6.7.1 Invoking the Kernel Debugger at Boot Time 6-23
6.7.2 Invoking the Kernel Debugger on a Running System 6-24

6.8 ADA TASKING AND REMOTE DEBUGGING 6-24
6.8.1 Stack Storage 6-25
6.8.2 Monitoring Tasking Performance 6-25

6.9 SAMPLE REMOTE DEBUGGING SESSION 6-27

PART III RUN-TIME RELATED TOPICS

CHAPTER 7 INPUT-OUTPUT 7-1

7.1 FILES AND FILE ACCESS 7-2
7.1.1 Disk Files 7-2
7.1.2 Tape Files 7-3
7.1.3 Terminals and Printers 7-4
7.1.4 Network Files 7-4
7.1.5 VAXELN Circuits 7-5
7.1.6 Other Devices 7-6

7.2 NAMING EXTERNAL FILES 7-6
7.2.1 Node Names 7-6
7.2.2 Device Names 7-7
7.2.3 Directory Name 7-8
7.2.4 File Type 7-8

7.3 STANDARD INPUT-OUTPUT FILES 7-8

7.4 SPECIFYING EXTERNAL FILE ATTRIBUTES WITH THE FILE DEFINITION LANGUAGE (FDL) 7-9
7.4.1 FDL Primary and Secondary Attributes 7-10
7.4.2 VAXELN_CIRCUIT Attribute 7-12
7.4.3 Commonly Used FDL Attributes 7-13

CHAPTER 8 TASKING 8-1

8.1 INTRODUCTION TO USING TASKS ON VAXELN 8-1

8.2 TASK STORAGE ALLOCATION 8-3
8.2.1 Storage Created for a Task Object—the Task Control Block 8-4
8.2.2 Storage Created for a Task Activation—the Task Stack 8-5
8.2.2.1 Controlling the Size of a Main Task Stack • 8-6
8.2.2.2 Controlling the Size of Other Task Stacks • 8-7
8.2.3 Stack Overflow and Non-Ada Code 8-8

8.3 TASK SWITCHING 8-8

8.4 SPECIAL TASKING CONSIDERATIONS 8-10

8.5 CALLING VAXELN KERNEL SERVICES FROM TASKS 8-10

8.6 MEASURING AND TUNING TASKING PERFORMANCE 8-10

CHAPTER 9 CALLING VAXELN SERVICES 9-1

9.1 OBJECTS IN VAXELN SERVICES 9-1
9.1.1 AREA Object 9-2
9.1.2 DEVICE Object 9-3
9.1.3 EVENT Object 9-3
9.1.4 MESSAGE Object 9-4
9.1.5 MUTEX Object 9-5
9.1.6 NAME Object 9-6
9.1.7 PORT Object 9-6
9.1.8 PROCESS Object 9-7
9.1.9 SEMAPHORE Object 9-8

9.2 VAXELN TYPES 9-9

9.3 USING STRINGS WITH VAXELN SERVICES 9-10

9.3.1 Fixed-Length Strings 9-10

9.3.2 Varying-Length Strings 9-10

9.3.2.1 Uses of Varying Strings • 9-11

9.3.2.2 Varying String Descriptors • 9-12

9.4 VAXELN SERVICES 9-12

9.4.1 Authorization Utility Procedures 9-12

9.4.2 Authorization Service Utility Procedures 9-13

9.4.3 Device Driver Utility Procedures 9-13

9.4.3.1 Analog-to-Digital Converter Procedures • 9-14

9.4.3.2 Real-Time Clock Procedures • 9-14

9.4.3.3 Asynchronous Serial Line Controller Procedures • 9-14

9.4.3.4 Parallel Line Interface Procedures • 9-15

9.4.4 DMA Device Handling Procedures 9-15

9.4.5 Exception Handling Procedures 9-16

9.4.6 Exit Utility Procedures 9-17

9.4.7 File Service Procedures 9-17

9.4.7.1 Disk Utility Procedures • 9-17

9.4.7.2 File Utility Procedures • 9-17

9.4.7.3 Tape Utility Procedures • 9-18

9.4.8 Interrupt Priority Level Procedures 9-18

9.4.9 Memory Allocation Procedures 9-19

9.4.10 Message Transmission Procedures 9-19

9.4.11 Program Argument Procedures 9-20

9.4.12 Program Loader Procedures 9-20

9.4.13 Time-Representation Procedure 9-21

9.4.14 Virtual-to-Physical Address Procedure 9-21

9.4.15 Other Services 9-21

9.5 VAX/VMS SERVICES AVAILABLE WITH VAXELN ADA 9-22

APPENDIX A PACKAGE VAXELN_SERVICES A-1

A.1 TYPE DEFINITIONS A-1

A.2 VAXELN SERVICE PROCEDURE DESCRIPTIONS A-21

ACCEPT_CIRCUIT A-22
ALLOCATE_MAP A-24
ALLOCATE_MEMORY A-26
ALLOCATE_PATH A-28
AUTH_ADD_USER A-30
AUTH_MODIFY_USER A-33
AUTH_REMOVE_USER A-36
AUTH_SHOW_USER A-38
AXV_INITIALIZE A-40
AXV_READ A-42
AXV_WRITE A-45
CANCEL_EXIT_HANDLER A-46
CLEAR_EVENT A-47
CONNECT_CIRCUIT A-48
COPY_FILE A-51
CREATE_AREA A-53
CREATE_DEVICE A-55
CREATE_DIRECTORY A-59
CREATE_EVENT A-61
CREATE_JOB A-62
CREATE_MESSAGE A-64
CREATE_MUTEX A-66
CREATE_NAME A-67
CREATE_PORT A-69
CREATE_PROCESS A-71
CREATE_SEMAPHORE A-73
CURRENT_PROCESS A-75

DECLARE_EXIT_HANDLER	A-76
DELETE_AREA	A-77
DELETE_DEVICE	A-78
DELETE_EVENT	A-79
DELETE_FILE	A-80
DELETE_MESSAGE	A-81
DELETE_MUTEX	A-82
DELETE_NAME	A-83
DELETE_PORT	A-84
DELETE_PROCESS	A-85
DELETE_SEMAPHORE	A-86
DIRECTORY_CLOSE	A-87
DIRECTORY_LIST	A-88
DIRECTORY_OPEN	A-90
DISABLE_ASYNCH_EXCEPTION	A-92
DISABLE_INTERRUPT	A-93
DISABLE_SWITCH	A-94
DISCONNECT_CIRCUIT	A-95
DISMOUNT_TAPE_VOLUME	A-96
DISMOUNT_VOLUME	A-97
DLV_INITIALIZE	A-98
DLV_READ_BLOCK	A-100
DLV_READ_STRING	A-101
DLV_WRITE_STRING	A-102
DRV_INITIALIZE	A-103
DRV_READ	A-105
DRV_WRITE	A-106
ENABLE_ASYNCH_EXCEPTION	A-107
ENABLE_INTERRUPT	A-108
ENABLE_SWITCH	A-109
ENTER_KERNEL_CONTEXT	A-111
EXIT_PROCESS	A-112
FREE_MAP	A-113

