, priority 4 is waiting.
 Job 5, program FALSERVER, priority 16 is waiting.
 Job 6, program TESTLOAD, priority 16 is waiting.
```

SHOW TIME

Display the time on the current system or a specified node.

SHOW TIME *nodename*

SHOW TRANSLATION

Display the translation (PORT object value) associated with the given name on the given node. If no node name is specified, the default is the node associated with the current session. A quoted string can be used in lieu of an identifier to allow characters that are not valid in identifiers.

SHOW TRANSLATION identifier *nodename*

SHOW TRANSLATION string *nodename*

For example:

```
Edebug 4,5> SHOW TRANSLATION console
 Node: AA-00-04-00-E0-20, Network: 0, Object: 131316
 000020E0 000400AA 00000000 000200F4 | 00000000 | ..............

Edebug 4,5> SHOW TRANSLATION 'console'
 Node: AA-00-04-00-E0-20, Network: 0, Object: 131316
 000020E0 000400AA 00000000 000200F4 | 00000000 | ..............

Edebug 4,5> SHOW TRANSLATION console sea4
 Node: AA-00-04-00-E0-20, Network: 0, Object: 131316
 000020E0 000400AA 00000000 000200F4 | 00000000 | ..............
```

STEP

Execute the next instruction or line. This command executes the single unit and returns control to the user. If the next instruction or line calls a procedure or function, STEP/INTO will stop in the routine; STEP/OVER will stop after the routine's return. The default, unless modified by the SET STEP command, is to step over one line.

If an associated program has line number information, STEP/LINE (or STEP/SOURCE, since SOURCE and LINE have the same meaning) steps over all the instructions associated with the current line. If there is no associated program or if the session is not at a point where there is a line number, then only one instruction is stepped over.

See the SET MODE command description for more information on step actions.

In any case, the INSTRUCTION qualifier can be used explicitly to step over one instruction.

```
STEP
STEP /INTO /INSTRUCTION
STEP /OVER /INSTRUCTION
STEP /INTO /LINE
STEP /OVER /LINE
STEP /INTO /SOURCE
STEP /OVER /SOURCE
```

All forms of STEP accept an optional expression; for example, STEP 10, which repeats the STEP command 10 times.

SUCCESSOR

Move the current session's variable reference scope a given number of call frames "forward" in the calling order (following use of the PREDECESSOR command). If the expression is omitted, 1 is the default. For example, SUCCESSOR 1 allows you to use variable names or other names declared in the next routine called from the current routine. The context in which the session is currently stopped is not affected.

```
SUCCESSOR expression
```

TYPE

Display the source program lines in a specified range. The range is given by providing a module name, a backslash, and one or two line number values.

If the module name and backslash are not given, the module where the session's variable reference scope is set or the last module used in a TYPE or SEARCH command is assumed. If no line numbers are given, the line after the last TYPE or SEARCH command is displayed.

TYPE *module–name \ expression : expression*

Examples of the TYPE command:

```
Edebug 4,5> TYPE 1
 Module  TSTEDEBUG
 1: module tstedebug;

Edebug 4,5> TYPE 1 : 5
 Module  TSTEDEBUG
 1: module tstedebug;
 2:
 3: VAR      gbldef_i,gbldef_j : integer;
 4:
 5: procedure macsub; separate;

Edebug 4,5> TYPE passubs\ 1 : 6
 Module  PASSUBS
 1: MODULE passubs;
 2: procedure passubs(i,j : integer; k : integer);
 3:
 4: begin
 5:   k := i + j;
 6:   end;
```

UNLOAD

Remove a previously loaded program image from the system. This command is not available in the local debugger.

```
UNLOAD program_name node
```

Appendix A
VAX–11/750 Microcode Patch

System Revision Level 5 of the VAX–11/750 and VAX–11/751 computers allows for the patching of the machine's microcode control store. Without the patch, the 11/750 will run, but not at the latest revision level.

The microcode control store patches must be loaded on system power-up. To do this in VAXELN, you must include a special program in each system you load onto the computer. The program must be compiled on site to take advantage of the latest patch set.

Note: If your development system is not a VAX–11/750, you may have to copy the current patch file into the system's SYS$SYSTEM directory. Once built into a VAXELN system, the program will continue running and reload the microcode in the event of a power failure.

If in doubt about the revision level of your machine, check with your local DIGITAL Field Service representative.

Procedure

Perform the following procedures in this order; they must be performed on a VAX–11/750 running VAX/VMS:

1. Set the default directory to ELN$:

   ```
   $ SET DEFAULT ELN$
   ```

2. Define the command to convert the patch file to an object file:

   ```
   $ SET COMMAND DATATOBJ
   ```

3. Create the patch object file:

   ```
   $ DATATOBJ SYS$SYSTEM:PCS750.BIN-
   PCS750.OBJ
   ```

4. Link the resultant object file with the 11/750 microcode patch utility:

   ```
   $ LINK P750UCODE + PCS750.OBJ + RTLSHARE/LIB-
    + RTL/LIB
   ```

5. Include P750UCODE.EXE with a System Builder program description, with the following characteristics:

Init required	*Yes*
Mode	*Kernel*
Job priority	*1*
Powerfailure exception	*Yes*

Appendix B
Kernel Procedures

The procedures in Table B-1 are performed by the VAXELN kernel.

Internal Call Notation

The following form is shown for constructing calls from languages other than VAXELN Pascal. Each argument has the form:

```
name.access_type + data_type.passing_mechanism + parameter_form
```

The components are defined below.

name:

KER$ procedure name

access_type:

w	written by the procedure
r	read by the procedure

data_type:

lc	longword containing a completion code
l	signed longword
lu	unsigned longword (including EVENT or other object values except PORT)
q	signed quadword (64 bits; time values)
ou	unsigned octaword (128 bits; PORT values)
a	virtual address (pointer values)
t	character-coded text string
vt	varying character-coded text string
zem	entry mask of unbound

passing_mechanism:

r	by reference
v	by immediate value
d	by descriptor

parameter_form: (usually null, indicating a scalar data item of the given type)

s	address of a string descriptor
a	array reference.

Arguments in square brackets ([]) are optional; omitted arguments must be specified with commas or zeros, as applicable to the programming language in use (the argument list cannot be shortened). Arguments that can be repeated are followed by an ellipsis (...).

Table B-1. Kernel Procedures

Call Format	Meaning
KER$ACCEPT_CIRCUIT([status.wlc.r], source_port.rou.r, [connect_port.rou.r], [full_error.rlu.v], [accept_data.rvt.r], [connect_data.wvt.r])	Establish circuit between *source_port* and originator of connection request; if *full_error* is disabled (FALSE, default), SEND will wait implicitly when the partner port is full (otherwise, an error status is returned by SEND); varying strings supply optional data to the originator (*accept_data*) or receive data (*connect_data*). The optional *status* receives the completion status, and the optional *connect_port* specifies a different port on which to make the actual connection.
KER$ALLOCATE_MAP([status.wlc.r], register.wa.r, number.wl.r, count.rl.v, device_object.rlu.v, [spt_address.wa.r])	Allocate *count* UNIBUS or QBUS map registers, returning first register *number*, for DEVICE value *device_object*, and return pointer to first in *register*. The optional *status* receives the completion status, and the optional *spt_address* receives a pointer to the system page table base.
KER$ALLOCATE_MEMORY([status.wlc.r], mempointer.wa.r, size.rl.v, [virtual_address.ra.v], [physical_address.ra.v])	Allocate *size* bytes of memory, optionally beginning at *virtual_address* or *physical_address*, and return in *mempointer*. The optional *status* receives the completion status.
KER$ALLOCATE_PATH([status.wlc.r], register.wa.r, number.wl.r, dev_value.rlu.v)	Allocate UNIBUS buffered datapath for DEVICE *dev_value*, returning register *number*, and pointer to datapath register in *register*.

Table B-1. Continued

Call Format	Meaning
KER$CLEAR_EVENT([status.wlc.r], event.rlu.v)	Set state of *event* to EVENT$CLEARED. The optional *status* receives the completion status.
KER$CONNECT_CIRCUIT([status.wlc.r], source_port.rou.r, [destination_port.rou.r], [destination_name.rt.ds], [full_error.rlu.v], [connect_data.rvt.r], [accept_data.wvt.r])	Request circuit connection between *source_port* and *destination_name* or *destination_port*; if *full_error* is disabled (FALSE, default), SEND will wait implicitly when the partner port is full (otherwise, an error status is returned by SEND); varying strings supply optional data to the destination (*connect_data*) or receive data when the destination accepts (*accept_data*). (The destination must be specified either by NAME or PORT value.) The optional *status* receives the completion status.
KER$CREATE_AREA([status.wlc.r], area_variable.wlu.r, data_pointer.wa.r, area_size.rlu.v, area_name.rt.ds, [virtual_address.ra.v])	Create a new area (of size *area_size*) or map an existing area of memory with a unique *area_name* and return the AREA value in *area_variable*. The variable *data_pointer* receives a pointer to the beginning of the allocated memory. The optional *status* receives the completion status, and the optional *virtual_address* specifies the exact P0 base address.

Table B-1. Continued

Call Format	Meaning
KER$CREATE_DEVICE([status.wlc.r], device_name.rt.ds, [relative_vector.rl.v], [service_routine.rzem.r], [region_size.rlu.v], [region_pointer.wa.r], [register_pointer.wa.r], [adapter_pointer.wa.r], [vector_pointer.wa.r], [interrupt_priority.wlu.r], device_variable.wlu.ra, device_count.rlu.v, [pwr_isr.rzem.r])	Connect to *device_name* interrupt and return the DEVICE value in *device_variable*. The *relative_vector* is an integer from 1 (default) to 128. Interrupt service routines can be supplied for interrupt handling (*service_routine*) and power recovery handling (*pwr_isr*). The variable *region_pointer* receives a pointer to the communication region; its size is supplied by *region_size*. The variables *register_pointer* and *adapter_pointer* receive pointers to the first device control register and first adapter control register, respectively; *vector_pointer* receives a pointer to the interrupt vector. The variable *interrupt_priority* receives the interrupt priority level of the device. The *device_variable* can be a single DEVICE variable or an array of up to 16 elements; the number of elements is specified in *device_count*. The optional *status* receives the completion status.
KER$CREATE_EVENT([status.wlc.r], event.wlu.r, initial_state.rl.v)	Create an event with *initial_state* EVENT$SIGNALED or EVENT$CLEARED, and return EVENT value in *event*. The optional *status* receives the completion status.
KER$CREATE_JOB([status.wlc.r], job_port.wou.r, program_name.rt.ds, [exit_port.rou.r], [argument.rt.ds...])	Create a job running *program_name*, with optional *argument*(s) supplied to the program. The variable *job_port* receives the PORT value of the new job's job port. The value *exit_port* supplies the PORT value of the port that receives notification of the job's termination. The optional *status* receives the completion status.

Table B-1. Continued

Call Format	Meaning
KER$CREATE_MESSAGE([status.wlc.r], message_variable.wlu.r, data_pointer.wa.r, message_size.rlu.v)	Create a message with a buffer of size *message_size,* return a pointer to the buffer in *data_pointer*, and return the MESSAGE value in *message_variable.* The optional *status* receives the completion status.
KER$CREATE_NAME([status.wlc.r], name_variable.wlu.r, string.rt.ds, port.rou.r, scope.rl.v)	Make *string* the name of *port*, with *scope* NAME$LOCAL, NAME$UNIVERSAL, or NAME$BOTH, and return the NAME value in *name_variable.* The optional *status* receives the completion status.
KER$CREATE_PORT([status.wlc.r], port_variable.wou.r, [message_limit.rl.v])	Create a message port able to hold up to *message_limit* (default 4) messages, and return the PORT value in *port_variable.* The optional *status* receives the completion status.
KER$CREATE_PROCESS([status.wlc.r], process_variable.wlu.r, routine.rzem.r, [exit_variable.wlc.r], [argument.rlu.v...])	Create a subprocess running *routine*, with optional *argument*(s) supplied to the subprocess, and return the PROCESS value in *process_variable.* The optional *status* receives the completion status, and the optional *exit_variable* receives the final (exit) status of the subprocess.

Table B-1. Continued

Call Format	Meaning
KER$CREATE_SEMAPHORE([status.wlc.r], semaphore_variable.wlu.r, initial_count.rl.v, maximum_count.rl.v)	Create a semaphore with the specified *initial_count* and *maximum_count*, and return the SEMAPHORE value in *semaphore_variable.* The optional *status* receives the completion status.