FREE_MEMORY	A-115
FREE_PATH	A-116
GET_TIME	A-118
GET_USER	A-119
INITIALIZATION_DONE	A-121
INIT_TAPE_VOLUME	A-122
INIT_VOLUME	A-124
JOB_PORT	A-129
KWV_INITIALIZE	A-130
KWV_READ	A-133
KWV_WRITE	A-135
LOAD_PROGRAM	A-137
LOAD_UNIBUS_MAP	A-140
LOCK_MUTEX	A-142
MEMORY_SIZE	A-143
MOUNT_TAPE_VOLUME	A-144
MOUNT_VOLUME	A-146
PHYSICAL_ADDRESS	A-148
PROGRAM_ARGUMENT	A-149
PROGRAM_ARGUMENT_COUNT	A-150
PROTECT_FILE	A-151
RAISE_PROCESS_EXCEPTION	A-153
RECEIVE	A-154
RENAME_FILE	A-157
RESUME	A-159
SEND	A-160
SET_JOB_PRIORITY	A-163
SET_PROCESS_PRIORITY	A-164
SET_PROTECTION	A-166
SET_TIME	A-168
SET_USER	A-169
SIGNAL_AREA	A-170
SIGNAL_DEVICE	A-171

SIGNAL_EVENT A-172
SIGNAL_PROCESS A-173
SIGNAL_SEMAPHORE A-174
SUSPEND A-176
TRANSLATE_NAME A-177
UNIBUS_MAP A-179
UNIBUS_UNMAP A-181
UNLOAD_PROGRAM A-182
UNLOCK_MUTEX A-183
WAIT_ALL OR WAIT_ANY A-184

APPENDIX B DEBUGGER COMMAND SUMMARY B-1

B.1 VAXELN REMOTE DEBUGGER COMMANDS B-1

B.2 DEBUGGER COMMAND SUMMARY B-5
B.2.1 Starting and Terminating a Debugging Session B-6
B.2.2 Controlling and Monitoring Program Execution B-6
B.2.3 Examining and Manipulating Data B-7
B.2.4 Controlling Type Selection and Symbolization B-7
B.2.5 Controlling Symbol Lookup B-8
B.2.6 Displaying Source Code B-8
B.2.7 Screen Mode B-9
B.2.8 Source Editing B-9
B.2.9 Defining Symbols B-10
B.2.10 Keypad Mode B-10
B.2.11 Command Procedures and Log Files B-10
B.2.12 Control Structures B-11
B.2.13 Debugging Special Cases B-11

APPENDIX C VAXELN-RELATED FEATURES ADDED TO VAX ADA C-1

C.1 MAIN TASK STORAGE ALLOCATION C-1

C.2 THE PACKAGE SYSTEM C-2

C.3 VAX ADA ADDITIONS TO PACKAGE SYSTEM C-2
C.3.1 Register Operations C-2
C.3.2 Interlocked Instructions C-4
C.3.3 Queue Instructions C-5

INDEX

EXAMPLES
6–1 Debugging a VAXELN Ada Program 6-28

FIGURES
1–1 VAXELN Ada Development Environment 1-3
2–1 Square-Root Server Application 2-7
4–1 Sample System Builder Menu 4-5
6–1 Main System Menu 6-7

TABLES
3–1 ACS Program Library Management Commands 3-13
3–2 Additional ACS Commands 3-16
3–3 Compilation Commands 3-19
3–4 ADA, COMPILE, and RECOMPILE Qualifiers 3-21
3–5 ACS LINK Command Qualifiers 3-24
4–1 EBUILD Command Qualifiers 4-2
4–2 Edit System Characteristics Menu 4-7
4–3 Edit Network Node Characteristics Menu 4-13

4–4 Add Program Description Menu 4-16

4–5 Add Device Description Menu 4-19

4–6 Common QBUS Device Names and Their SYSGEN Equivalents 4-21

4–7 Common UNIBUS Device Names and Their SYSGEN Equivalents 4-21

4–8 Add Terminal Description Menu 4-23

4–9 Edit Console Characteristics Menu 4-25

5–1 Datalink Device Default Addresses 5-4

7–1 FDL Primary and Secondary Attribute Descriptions 7-11

7–2 Commonly Used FDL Attributes 7-13

8–1 Comparison of VAX/VMS and VAXELN Ada Task Implementations 8-2

Preface

Intended Audience

This manual is intended for programmers who are building embedded, real-time, or turnkey applications in the VAX Ada language for use with the VAXELN executive.

Structure of This Document

The *VAXELN Ada User's Manual* has three parts, which are organized as follows:

Part I, Introduction to VAXELN Ada, contains the following chapter:

- Chapter 1 gives an overview of VAXELN Ada and its development environment. It also compares VAX Ada and VAXELN Ada features.

Part II, Program Development, contains the following chapters:

- Chapter 2 shows how to get started by introducing the compiler, the program library manager (ACS), the VAXELN System Builder Utility (EBUILD), and the VAXELN Remote Debugger. A sample VAXELN Ada application is also included.
- Chapter 3 gives information on how to compile and link VAXELN Ada programs. It includes information on Ada program libraries and sublibraries, and summarizes the syntax of the ACS commands for program library management, compilation, and linking.
- Chapter 4 describes how to build a VAXELN system with the VAXELN System Builder Utility. It gives the syntax of the EBUILD command and explains the use of System Builder menus.
- Chapter 5 explains the various methods for booting and running VAXELN systems on a target machine.

Part III, Run-Time Related Topics, contains the following chapters:

- Chapter 6 includes a description of the VAXELN Remote Debugger and the remote debugger commands. This debugger is used at a host system to debug, by means of an Ethernet connection, a program running on a target machine.
- Chapter 7 lists the VAXELN Ada input-output packages and provides detailed information about files and file access, as well as summary information about specifying external file attributes with the VAX/VMS File Definition Language (FDL).
- Chapter 8 discusses tasking issues relevant to the VAXELN environment.
- Chapter 9 discusses the package VAXELN_SERVICES and the use of strings described in this package. In addition, it lists VAXELN services according to function.

The following appendixes are also included:

- Appendix A describes the types, subtypes, and constants used in VAXELN Ada service calls. It also contains the procedure declarations for each VAXELN service and fully describes the arguments and possible status values for each procedure.
- Appendix B lists and describes the VAXELN Remote Debugger commands and summarizes VAX/VMS Debugger commands that can be used with the remote debugger.
- Appendix C summarizes the features relevant to the VAXELN system.

Associated Documents

A description of the VAX Ada implementation can be found in the VAX Ada documentation set, which consists of the following volumes:

VAX Ada Language Reference Manual
VAX Ada Programmer's Run-Time Reference Manual
Developing Ada Programs on VAX/VMS

VAXELN information can be found in the *VAXELN User's Guide*.

Information on installing VAXELN Ada is in the *VAXELN Ada Installation Guide*.

Conventions Used in This Document

This document uses the following conventions:

Convention	Meaning
RET	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, RET.
CTRL/X	The phrase CTRL/X indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/U.
$ SHOW TIME 07-JAN-1986 10:35:13	Interactive examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
out	Boldface indicates VAXELN Ada reserved words.
[expression]	Square brackets indicate that the enclosed item is optional.
[job_specifier,...]	Horizontal ellipsis means that the item may be repeated zero or more times.

Part I Introduction to VAXELN Ada

This section gives an overview of VAXELN Ada and discusses differences between VAXELN Ada and VAX Ada.

Chapter 1

Introduction to VAXELN Ada

This chapter provides an overview of VAXELN™ Ada® , its development environment, and its differences from VAX™ Ada® .

1.1 What Is VAXELN Ada?

VAXELN Ada is a VAX/VMS layered product that provides the capability for developing, debugging, and running Ada standalone applications on VAX processors using the VAXELN real-time executive.

VAXELN Ada components are

- VAXELN Ada run-time library, which supports all standard Ada features, including the packages TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO. It also supports all VAX Ada-specific features, except nonsequential files, ASTs, and timeslicing.
- Additional units for the VAX Ada library of predefined units.
- VAXELN debugging support, including the VAXELN Remote Debugger. Based on the VAX/VMS Debugger (DEBUG), the remote debugger provides the means for debugging VAXELN Ada applications running on a target VAX machine from a host VAX/VMS system, connected by Ethernet.

VAXELN Ada is layered on VAX Ada, the VAXELN Toolkit (VAXELN), and the VAX/VMS operating system. It can be used with development environment tools such as the VAX Language-Sensitive Editor and DEC/CMS.

™ VAX and VAXELN are trademarks of Digital Equipment Corporation.
® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

VAXELN Ada is based on the VAX Ada compiler and program library manager (ACS); the VAXELN Ada run-time components are VAXELN-specific versions of VAX Ada components.

1.2 The VAXELN Ada Program Development Environment

VAXELN Ada applications are developed on a host VAX/VMS system using the VAX Ada compiler, the VAX Ada program library manager (ACS), and the VAX/VMS Linker and ACS LINK command. Chapter 2 gives an example of a VAXELN Ada application.

Once the Ada application is developed, you build a VAXELN system with the VAXELN System Builder Utility (EBUILD). The System Builder combines the Ada program with the VAXELN kernel, VAXELN services, and the VAXELN Ada run-time library to produce a bootable system. You can load a system by means of a disk, tape cartridge, PROM, or Ethernet to a VAX processor. The processor serves as a dedicated machine and runs the VAXELN system as a standalone application.

You can use the VAXELN Remote Debugger to debug VAXELN Ada programs running on a target. The remote debugger is part of the VAX/VMS environment and is used from a terminal connected to the host system. The remote debugger communicates with the target machine by means of an Ethernet connection.

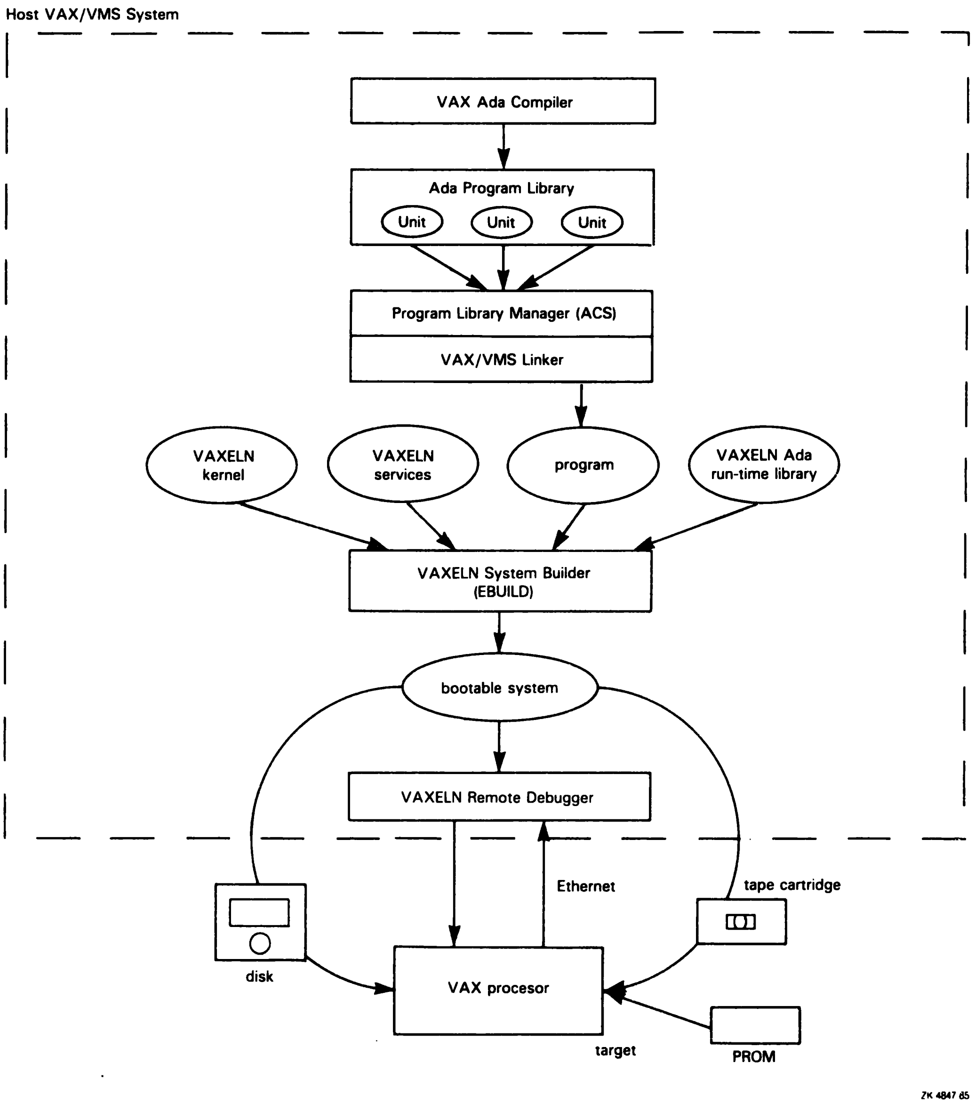
Figure 1–1 is an overview of the VAXELN Ada development environment and the steps involved in producing a VAXELN Ada application.

1.3 Differences from VAX Ada

VAXELN Ada supports all of the VAX Ada language and implementation-defined features except for the following:

- Pragma TIME_SLICE
- Pragma AST_ENTRY
- AST_ENTRY attribute
- Packages RELATIVE_IO, RELATIVE_MIXED_IO, INDEXED_IO, INDEXED_MIXED_IO

Figure 1-1: VAXELN Ada Development Environment



In addition, the implementation of the form string (used to specify external file characteristics in the input-output procedures CREATE and OPEN) differs slightly for VAXELN Ada.