KER$CURRENT_PROCESS([status.wlc.r], process_variable.wlu.r)	Return the value of the current process in *process_variable.* The optional *status* receives the completion status.
KER$DELETE([status.wlc.r], system_value.rlu.v)	Delete the AREA, DEVICE, EVENT, MESSAGE, NAME, PORT, PROCESS, or SEMAPHORE value supplied by *system_value* from the system. The optional *status* receives the completion status. Note: PORT values are octawords passed by reference (port_value.rou.r).
KER$DISABLE_ASYNCH_EXCEPTION([status.wlc.r])	Disable delivery of asynchronous exceptions to the calling process. The optional *status* receives the completion status.
KER$DISABLE_SWITCH([status.wlc.r])	Disable process switching for the job from which it is called. The optional *status* receives the completion status.
KER$DISCONNECT_CIRCUIT([status.wlc.r], port.rou.r)	Break circuit, where *port* is the one in the current job. The optional *status* receives the completion status.
KER$ENABLE_ASYNCH_EXCEPTION([status.wlc.r])	Allow delivery of asynchronous exceptions to the calling process. The optional *status* receives the completion status.

Table B-1. Continued

Call Format	Meaning
KER$ENABLE_SWITCH([status.wlc.r])	Resume process switching for the calling job. The optional *status* receives the completion status.
KER$ENTER_KERNEL_CONTEXT([status.wlc.r] routine.rzem.r, argument_block.rlu.ra)	Call *routine* in kernel mode, with *argument_block* supplying the address of the VAX argument list to be passed to the routine; the argument list is a block of longwords in standard VAX format: a longword containing the argument count, followed by the argument longwords themselves. The optional *status* receives the completion status of *routine.*
KER$EXIT([status.wlc.r], [exit_status.rlc.v])	End current process, with optional *exit_status* delivered to creator. The optional *status* receives the completion status.
KER$FREE_MAP([status.wlc.r], count.rlu.v, number.rlu.v, device_object.rlu.v)	Free *count* UNIBUS or QBUS map registers, starting with register *number*, previously allocated by KER$ALLOCATE_MAP for DEVICE value *device_object.* The optional *status* receives the completion status.
KER$FREE_MEMORY([status.wlc.r], size.rlu.v, virtual_address.ra.v)	Free *size* bytes of memory at *virtual_address*, previously allocated by KER$ALLOCATE_MEMORY. The optional *status* receives the completion status.
KER$FREE_PATH([status.wlc.r], number.rlu.v, device.rlu.v)	Free UNIBUS datapath *number*, previously allocated for DEVICE *device* by ALLOCATE_PATH. The optional *status* receives the completion status.

Table B-1. Continued

Call Format	Meaning
KER$GET_TIME([status.wlc.r], time.wq.r)	Return current system time in *time*. The optional *status* receives the completion status.
KER$GET_USER([status.wlc.r], [circuit.rou.r], [username.wvt.r], [uic.wlu.r])	Return *username* and/or *uic* of either the calling process or the partner process connected by a circuit to the caller's port; the PORT value of the partner process's port is supplied in *circuit*. The optional *status* receives the completion status.
KER$INITIALIZATION_DONE([status.wlc.r])	Inform kernel that current process has completed initialization sequence. The optional *status* receives the completion status.
KER$JOB_PORT([status.wlc.r], port_variable.wou.r)	Return PORT value of caller's job port in *port_variable*. The optional *status* receives the completion status.
KER$MEMORY_SIZE([status.wlc.r], memory_size.wlu.r, free_size.wlu.r, largest_size.wlu.r)	Scan the kernel memory database and return the size of the initial main memory in *memory_size*, the size of the current free memory in *free_size*, and the size of the largest physically contiguous block of free memory in *largest_size*. The optional *status* receives the completion status.
KER$RAISE_DEBUG_EXCEPTION([status.wlc.r], job_id.rlu.v, process_id.rlu.v)	Raise the asynchronous exception KER$_DEBUG_SIGNAL in the specified context. The optional *status* receives the completion status.

Table B-1. Continued

Call Format	Meaning
KER$RAISE_EXCEPTION([status.wlc.r], name.rl.v, [argument.rl.v...])	Raise exception *name* in the calling process, with additional arguments, if any, given by *argument*(s). The optional *status* receives the completion status.
KER$RAISE_PROCESS_EXCEPTION([status.wlc.r], process.rlu.v)	Raise the asynchronous exception KER$_PROCESS_ATTENTION in the specified *process*. The optional *status* receives the completion status.
KER$RECEIVE([status.wlc.r], message_variable.wlu.r, pointer_variable.wa.r, message_size.wlu.r, source_port.rou.r, [destination_port.wou.r], [reply_port.wou.r])	Receive message from *source_port*, return its MESSAGE value in *message_variable* and a pointer to its data part in *pointer_variable*. The variable *message_size* receives the data area's size in bytes, and the optional *status* receives the completion status. The variables *destination_port* and *reply_port* optionally receive the destination port specified by the sender and a port for replies, respectively.
KER$RESUME([status.wlc.r], process.rlu.v)	Resumes a previously suspended *process*. The optional *status* receives the completion status.
KER$SEND([status.wlc.r], message.rlu.v, size.rlu.v, destination_port.rou.r, [reply_port.rou.r], expedite.rlu.v)	Send *message* to *destination_port*, specifying the data area's *size* in bytes (if specified as −1, the size of the message as created is used), and optionally specifying a *reply_port*, and whether (TRUE or FALSE) to *expedite* the message. The optional *status* receives the completion status.

Table B-1. Continued

Call Format	Meaning
KER$SET_JOB_PRIORITY([status.wlc.r], priority.rlu.v)	Set priority of current job to *priority* (0–31, 0 highest). The optional *status* receives the completion status.
KER$SET_PROCESS_PRIORITY([status.wlc.r], process.rlu.v, priority.rlu.v)	Set priority of *process* to *priority* (0–15, 0 highest). The optional *status* receives the completion status.
KER$SET_PROTECTION([status.wlc.r], size.rlu.v, base_address.ra.v, code.rl.v)	Set protection of *size* bytes of memory at virtual *base_address* to *code* (0 for read-only access, 1 for read/write access, 2 for no access). The optional *status* receives the completion status.
KER$SET_TIME([status.wlc.r], absolute_time.rq.r)	Set current time to *absolute_time* (a nonnegative LARGE_INTEGER). The optional *status* receives the completion status.
KER$SET_USER([status.wlc.r], username.rvt.r, uic.rlu.v)	Set the user identity of the current process, specifying the *username* and *uic*. The optional *status* receives the completion status.
KER$SIGNAL([status.wlc.r], value.rlu.v)	Signal *value* of type AREA, EVENT, SEMAPHORE, or PROCESS. The optional *status* receives the completion status.

Table B-1. Continued

Call Format	Meaning
KER$SIGNAL_DEVICE([status.wlc.r], dev_number.rl.v)	Signal DEVICE object from interrupt service routine; *dev_number* supplies an integer in the range 0–15 identifying the device or element in a device array to be signaled. The optional *status* receives the completion status.
KER$SUSPEND([status.wlc.r], process.rlu.v)	Suspend the execution of of *process*. The optional *status* receives the completion status.
KER$TRANSLATE_NAME([status.wlc.r], port_variable.wou.r, string.rt.ds, scope.rl.v)	Translate *string*, searching in *scope* NAME$LOCAL, NAME$UNIVERSAL, or NAME$BOTH, and return the associated PORT value in *port_variable*. The optional *status* receives the completion status.