VAXELN Ada also provides a number of implementation-dependent features designed to aid in the writing of Ada programs for the VAXELN target.

These differences are summarized in the following sections.

1.3.1 Time Slicing

The VAXELN system does not have time-sliced scheduling. Thus, the VAX Ada pragma TIME_SLICE, if specified in a compilation unit, is not supported for VAXELN Ada. The compiler will diagnose and ignore the pragma when the value of SYSTEM.SYSTEM_NAME is VAXELN. In other words, an Ada program unit containing pragma TIME_SLICE depends on a value of VAX_VMS for SYSTEM.SYSTEM_NAME, and cannot be linked into a VAXELN target image.

SYSTEM.SYSTEM_NAME and unit dependences are discussed in Chapter 3.

1.3.2 Asynchronous System Traps

The VAXELN system does not support the VAX/VMS asynchronous system trap (AST) mechanism. Therefore, neither the VAX Ada pragma AST_ENTRY nor the VAX Ada AST_ENTRY attribute is supported in VAXELN Ada. The compiler diagnoses and ignores the use of both the pragma and the attribute when the current value of SYSTEM.SYSTEM_NAME is VAXELN. In other words, an Ada program unit containing pragma AST_ENTRY or the AST_ENTRY attribute depends on a value of VAX_VMS for SYSTEM.SYSTEM_NAME, and cannot be linked into a VAXELN target image.

SYSTEM.SYSTEM_NAME and unit dependences are discussed in Chapter 3.

1.3.3 Implementation of the Form String

All of the Ada input-output packages provide CREATE and OPEN procedures. All of these procedures, in turn, have a FORM parameter that allows you to specify the characteristics of an external file from your Ada program. In VAX Ada, the value of the FORM parameter (called a form string) can be either a string of VAX Record Management Services (RMS) File Definition Language (FDL) statements, or a string referring to a file of FDL statements. In VAXELN Ada, the value can only be a string of FDL statements. Also, VAXELN Ada recognizes a different set of FDL statements (a subset of the statements recognized by VAX Ada, plus an additional statement that is useful only with VAXELN Ada).

See Chapter 6 for a full explanation.

1.3.4 Relative and Indexed Files

The VAXELN system does not provide relative and indexed files, so the VAX Ada packages RELATIVE_IO, INDEXED_IO, RELATIVE_MIXED_IO, and INDEXED_MIXED_IO are not supported by VAXELN Ada, and the bodies of these packages require that they be used only in programs that are executed on VAX/VMS targets. In other words, the relative and indexed packages depend on a value of VAX_VMS for the predefined constant SYSTEM.SYSTEM_NAME, and cannot be linked into a VAXELN target image.

SYSTEM.SYSTEM_NAME and unit dependences are discussed in Chapter 3.

1.3.5 VAXELN-Related Language Features

To allow you to specify fixed-size stack and stack storage areas for the task associated with the main program (main task), VAX Ada provides

Pragma MAIN_STORAGE

For distinguishing between target systems, VAX Ada provides

The enumeration literals VAXELN and VAX_VMS for type SYSTEM.SYSTEM_NAME

For communicating with device registers and internal processor registers, VAX Ada provides

Function SYSTEM.READ_REGISTER
Procedure SYSTEM.WRITE_REGISTER
Function SYSTEM.MFPR
Procedure SYSTEM.MTPR

For coding VAX interlocked instructions directly from an Ada program, VAX Ada provides

Type SYSTEM.ALIGNED_SHORT_INTEGER
Procedure SYSTEM.ADD_INTERLOCKED
Procedure SYSTEM.CLEAR_INTERLOCKED
Procedure SYSTEM.SET_INTERLOCKED

For coding VAX queue instructions directly from an Ada program, VAX Ada provides

Type SYSTEM.INSQ_STATUS
Type SYSTEM.REMQ_STATUS
Procedure SYSTEM.INSQHI
Procedure SYSTEM.INSQTI
Procedure SYSTEM.REMQHI
Procedure SYSTEM.REMQTI

The specifications for these features are documented in Appendix C.

VAXELN Ada also provides the package VAXELN_SERVICES, which defines interfaces to the VAXELN system services, utility routines, and device drivers. The package contains definitions of the special types needed by the VAXELN routines as well as definitions of the routines themselves. This package is described in Chapter 9 and specified in Appendix A.

Part II Program Development

This section presents information on program development using VAXELN Ada. It provides an overview of the steps in program development and gives detailed information on using the VAX Ada compiler and program library manager, VAXELN system building, loading finished VAXELN systems to target machines, and remote debugging.

Chapter 2

Getting Started

This chapter presents the basic steps you need to follow in building a VAXELN Ada application. It tells you how to use

- The VAX Ada compiler and program library manager (ACS) to compile and link an Ada program
- The VAXELN System Builder Utility (EBUILD) to produce a finished VAXELN application
- Downline loading capabilities to load the application to a target machine
- The VAXELN Remote Debugger to debug the application from the host system

The Ada source code for the sample application discussed here is included at the end of the chapter. VAXELN Ada program development is discussed more fully in Chapter 3 and Chapter 4. The remote debugger is discussed in Chapter 6.

2.1 Creating a VAXELN Ada Application

The following sections describe how to build a VAXELN Ada application.

2.1.1 Using the VAX Ada Compiler and Program Library Manager (ACS)

Programs are developed on the VAX/VMS host system, using the VAX Ada compiler and program library manager (ACS). ACS provides the user interface to the compiler and VAX/VMS Linker, which is used to link VAXELN Ada programs.

The following steps illustrate how to create and select an Ada library, and how to compile and link an Ada program. The sample application used here has three units. The source code, an explanatory diagram, and a full description are given in Section 2.2.

1. Create the Ada program library.

   ```
   $ ACS CREATE LIBRARY [.ADALIB]
   ```

2. Define the current program library.

   ```
   $ ACS SET LIBRARY [.ADALIB]
   ```

3. Compile the Ada programs.

   The following example shows how you would compile each unit separately. The /DEBUG qualifier is a default on the ADA command, but is shown here for clarity.

   The ERROR_HANDLING unit must be compiled first, because it must be defined in the current program library when you compile the other two units.

   ```
   $ ADA /DEBUG ERROR_HANDLING.ADA
   $ ADA /DEBUG SQRT_SERVER.ADA
   $ ADA /DEBUG SQRT_SERVER_TESTER.ADA
   ```

   The /DEBUG qualifier requests the compiler to write the symbol records associated with the units being compiled into the resulting object modules and allows you to run each unit under debugger control.

4. Link the Ada programs.

   After compiling the programs, you can link them using the /DEBUG qualifier and the /SYSTEM_NAME=VAXELN qualifier. The ELN$:RTL /INCLUDE=(KER$MSGDEF_TEXT) parameter causes the message text for messages generated by the VAXELN kernel to be included in the images.

```
$  ACS LINK/DEBUG/SYSTEM_NAME=VAXELN SQRT_SERVER -
_$  ELN$:RTL /INCLUDE=(KER$MSGDEF_TEXT)
$  ACS LINK/DEBUG/SYSTEM_NAME=VAXELN SQRT_SERVER_TESTER -
_$  ELN$:RTL /INCLUDE=(KER$MSGDEF_TEXT)
```

The /DEBUG qualifier on the ACS LINK command requests the linker to include all symbol information that is contained in the object modules in the executable image.

The /SYSTEM_NAME=VAXELN qualifier directs ACS to produce an image to be run under the VAXELN executive.

2.1.2 Using the VAXELN System Builder Utility

Before you can run the application, you must invoke the VAXELN System Builder Utility with the EBUILD command to combine the application programs with the VAXELN kernel to form a bootable system image.

The EBUILD command invokes the System Builder to combine one or more program images into a bootable system image. The following command creates a VAXELN system image, SQRT_SERVER.SYS, whose system characteristics are described in SQRT_SERVER.DAT.

```
$  EBUILD/NOEDIT SQRT_SERVER
```

The /NOEDIT qualifier in the previous example indicates that the system was built using a noninteractive mode. This causes the System Builder to build a system image immediately, from the current contents of a specified data file.

The interactive mode of system building, specified with the default /EDIT qualifier, allows you to select various system options or attributes from a series of menus.

See the *VAXELN User's Guide* and Chapter 4 of this book for more information on the VAXELN System Builder.

This discussion uses the data file SQRT_SERVER.DAT, constructed with the EBUILD command. This file describes the characteristics of this particular system and is the mechanism for giving the collected information to the System Builder. The data file contains:

```
characteristic /noconsole /nofile /name=ARTHUR /node_address= 42
/server /emulator=both /boot_method=downline
/debug=remote /network
program SQRT_SERVER        /run   /debug    /initialize
program SQRT_SERVER_TESTER /norun /nodebug
program SQRT_SERVER_TESTER /run   /debug
```

Note that there are two program descriptors for the program SQRT_SERVER_TESTER. By having multiple program descriptors, you can create jobs running the same program, but with different attributes. The System Builder names these jobs SQRT_SERVER_TESTER and SQRT_SERVER_TESTER;1.

During system startup, the VAXELN kernel creates jobs for all programs that have the /initialize and /run attributes. Therefore, a job will be created running the program SQRT_SERVER. Since the program has the /debug attribute, the job will start under debugger control.

Any other programs that have the /run attribute (but not the /initialize attribute) run when the jobs with the /initialize attribute either exit, or call the INITIALIZATION_DONE procedure. Therefore, once SQRT_SERVER calls the INITIALIZATION_DONE procedure, the VAXELN kernel will create a job running the program SQRT_SERVER_TESTER.

2.1.3 Downline Loading the System Image

After the system has been built on the VAX/VMS host machine, it can be downline loaded to a target machine running the VAXELN executive. The host must have service enabled for the circuit to allow remote triggering of the node.

To downline load the target machine, the target machine must be running the downline load bootstrap loader. On a MicroVAX I or II, press the HALT button twice (in, and then out) and type:

```
>>> B XQAO
```

On the host, the following command causes the specified program to be downline loaded and begins a debugging session:

```
$ DEBUG/REMOTE ARTHUR /LOAD=SQRT_SERVER.SYS

        VAXELN Remote Debugger  Version V1.0-00

%RDEBUG-I-ATTEMPT_LOAD, Setting load file for node ARTHUR
        to TESTD:[ADA.EXE]SQRT_SERVER.SYS;
%RDEBUG-I-TRIGGER, Triggering node ARTHUR
%RDEBUG-I-ATTEMPT_CONNECT, Connecting to node ARTHUR
%RDEBUG-I-RETRY_CONNECT, Retrying connect to node ARTHUR
```

Note that if the host machine or the Ethernet is heavily loaded, the operation may timeout, as indicated by the following message.

```
%BOOT-F-ERROR, No response from load server XQAO
```

If this occurs, type

```
>>> B XQAO
```

on the target system's hardware console to restart the boot operation.

When the VAXELN system starts, the VAXELN kernel announces its presence on the target system's hardware console terminal:

```
                VAXELN V2.1-03
```

See Chapter 5 and the *VAXELN User's Guide* for further information on this topic and on downline loading non-MicroVAX machines.

2.1.4 Using the VAXELN Remote Debugger

In order to run a job under remote debugger control, you must have compiled and linked it with the /DEBUG qualifier in effect and you must specify that you want to run the job under debugger control by selecting the appropriate menu options when you build the system.

Once the remote debugger connects to the target node, it reports the job number, name, and state of each job in the system that is in a debug-wait state. In this case, SQRT_SERVER has started under control of the debugger and is therefore in a debug-wait state.

```
%RDEBUG-S-CONNECTION, Connected to node ARTHUR
Job 4.1 (SQRT_SERVER) is waiting for your attention
%RDEBUG-S-SET_TIME, System time set on node ARTHUR
RDBG*>
```

Type the SHOW SYSTEM command to get an overview of the current state of the jobs in the system. For example:

```
RDBG*> SHOW SYSTEM

      Job   Program     Priority   State    Shared Size  Readonly size
        2   XQDRIVER        1     waiting      31232         30208
        3   EDEBUGREM       3     running       5120         11264
        4   SQRT_SERVER    16    debug-wait     3584         66560
```

This command shows the jobs that are currently running on the system. The XQDRIVER job is the network driver. The EDEBUGREM job is the remote debugger nucleus. The SQRT_SERVER job is running the application to be debugged. Since the SQRT_SERVER job is in a debug-wait state, it is ready to start a command session. The SET JOB/CURRENT command begins a command session. For example:

```
RDBG*>  SET JOB/CURRENT SQRT_SERVER
%RDEBUG-I-SESSION_INIT, Loading symbols for Job 4. (SQRT_SERVER)
-RDEBUG-I-FROM, from file TESTD:[ADA.EXE]SQRT_SERVER.EXE;1
%RDEBUG-I-INITIAL, language is ADA, module set to 'SQRT_SERVER'
%RDEBUG-I-NOTATMAIN, type GO to get to start of main program
RDBG>
```

This command begins a command session with the job SQRT_SERVER. The remote debugger reads the symbol table information from the image file associated with the job and prepares to accept commands that may be directed at that job.

```
RDBG>  GO
break at routine SQRT_SERVER
       21: procedure SQRT_SERVER is
```

Typing the GO command causes the program to proceed directly to the beginning of the program.

For more information on how to use the remote debugger, see Chapter 6.