KER$UNWIND([status.wlc.r], new_fp.ra.v, [new_pc.ra.v])	Unwind call stack to new location. The target frame pointer is supplied by *new_fp* and the new program counter is optionally supplied by *new_pc*. The optional *status* receives the completion status.
KER$WAIT_ALL([status.wlc.r], [wait_result.wl.r], [time_value.rq.r], [object_list.rlu.v...])	Make calling process wait for all AREA, DEVICE, EVENT, PORT, PROCESS, or SEMAPHORE objects in *object_list* to satisfy the wait. Zero to four object values can be specified; the optional *wait_result* receives a nonzero value if the objects satisfied the wait or 0 if the procedure timed out. The optional *time_value* specifies a time interval or absolute time defining the timeout; the timeout is irrelevant, and the wait result is nonzero, if the necessary conditions were satisfied before the call. The optional *status* receives the completion status. Note: PORT values are octawords passed by reference (port_value.rou.r).

Table B-1. Continued

Call Format	Meaning
KER$WAIT_ANY([status.wlc.r], [wait_result.wl.r], [time_value.rq.r], [object_list.rlu.v...])	Make calling process wait for any object in *object_list* to satisfy the wait. If one or more object value is specified, the optional *wait_result* receives the argument number of the object that satisfied the wait or 0 if the procedure timed out. The optional *time_value* specifies a time interval or absolute time defining the timeout; the timeout is irrelevant, and the wait result is nonzero, if the necessary conditions were satisfied before the call. The optional *status* receives the completion status. Note: PORT values are octawords passed by reference (port_value.rou.r).

Appendix C
Status Values/Exception Names

All kernel procedures have an optional status parameter that returns the procedure's completion status. If you do not request the completion status (that is, if you omit the status parameter from a kernel procedure call), an exception is raised if the completion is unsuccessful. The 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 request the status).

The idea is that you can decide *not* to check the status after every call and can, instead, take an exception in the event of an error. The exception can then be handled by exception handlers, as explained in Chapter 11, "Exception Handling."

Note: To be used in exception handlers, the SS$ exception names must be declared in your program with the EXTERNAL and VALUE attributes.

Table C-1 lists the status values/exception names that are raised in VAXELN.

Table C-1. Status Values/Exception Names

Name	Source of Exception	Description
KER$_BAD_COUNT	Kernel procedures	The procedure call specified an incorrect number of arguments.
KER$_BAD_LENGTH	CREATE_JOB, CREATE_NAME, GET_USER	A string argument was too long.
KER$_BAD_MESSAGE_SIZE	SEND	The message data is too large to be sent to the destination port.
KER$_BAD_MODE	ALLOCATE procedures, FREE procedures, CREATE_DEVICE	A procedure that requires that the caller be executing in kernel mode was called from user mode.
KER$_BAD_STACK	RAISE_EXCEPTION, GOTO, hardware	The stack size was insufficient during a RAISE_EXCEPTION call or hardware exception, or the destination stack frame of a nonlocal GOTO could not be found.
KER$_BAD_STATE	Circuit procedures, DELETE, RESUME, SET_USER	An object specified in a kernel procedure is in an invalid state for the attempted operation: a port specified to CONNECT_CIRCUIT or ACCEPT_CIRCUIT contains unreceived messages or has an incomplete CONNECT_CIRCUIT or ACCEPT_CIRCUIT pending; a port specified to DISCONNECT_CIRCUIT was not connected; a device specified to DELETE has an interrupt pending; or a process specified to RESUME is not suspended.
KER$_BAD_TYPE	Kernel procedures	An argument to the procedure has the wrong data type.
KER$_BAD_VALUE	Kernel procedures	An argument to the procedure is out of range or otherwise invalid.
KER$_CONNECT_PENDING	ACCEPT_CIRCUIT, SEND, RECEIVE	A CONNECT_CIRCUIT is pending, and the port cannot be used for another purpose until the connection has completed.

Table C-1. Continued

Name	Source of Exception	Description
KER$_CONNECT_TIMEOUT	CONNECT_CIRCUIT	A CONNECT_CIRCUIT request was not accepted by the destination in the timeout limit. (The connect circuit timeout can be set with the System Builder.)
KER$_COUNT_OVERFLOW	DISABLE_SWITCH, SEND, SIGNAL	The SIGNAL procedure was called for a semaphore already at its maximum value, or the SEND procedure was called for a port containing its limit of unreceived messages.
KER$_COUNT_UNDERFLOW	ENABLE_SWITCH	The ENABLE_SWITCH procedure was called more times than the DISABLE_SWITCH procedure was called.
KER$_DEVICE_CONNECTED	CREATE_DEVICE	The device named in a CREATE_DEVICE procedure call is already connected to a DEVICE data item.
KER$_DISCONNECT	RECEIVE, SEND	The circuit was disconnected by the partner process.
KER$_DUPLICATE	CREATE_NAME	The CREATE_NAME procedure was called with a name string that duplicates an existing name.
KER$_EXPEDITED	RECEIVE	The procedure completed successfully, and the received message is an expedited message.
KER$_KERNEL_STACK	Software exception	The kernel stack is insufficient in size or the kernel stack pointer is invalid.
KER$_MACHINE_CHECK	Hardware exception	The processor detected a hardware failure.
KER$_NO_ACCESS	Kernel procedures	An argument specified in a kernel procedure call is not accessible by the calling program; for example, an output argument is a variable with the READONLY attribute.
KER$_NO_DESTINATION	CONNECT_CIRCUIT	Neither a destination PORT value nor a destination NAME value was specified in a CONNECT_CIRCUIT procedure call.
KER$_NO_INITIALIZATION	INITIALIZATION_DONE	No job initialization was specified when the program was added to the system by the System Builder.
KER$_NO_MAP_REGISTER	ALLOCATE_MAP	No free UNIBUS or QBUS map registers are currently available. There are 496 map registers per UNIBUS or QBUS.

Table C-1. Continued

Name	Source of Exception	Description
KER$_NO_MEMORY	ALLOCATE_MEMORY, CREATE_AREA, CREATE_JOB, CREATE_MESSAGE, CREATE_PROCESS	No free pages of physical memory are currently available.
KER$_NO_MESSAGE	RECEIVE	No unreceived messages are currently in the port.
KER$_NO_OBJECT	All CREATE procedures, RECEIVE	No job object table entries are currently available. There are a maximum of 1024 entries per job; that is, only 1024 data itmes of the system types can exist at once in a job.
KER$_NO_PAGE_TABLE	CREATE_JOB, CREATE_PROCESS	No free process page table is currently available. The number of process page tables can be set with the System Builder.
KER$_NO_PATH_REGISTER	ALLOCATE_PATH	No free UNIBUS adapter datapath register is currently available. There are three buffered datapaths per VAX–11/750 UNIBUS adapter.
KER$_NO_POOL	All CREATE procedures	No system dynamic memory is currently available. The size of the system dynamic memory pool can be set with the System Builder.
KER$_NO_PORT	CREATE_PORT	No system port table entries are currently available. The size of the system port table can be set with the System Builder.
KER$_NO_STATUS	CREATE_PROCESS	The process was deleted and so no exit status value is available to return.
KER$_NO_SUCH_DEVICE	CREATE_DEVICE	The device name specified in a CREATE_DEVICE call cannot be found in the list of devices created with the System Builder.