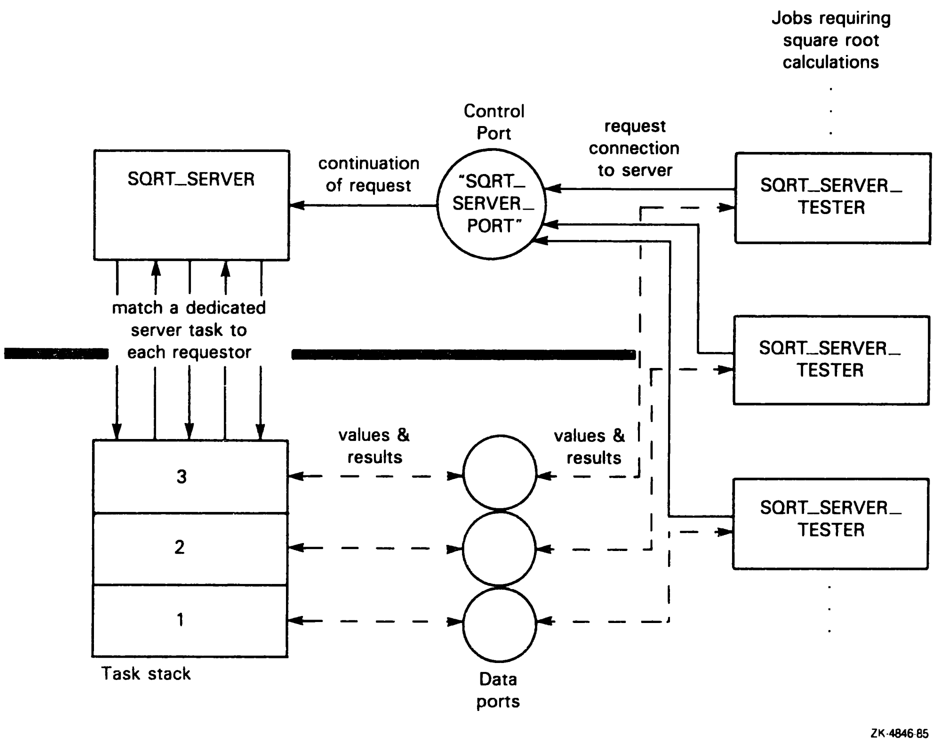
2.2 Sample Application

The following section presents a sample VAXELN Ada application. The application consists of three parts:

- The program SQRT_SERVER.ADA, which is an example of how a typical server might be implemented using Ada tasks and calls to VAXELN service routines. The function of this server is to compute square roots.
- The package ERROR_HANDLING, which contains a routine to display error messages.
- The program SQRT_SERVER_TESTER.ADA, which makes use of SQRT_SERVER.ADA, the sample server example program.

Figure 2–1 outlines the operations of this application. Program comments give a detailed explanation of the design and operation of the program.