KER$_NO_SUCH_NAME	CONNECT_CIRCUIT, TRANSLATE_NAME	The translation for a name cannot be found.

Table C-1. Continued

Name	Source of Exception	Description
KER$_NO_SUCH_PORT	Circuit procedures SEND, RECEIVE, SET_USER	No port with the specified value can be found in the system or network, or the port is not owned by the current job as required by the procedure.
KER$_NO_SUCH_PROGRAM	CREATE_JOB	No program with the specified name can be found in the program list created with the System Builder.
KER$_NO_SYSTEM_PAGE	CREATE_DEVICE	No free system page table entries are currently available to map the I/O region.
KER$_NO_VIRTUAL	ALLOCATE_MEMORY, CREATE_AREA, RECEIVE	No free virtual address space is currently available for the process. The size of process virtual address space can be set with the System Builder.
KER$_POWER_SIGNAL	Asynchronous hardware exception	System power recovery is in progress.
KER$_PROCESS_ATTENTION	Asynchronous software exception	The procedure KER$RAISE_PROCESS_EXCEPTION was called.
KER$_QUIT_SIGNAL	Asynchronous software exception	Another process in the job has signaled the current process with SIGNAL.
KER$_SUCCESS	Kernel procedures	The procedure completed successfully.

Table C-1. Continued

Name	Source of Exception	Description
SS$_ACCVIO	Run-time operation	Access violation
SS$_BREAK	Run-time operation	Breakpoint fault
SS$_CMODUSER	Run-time operation	Change mode to user trap
SS$_COMPAT	Run-time operation	Compatibility mode fault
SS$_DECOVF	Run-time operation	Arithmetic trap, decimal overflow
SS$_INTOVF	Run-time operation	Arithmetic trap, integer overflow
SS$_INTDIV	Run-time operation	Arithmetic trap, integer divide by zero
SS$_FLTDIV	Run-time operation	Arithmetic trap, floating/decimal divide by zero
SS$_FLTDIV_F	Run-time operation	Arithmetic fault, floating divide by zero
SS$_FLTOVF	Run-time operation	Arithmetic trap, floating overflow
SS$_FLTOVF_F	Run-time operation	Arithmetic fault, floating overflow
SS$_FLTUND	Run-time operation	Arithmetic trap, floating underflow
SS$_FLTUND_F	Run-time operation	Arithmetic fault, floating underflow
SS$_OPCCUS	Run-time operation	Opcode reserved to customer fault
SS$_OPCDEC	Run-time operation	Opcode reserved to DIGITAL fault
SS$_RADRMOD	Run-time operation	Reserved addressing fault
SS$_ROPRAND	Run-time operation	Reserved operand fault
SS$_SUBRNG	Run-time operation	Arithmetic trap, subscript out of range
SS$_TBIT	Run-time operation	T-bit pending trap

Table C-1. Continued

Name	Source of Exception	Description
ELN$_ADAWI	Run-time range checking	The delta argument of ADD_WORD_INTERLOCKED was out of range.
ELN$_AMBENUMSTR	CONVERT function	The enumerated type is ambiguous in a conversion of a string to an enumerated type.
ELN$_ARGUMENT	Run-time range checking	A nonexistent argument was referred to with the ARGUMENT function.
ELN$_ARRAYBOUND	Run-time range checking	In an operation requiring array type equivalence, the bounds of two arrays were different.
ELN$_ASSERT	Run-time assertion check	The value of an ASSERT expression was FALSE.
ELN$_CASELAB	Run-time range checking	In a CASE statement, there was no case constant for the current value of the case selector.
ELN$_CHARASGN	Run-time range checking	A string with more than one character, or an empty string, was assigned to a CHAR variable.
ELN$_CHR	Run-time range checking	The argument of a CHR function was not in the range 0–255.
ELN$_EOFNOTDEF	Run-time range checking	The EOF function was called with a file for which EOF is currently undefined.
ELN$_EOLN	Run-time range checking	The EOLN function was called with a file for which EOF is currently TRUE.
ELN$_FINDFIRST	Run-time range checking	In a FIND_FIRST_BIT function call, the *start index* argument was out of range.
ELN$_GENERIC	Run-time range checking	A statement was executed in which more than one range violation was detected by the compiler.
ELN$_INTCONVERT	Run-time range checking	In the CONVERT function, an integer was out of the range for conversion to BOOLEAN or an enumerated type.
ELN$_INVDBLSTR	CONVERT function	A double-precision number is specified incorrectly in a conversion of a string to DOUBLE.

Table C-1. Continued

Name	Source of Exception	Description
ELN$_INVENUMSTR	CONVERT function	An invalid enumerated value is specified in a conversion of a string to an enumerated type.
ELN$_INVENUMVAL	CONVERT function	An invalid enumerated value is specified in a conversion of an enumerated type to a string.
ELN$_INVREALSTR	CONVERT function	A single-precision number is specified incorrectly in a conversion of a string to REAL.
ELN$_INVTIMSTR	TIME_VALUE function	In the TIME_VALUE function, the argument was not a valid time string.
ELN$_INVTIMVAL	TIME_STRING function	In the TIME_STRING function, the argument was not a valid time value.
ELN$_NEGSIZE	Run-time range checking	The size of flexible-type data was negative.
ELN$_NEGSTRLEN	Run-time conversion	The length of a string was negative.
ELN$_NOTENUMSTR	CONVERT function	In a conversion of a string to an enumerated type, the string does not name a value of the type.
ELN$_PAOC	Run-time range checking	In a PACKED ARRAY[1..*n*] OF CHAR (used as a string), *n* was greater than 32,767.
ELN$_PAOCASGN	Run-time range checking	A string with the wrong number of characters was assigned to a packed array of CHAR.
ELN$_PRED	Run-time range checking	An argument of the PRED function had too small an ordinal value.
ELN$_PROBESIZE	PROBE_READ function, PROBE_WRITE function	The size of the variable being probed is > 65,535 bytes.
ELN$_RECEIVE	Run-time range checking	The size of the data in a received message (RECEIVE procedure) does not match the size of the type associated with the procedure's data pointer argument.
ELN$_SETASGN	Run-time range checking	In an assignment, a source set had members outside the range of the target set.

Table C-1. Continued

Name	Source of Exception	Description
ELN$_SETCONSTR	Run-time range checking	In a set constructor, the left-hand expression in a member designator (expres1..expres2), or a single expression, was not in the range 0–32,766; or, the right-hand expression in a member designator was greater than 32,766.
ELN$_STRLEN	Run-time range checking	The length of a string was greater than 32,767.
ELN$_SUBRASGN	Run-time range checking	In an assignment, the source expression's result was outside the range of the target subrange type.
ELN$_SUBSCR	Run-time range checking	An array subscript was out of range.
ELN$_SUBSTR	Run-time range checking	An argument of the SUBSTR function was out of range.
ELN$_SUCC	Run-time range checking	An argument of the SUCC function had too large an ordinal value.
ELN$_TRANSLATE	Run-time range checking	In the TRANSLATE_STRING function, there was no translation character for a character in the source string.
ELN$_TYPECAST	Run-time range checking	In a typecast variable, the target type was larger than the variable's actual type.
ELN$_TYPEEXTENT	Run-time range checking	In an operation requiring type equivalence, the extents of two types were different.
ELN$_ZEROSIZE	Run-time range checking	The target of the ZERO function exceeded 65,535 bytes.

Table C-1. Continued

Name	Source of Exception	Description	
C$_E2BIG	VAXELN C RTL	Argument list too long	See **Note** below
C$_EACCES	VAXELN C RTL	Permission denied	
C$_EAGAIN	VAXELN C RTL	No more processes	
C$_EBADF	VAXELN C RTL	Bad file number	
C$_EBUSY	VAXELN C RTL	Mount device busy	