Figure 2–1: Square-Root Server Application



2.2.1 Square-Root Server Program

```
-- SQRT_SERVER.ADA
--
with STARLET;
with VAXELN_SERVICES;
with CONDITION_HANDLING;  use CONDITION_HANDLING;
with SYSTEM; use SYSTEM;
with TEXT_IO; use TEXT_IO;
with FLOAT_MATH_LIB; use FLOAT_MATH_LIB;
with ERROR_HANDLING; use ERROR_HANDLING;

-- This program shows how a simple network-wide multithread server
-- can be implemented using VAXELN Ada and VAXELN kernel service routines.
--
-- The server uses a global VAXELN job port to receive VAXELN circuit
-- requests for square-root calculations from other VAXELN jobs.
-- When the server receives a request, it connects the requestor
-- (using a VAXELN circuit) to a dedicated Ada server task that
-- computes one or more square root values, depending on the
-- requestor's needs. When a server task detects that its service
-- is ended (the circuit with the requestor is disconnected), the
-- task makes itself available for another requestor and more
-- computations.
--
-- The square root calculations are passed as VAXELN messages between
-- the requestor and its Ada server task. Each message contains one
-- floating-point value. The square root of each value received by
-- the server task is computed using the SQRT function in the
-- MATH_LIB package available with VAXELN (and VAX) Ada.
--
-- Note that this program depends on the value of SYSTEM.SYSTEM_NAME,
-- which must be VAXELN.
--
```

```
procedure SQRT_SERVER is

   -- Server task availability is controlled using a stack of server
   -- task pointers. In this example, the stack size is 3.
   --
   MAX_SERVER_TASKS : constant := 3;
   subtype TASK_INDEX is INTEGER range 0 .. MAX_SERVER_TASKS;

   -- Task type for server tasks; note that to keep them from competing
   -- with the main task (procedure SQRT_SERVER), their priority is set
   -- to a lower priority of 4 (the main task has the default priority of 7).
   --
   task type SQRT_TASK is
      entry SQRT_ENTRY(TASK_ARRAY_INDEX : TASK_INDEX);
      pragma PRIORITY(4);
   end;

   -- The default working storage size for a VAXELN Ada task is 60
   -- physical pages (the pages are physical, not virtual, because
   -- VAXELN is a nonpaging system). Since the server task or tasks in
   -- this program do not use large amounts of stack space, it is safe and
   -- efficient to use a smaller working storage size of 12 pages.
   --
   for SQRT_TASK'STORAGE_SIZE use 12*512;

   type ACCESS_TO_TASK is access SQRT_TASK;

   -- The following objects and exceptions are used in the
   -- management of the server task stack:
   --
   STACK_GUARDIAN           : VAXELN_SERVICES.MUTEX_TYPE;
   NAME_OBJECT              : VAXELN_SERVICES.NAME_TYPE;
   JOB_PORT                 : VAXELN_SERVICES.PORT_TYPE;
   SERVER_TASK_AVAILABLE    : VAXELN_SERVICES.EVENT_TYPE;
   STATUS                   : CONDITION_HANDLING.COND_VALUE_TYPE;
   FIVE_MINUTES             : STARLET.DATE_TIME_TYPE;

   VAXELN_SERVICE_ERROR     : exception;
   SERVER_ALREADY_RUNNING   : exception;

   NUM_ACTIVE_SERVER_TASKS : INTEGER range 0 .. MAX_SERVER_TASKS := 0;
   ALL_TASKS_WERE_BUSY      : BOOLEAN;

   SERVER_TASK              : array (1 .. MAX_SERVER_TASKS) of ACCESS_TO_TASK;
   TASK_ARRAY_INDEX         : TASK_INDEX := 0;

   STACK_OF_IDLE_TASKS      : array (1 .. MAX_SERVER_TASKS) of TASK_INDEX;
   STACK_OF_IDLE_TASKS_TOP : TASK_INDEX := 0;
```

```
-- Task body for server tasks.
--
task body SQRT_TASK is

   MY_TASK_INDEX         : TASK_INDEX;
   RECEIVED_MESSAGE      : ADDRESS;
   MESSAGE_OBJECT        : VAXELN_SERVICES.MESSAGE_TYPE;
   RECEIVED_MESSAGE_SIZE : SYSTEM.UNSIGNED_LONGWORD;
   STATUS                : CONDITION_HANDLING.COND_VALUE_TYPE;
   WAIT_RESULT           : INTEGER;
   CIRCUIT_PORT          : VAXELN_SERVICES.PORT_TYPE;
   PARTNER_EXITED        : exception;
   TIMEOUT               : exception;
   BAD_MESSAGE_SIZE      : exception;
   SQRT_NEGATIVE         : exception;
   pragma IMPORT_EXCEPTION (SQRT_NEGATIVE, "MTH$_SQUROONEG");

   procedure MAKE_TASK_AVAILABLE is

   -- Each server task calls this procedure when the task
   -- is available for more work. In other words, MAKE_TASK_AVAILABLE
   -- is called when, for one reason or another, a server task has
   -- finished servicing one requestor and is ready to service another.
   --
   begin

      -- Push the calling server task on the STACK_OF_IDLE_TASKS.
      -- (In this example, a mutex is used to synchronize access
      -- to the stack.  You could also use an Ada task to achieve
      -- the same effect.)
      --
      VAXELN_SERVICES.LOCK_MUTEX (MUTEX => STACK_GUARDIAN);

      STACK_OF_IDLE_TASKS_TOP := STACK_OF_IDLE_TASKS_TOP + 1;
      STACK_OF_IDLE_TASKS(STACK_OF_IDLE_TASKS_TOP) := MY_TASK_INDEX;

      VAXELN_SERVICES.UNLOCK_MUTEX (MUTEX => STACK_GUARDIAN);
      -- Signal the VAXELN event that tells the main task
      -- that a server task is ready for more work.
      --
      VAXELN_SERVICES.SIGNAL_EVENT(
         STATUS => STATUS,
         EVENT  => SERVER_TASK_AVAILABLE);

      if not SUCCESS(STATUS) then
         raise VAXELN_SERVICE_ERROR;
      end if;

   end MAKE_TASK_AVAILABLE;
```

```
begin
   -- Create a VAXELN port that will be used by the server task
   -- to communicate with the requestor.
   --
   VAXELN_SERVICES.CREATE_PORT(
      STATUS => STATUS,
      PORT   => CIRCUIT_PORT);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;

   loop
      begin

         -- Force a circuit disconnection in case the finished
         -- requestor did not disconnect properly.
         --
         VAXELN_SERVICES.DISCONNECT_CIRCUIT(
            STATUS => STATUS,
            PORT   => CIRCUIT_PORT);

         accept SQRT_ENTRY(TASK_ARRAY_INDEX : TASK_INDEX) do

            -- Set the index into the server task stack (an array
            -- of pointers to SQRT_TASKs).
            --
            MY_TASK_INDEX := TASK_ARRAY_INDEX;

            -- Wait for a circuit connection to be established with
            -- a requestor.
            --
            VAXELN_SERVICES.ACCEPT_CIRCUIT(
               STATUS       => STATUS,
               CONNECT_PORT=> CIRCUIT_PORT,
               SOURCE_PORT => JOB_PORT);

         end SQRT_ENTRY;
         -- Check the status of the circuit connection (the check is
         -- made outside of the accept statement so that a possible
         -- resulting exception is not propagated up to the main task).
         --
         if not SUCCESS(STATUS) then
            raise VAXELN_SERVICE_ERROR;
         end if;
```

```
loop
   -- Wait for a message containing a computable floating-point
   -- value to arrive in the port (or timeout after 5 minutes).
   -- If the server does not receive a message from the
   -- requestor before the timeout expires, the server assumes
   -- that the requestor has implicitly terminated the dialog.
   --
   VAXELN_SERVICES.WAIT_ANY(
      STATUS      => STATUS,
      VALUE1      => CIRCUIT_PORT,
      TIME        => FIVE_MINUTES,
      RESULT      => WAIT_RESULT);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;

   if WAIT_RESULT = 0  then -- A WAIT_RESULT of 0 means that the
                            -- wait was satisfied by a timeout.
      raise TIMEOUT;
   end if;
   -- The wait has not timed-out; a message has arrived.
   -- Remove the message from the message port.
   --
   VAXELN_SERVICES.RECEIVE(
      STATUS       => STATUS,
      MESSAGE      => MESSAGE_OBJECT,
      DATA_ADDRESS => RECEIVED_MESSAGE,
      MESSAGE_SIZE => RECEIVED_MESSAGE_SIZE,
      SOURCE_PORT  => CIRCUIT_PORT);

   -- Check to see if the requestor has become disconnected.
   --
   if STATUS = VAXELN_SERVICES.KER_DISCONNECT then
      raise PARTNER_EXITED;
   else
      if not SUCCESS(STATUS) then
         raise VAXELN_SERVICE_ERROR;
      end if;
   end if;
```

```
      -- Check the size of the message, and raise an exception if
      -- it is not the same size as a floating-point value.
      --
      if RECEIVED_MESSAGE_SIZE /= FLOAT'SIZE/8 then
         raise BAD_MESSAGE_SIZE;
      end if;

      -- Use an Ada address representation clause to obtain the
      -- floating-point value from the received message to be
      -- used in the square-root computation.
      --
      declare
         MESSAGE_DATA : FLOAT;
         for MESSAGE_DATA use at RECEIVED_MESSAGE;
      begin