C$_ECHILD	VAXELN C RTL	No children	
C$_EDOM	VAXELN C RTL	Math argument error	
C$_EEXIST	VAXELN C RTL	File exists	
C$_EFAULT	VAXELN C RTL	Bad address	
C$_EFBIG	VAXELN C RTL	File too large	
C$_EINVAL	VAXELN C RTL	Invalid argument	
C$_EINTR	VAXELN C RTL	Interrupted system call	
C$_EIO	VAXELN C RTL	I/O error	
C$_EISDIR	VAXELN C RTL	Is a directory	
C$_EMFILE	VAXELN C RTL	Too many open files	
C$_EMLINK	VAXELN C RTL	Too many links	
C$_ENFILE	VAXELN C RTL	File table overflow	
C$_ENOSPC	VAXELN C RTL	No space left on device	
C$_ENODEV	VAXELN C RTL	No such device	
C$_ENOTBLK	VAXELN C RTL	Block device required	
C$_ENOMEM	VAXELN C RTL	Not enough core	
C$_ENOTDIR	VAXELN C RTL	Not a directory	
C$_ENOTTY	VAXELN C RTL	Not a typewriter	
C$_ENOEXEC	VAXELN C RTL	Exec format error	
C$_ENOENT	VAXELN C RTL	No such file or directory	

Table C-1. Continued

Name	Source of Exception	Description
C$_ENXIO	VAXELN C RTL	No such device or address
C$_EPERM	VAXELN C RTL	Not owner
C$_EPIPE	VAXELN C RTL	Broken pipe
C$_ERANGE	VAXELN C RTL	Result too large
C$_EROFS	VAXELN C RTL	Read-only file system
C$_ESPIPE	VAXELN C RTL	Illegal seek
C$_ESRCH	VAXELN C RTL	No such process
C$_ETXTBSY	VAXELN C RTL	Text file busy
C$_EXDEV	VAXELN C RTL	Cross-device link

Note: All of the C RTL status codes on this page and the previous page are generated by the VAXELN C run-time library as the result of run-time range checking or explicit signalling by C user code. The names and descriptions are derived from the corresponding UNIX* error codes. These status values are usually confined inside the execution stream of a C program, but may be received by a user's condition handler if that handler is declared after the C "last-chance" condition handler is established. It is established before the user's "main" function is invoked.

*UNIX is a trademark of AT&T Bell Laboratories

Appendix D
VAXELN Performance Evaluation

This appendix lists the performance numbers for a VAXELN MicroVAX I system, measuring three common programming tasks:

- Process synchronization and management
- Message passing
- File input/output speed

These tasks are measured on an unloaded MicroVAX I system, with an RD52 disk drive and 1 megabyte of memory.

Process Synchronization and Management

Process synchronization and management is divided into four categories:

- The extra time it takes to synchronize processes
- The time it takes to create and begin execution of a subprocess
- The time it takes to delete and return execution from a subprocess back to its parent
- The time it takes to respond and return from an interrupt

Each category is considered in turn.

Synchronization

Process synchronization is handled with either a MUTEX variable using the LOCK_MUTEX and UNLOCK_MUTEX procedures, or a SEMAPHORE variable using the SIGNAL and WAIT_ANY or WAIT_ALL procedures. The extra overhead that it takes to synchronize processes depends on the type of variable used and whether there is any actual contention involved (that is, a context switch is needed).

Table D-1 summarizes the times needed to manipulate a mutex or semaphore with and without contention. The times shown are in microseconds (µs).

Table D-1. Process Synchronization Times

	Without Contention	With Contention (e.g., context switch)
Mutex Lock and Unlock	19 µs	1115 µs
Semaphore Wait and Signal	641 µs	1100 µs

Create and Delete

Table D-2 summarizes the amount of time it takes to create a subprocess and begin its execution (that is, the amount of time it takes from a CREATE_PROCESS call to executing the first instruction of the new subprocess).

This table also summarizes the amount of time it takes to exit from a subprocess and return control to its parent (that is, the amount of time it takes from a subprocess calling EXIT to executing the first instruction of a process that did a WAIT_ANY or WAIT_ALL on that subprocess).

The times shown in Table D-2 are in microseconds.

Table D-2. Process Create, Execute, and Delete Times

	Time
Create and begin execution of subprocess	2445 μs
Return and delete subprocess	2853 μs

Interrupt Processing

With device drivers, it is important to know how long it takes the system to respond to an interrupt (that is, to begin executing the first instruction of an interrupt service routine after an interrupt), and how long it takes to resume the execution of a process waiting on a device (that is, by an interrupt service routine using SIGNAL_DEVICE). Table D-3 summarizes these times in microseconds.

Table D-3. Interrupt Processing Times

	Time
Begin executing interrupt service routine	66 μs
Resume process that waited on a device	586 μs

Messages

Table D-4 summarizes the amount of time it takes to create, deliver, and delete messages via circuits. Times are shown in microseconds and throughput is expressed in kilobits per second (Kbit/sec).

Two configurations are tested:

- A single machine running two jobs that communicate via a circuit
- Two machines running the same two jobs, but communicating via a circuit over an Ethernet

In Table D-4, the single-machine delivery times are the actual times needed to create, send, and receive messages on a single machine with the appropriate number of context switches. The two-machine delivery times are the actual times needed to send a message between two machines over an Ethernet. With two machines, the creation and deletion of messages occurs simultaneously.

Table D-4. Message Times and Throughput

Size of Message (Bytes)	Create Time	Delete Time	Deliver on One Machine: Time	Deliver on One Machine: Throughput (Kbit/sec)	Deliver on Two Machines: Time	Deliver on Two Machines: Throughput (Kbit/sec)
0	370 μs	352 μs	2631 μs	N/A	11553 μs	N/A
512	599 μs	549 μs	3284 μs	1250	14239 μs	288
2048	778 μs	579 μs	3750 μs	4370	41396 μs	396
8192	877 μs	732 μs	4229 μs	15500	157314 μs	417

File Input/Output

The major operations for file I/O are: open, close, read, and write. The time required for each of these operations is dependent upon the file's access mode; that is, sequential versus direct access.

This section presents the times and throughput of each operation for sequential files first, followed by the times and throughput of each operation for direct access files. All of the times are taken from a MicroVAX I with an RD52 disk drive set to a cluster_size of 3 blocks.

Sequential Files

The times and throughput of the open, write, read, and close operations for sequential files are summarized in the following subsections.

Open

Opening a sequential file can take from 0.3 seconds to 0.5 seconds, depending on its file_history and filesize specifications.

The types of opens tested are:

- Opening a new file without any other specification, other than the filename
- Opening a new file, but specifying the filesize
- Opening an old file that is already the correct filesize for append and using REWRITE
- Opening an old file for read-only

Table D-5 summarizes the times required to open a file under the preceding schemes. The times shown are in milliseconds (ms).

Table D-5. Open Times for Sequential Files

	Time
Open a new file	480 ms
Open a new file, specifying the filesize	500 ms
Open an old file, with append	290 ms
Open an old file, read-only	290 ms

Write

The throughput that is achieved when writing to a sequential file is dependent upon how it was opened and the size of the records written. The following program fragment is part of the VAXELN Pascal program used to time the file system:

```
program write-sequential-file;

type rec = byte-data(1536);
var largefile : file of rec;
   i, num : integer;

begin
  ...
  num : = ... { number of records to write }
  ...
  open(largefile, file-name : = 'largefile.dat' );
  rewrite(largefile);
  for i : = 1 to num do put(largefile);
  close(largefile);
  ...
  open(largefile, file-name : = 'largefile.dat',
     filesize : = num * 3, contiguous : = true);
  rewrite(largefile);
  for i : = 1 to num do put(largefile);
  close(largefile);
  ...
  open(largefile, file_name : = 'largefile.dat',
     history : = HISTORY$OLD, append : = true);
  rewrite(largefile);
  for i : = 1 to num do put(largefile);
  close(largefile);
  ...
 end;
```

This program uses a record size of 1536 bytes and was run for files larger than 500 disk blocks. Analogous programs were also run using different record sizes.

Table D-6 summarizes the write throughput for various record sizes and openings. Throughput is expressed in kilobytes per second (Kbyte/sec).

Table D-6. Write Throughput for Sequential Files

Size of Record (Bytes)	New (Kbyte/sec)	New with Filesize (Kbyte/sec)	Old with Append (Kbyte/sec)
1536	53.0	58.1	59.8
512	38.0	40.4	43.2
128	25.6	26.6	26.9
32	8.6	8.6	8.7

Read

The throughput that is achieved in reading a sequential file is dependent upon the record size. The following program fragment is part of the VAXELN Pascal program used to time the file system:

```
program read–sequential–file;

type rec = byte–data(1536);
var largefile : file of rec;
   i, num : integer;

begin
 ...
 num : = ... { number of records to read }
 ...
 open(largefile, file–name : = 'largefile.dat',
    history : = HISTORY$OLD);
 reset(largefile);
 for i : = 1 to num do get(largefile);
 close(largefile);
 ...
end;
```

This program uses a record size of 1536 bytes and was run for files larger than 500 disk blocks. Analogous programs were also run using different record sizes.

Table D-7 summarizes the read throughput for various record sizes. The throughput is expressed in kilobytes per second.

Table D-7. Read Throughput for Sequential Files

Size of Record (Bytes)	Throughput (Kbyte/sec)
1536	59.0
512	47.5
128	25.7
32	7.9

Close

The amount of time it takes to close a sequential file is dependent upon how the file is opened. Table D-8 summarizes the time it takes to close a file, using a record size of 1536 bytes. The times shown are in milliseconds.

Table D-8. Close Times for Sequential Files

	Time
Close a new file	395 ms
Close a new file (opened with filesize)	361 ms
Close an old file (opened with append)	361 ms
Close an old file (opened read-only)	100 ms

Direct Access Files

The times and throughput of the open, write, read, and close operations for direct access files are summarized in the following subsections.

Open

The types of opens for direct access files tested are:

- Opening a new file with the number of disk blocks specified
- Opening an old file for read-only

Table D-9 summarizes the open times for both schemes. The times shown are in milliseconds.

Table D-9. Open Times for Direct Access Files

	Time
Open a new file, specifying the filesize	500 ms
Open an old file, read-only	290 ms

Write

The throughput for writing direct access files is dependent upon the record size chosen. The following program fragment is part of the VAXELN Pascal program used to time the file system:

```
program write_direct_file;

type rec = byte_data(1536);
var largefile : file of rec;
   i, num : integer;

begin
 ...
 num : = ... { number of records to write }
 ...
 open(largefile, file_name : = 'largefile.dat',
    filesize : = num * 3, contiguous : = true,
    access_method : = access$direct);
 for i : = 1 to num do
  begin locate(largefile, i); put(largefile) end;
 close(largefile);
 ...
end;
```

This program uses a record size of 1536 bytes and was run for files larger than 500 disk blocks. Analogous programs were also run using different record sizes.

Table D-10 summarizes the write throughput for direct access files. The throughput is expressed in kilobytes per second.

Table D-10. Write Throughput for Direct Access Files

Size of Record (Bytes)	Throughput (Kbyte/sec)
1536	36.0
512	25.9
128	11.8
32	3.3

Read

The throughput for reading direct access files is dependent upon the record size chosen. The following program fragment is part of the VAXELN Pascal program used to time the file system:

```
program read_direct_file;

type rec = byte_data(1536);
var largefile : file of rec;
   i, num : integer;

begin
 ...
 num : = ... { number of records to read }
 ...
 open(largefile, file_name : = 'largefile.dat',
    history : = HISTORY$OLD,
    access_method : = ACCESS$DIRECT);
 for i : = 1 to num do begin find(largefile, i); end;
 close(largefile);
```

```
...
end;
```

This program uses a record size of 1536 bytes and was run for files larger than 500 disk blocks. Analogous programs were also run using different record sizes.

Table D-11 summarizes the read throughput for direct access files. The throughput is expressed in kilobytes per second.

Table D-11. Read Throughput for Direct Access Files

Size of Record (Bytes)	Throughput (Kbyte/sec)
1536	58.6
512	25.4
128	11.9
32	3.5

Close

The amount of time it takes to close a direct access file is 100 milliseconds, regardless of how the file is opened.

Index

VAXELN User's Guide
AA-EU38A-TE

READER'S COMMENTS

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.) ______________________________

What features are most useful? ______________________________

Does the publication satisfy your needs? ______________________________

What errors have you found? ______________________________

Additional comments ______________________________

Name ______________________________

Title ______________________________

Company ____________________ Dept. ____________

Address ______________________________

City ____________ State ____________ Zip ____________

(staple here)

-------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

Attention Publications Manager
Digital Equipment Corporation
2265 116 Avenue Northeast
Bellevue,
Washington, 98004

-------Do Not Tear - Fold Here and Tape --

(staple here)