         -- Compute the square root of the value that was
         -- just received.
         --
         MESSAGE_DATA := SQRT(MESSAGE_DATA);

         -- Return the message to the sender.
         --
         VAXELN_SERVICES.SEND(
            STATUS           => STATUS,
            MESSAGE          => MESSAGE_OBJECT,
            MESSAGE_SIZE     => RECEIVED_MESSAGE_SIZE,
            DESTINATION_PORT => CIRCUIT_PORT);

         if not SUCCESS(STATUS) then
            raise VAXELN_SERVICE_ERROR;
         end if;
      end;

   end loop;
```

```
            -- Error handling related to message passing.
            --
            exception

               -- When the requestor disconnects the circuit,
               -- this task makes itself available for new work.
               --
               when PARTNER_EXITED =>
                  PUT_LINE("SQRT_SERVER_TASK - Partned Exited.");
                  MAKE_TASK_AVAILABLE;

               -- Note that this program does not attempt to recover from
               -- any errors (as a real server should). It simply displays
               -- a message and makes itself available for new work, thus
               -- ignoring the troublesome requestor.
               --
               when TIMEOUT =>
                  PUT_LINE("SQRT_SERVER_TASK - Timeout detected.");
                  MAKE_TASK_AVAILABLE;

               when BAD_MESSAGE_SIZE =>
                  PUT_LINE("SQRT_SERVER_TASK - A bad message was received"&
                           " (wrong size).");
                  MAKE_TASK_AVAILABLE;

               when SQRT_NEGATIVE =>
                  PUT_LINE("SQRT_SERVER_TASK - A bad message was received"&
                           " (negative number).");
                  MAKE_TASK_AVAILABLE;

            end;

      end loop;
      -- Error handling related to requestor connections.
      --
      exception
         when VAXELN_SERVICE_ERROR =>
            DISPLAY_ERROR_MESSAGE(STATUS);

            -- Clean up...
            -- Delete the port
            --
            VAXELN_SERVICES.DELETE_PORT(
               STATUS => STATUS,
               PORT   => CIRCUIT_PORT);

         when others =>
            PUT_LINE("SQRT_SERVER_TASK - An 'others' exception was raised."&
                     " Resignaling the exception");
            raise;

   end SQRT_TASK;
```

```
begin

   -- This is the body of the procedure SQRT_SERVER (the master process
   -- in the VAXELN job and the Ada program's main task). It services circuit
   -- requests coming in on the job port by calling an existing server task or
   -- creating a new one.
   --
   -- Begin by creating a VAXELN name object for the job port.
   --
   VAXELN_SERVICES.JOB_PORT(
      STATUS       => STATUS,
      PORT         => JOB_PORT);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;

   VAXELN_SERVICES.CREATE_NAME(
      STATUS    => STATUS,
      NAME      => NAME_OBJECT,
      PORT_NAME => "SQRT_SERVER_PORT",
      PORT      => JOB_PORT,
      SCOPE     => VAXELN_SERVICES.UNIVERSAL);

   -- If the name "SQRT_SERVER_PORT" is already defined,
   -- a SQRT_SERVER job is probably already running somewhere on
   -- the network, so exit.
   --
   if STATUS = VAXELN_SERVICES.KER_DUPLICATE then
      raise SERVER_ALREADY_RUNNING;
   end if;

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;

   -- Create the VAXELN mutex object that will serialize access to the
   -- stack of server tasks (note that stack access could also
   -- be controlled with a serializing Ada task).
   --
   VAXELN_SERVICES.CREATE_MUTEX(
      STATUS => STATUS,
      MUTEX  => STACK_GUARDIAN);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;
```

```
-- Create the VAXELN event object that each server task
-- will signal when it is ready for more work.
--
VAXELN_SERVICES.CREATE_EVENT(
   STATUS        => STATUS,
   EVENT         => SERVER_TASK_AVAILABLE,
   INITIAL_STATE => VAXELN_SERVICES.CLEARED);

if not SUCCESS(STATUS) then
   raise VAXELN_SERVICE_ERROR;
end if;

-- Get the delta time used in the calls to the VAXELN WAIT_ANY service
-- (the service is used to wait for connection requests coming in on
-- the job port named "SQRT_SERVER_PORT").
--
STARLET.BINTIM(
   STATUS => STATUS,
   TIMBUF => "0 00:05:00.00",
   TIMADR => FIVE_MINUTES);

-- This call to the VAXELN INITIALIZATION_DONE service informs the
-- VAXELN kernel that the job port initialization sequence is
-- complete (in this case, the job port name object has been created).
-- Jobs that count on this initialization being done can now be started.
--
-- Because of this required initialization, any job running this program
-- needs to have the Program Description Init required option set to "Yes"
-- in its System Builder data file.
--
VAXELN_SERVICES.INITIALIZATION_DONE (STATUS => STATUS);

if not SUCCESS(STATUS) then
   raise VAXELN_SERVICE_ERROR;
end if;

loop
   -- Wait for any requests for square roots on the job port.
   --
   VAXELN_SERVICES.WAIT_ANY(
      STATUS  => STATUS,
      VALUE1  => JOB_PORT);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;

   VAXELN_SERVICES.CLEAR_EVENT(
      STATUS => STATUS,
      EVENT  => SERVER_TASK_AVAILABLE);
```

```
         -- If there is a free server task, let it handle this request;
         -- or, if there are fewer than the maximum allowed tasks,
         -- create a new task to handle the request. If there are no
         -- free server tasks or all server tasks in the task stack
         -- are busy, wait for a server task to become free.
         --
         if STACK_OF_IDLE_TASKS_TOP = 0 and
            NUM_ACTIVE_SERVER_TASKS < MAX_SERVER_TASKS then

            -- Create a new server task to handle the incoming request.
            --
            NUM_ACTIVE_SERVER_TASKS := NUM_ACTIVE_SERVER_TASKS + 1;
            SERVER_TASK(NUM_ACTIVE_SERVER_TASKS) := new SQRT_TASK;

            -- Call the new task at its SQRT_ENTRY.
            --
            SERVER_TASK(NUM_ACTIVE_SERVER_TASKS).SQRT_ENTRY(NUM_ACTIVE_SERVER_TASKS);

         else

            -- No server tasks are available.  Wait for one to become inactive.
            --
            if STACK_OF_IDLE_TASKS_TOP = 0 then
               PUT_LINE("SQRT_SERVER - All server tasks are busy; waiting...");
               VAXELN_SERVICES.WAIT_ANY(SERVER_TASK_AVAILABLE);
               PUT_LINE("SQRT_SERVER - A server task is free; continuing...");
            end if;
            -- There is an inactive server task.
            -- First, lock the mutex that synchronizes access to the
            -- stack of idle tasks
            --
            VAXELN_SERVICES.LOCK_MUTEX (MUTEX => STACK_GUARDIAN);

            -- Next, choose the available task at the top of the stack,
            -- and call the task's SQRT_ENTRY.
            --
            TASK_ARRAY_INDEX := STACK_OF_IDLE_TASKS(STACK_OF_IDLE_TASKS_TOP);
            SERVER_TASK(TASK_ARRAY_INDEX).SQRT_ENTRY(TASK_ARRAY_INDEX);
            STACK_OF_IDLE_TASKS_TOP := STACK_OF_IDLE_TASKS_TOP - 1;

            -- Finally, allow other tasks to access the task stack.
            --
            VAXELN_SERVICES.UNLOCK_MUTEX (MUTEX => STACK_GUARDIAN);

         end if;
      end loop;

      exception
         when SERVER_ALREADY_RUNNING =>
            PUT_LINE("SQRT_SERVER already running elsewhere; exiting");

         when VAXELN_SERVICE_ERROR =>
            DISPLAY_ERROR_MESSAGE(STATUS);
         -- Clean up...
         -- Delete the name object "SQRT_SERVER_PORT"
         --
         VAXELN_SERVICES.DELETE_NAME(
            STATUS => STATUS,
            NAME   => NAME_OBJECT);

         -- Delete the mutex and event objects used by the main task
         --
         VAXELN_SERVICES.DELETE_MUTEX(
            STATUS => STATUS,
            MUTEX => STACK_GUARDIAN);

         VAXELN_SERVICES.DELETE_EVENT(
            STATUS => STATUS,
            EVENT  => SERVER_TASK_AVAILABLE);

end SQRT_SERVER;
```

2.2.2 Error-Handling Package

```
with SYSTEM;
with STARLET;
with CONDITION_HANDLING;

-- This package contains a simplified VAXELN interface to the VAX/VMS
-- Put Message service (STARLET.PUTMSG).  PUTMSG is a generalized
-- message formatting and output routine used to write informational
-- and error messages to the device used for error messages.
--
-- Note that this package does not depend on the value of SYSTEM.SYSTEM_NAME,
-- which can be either VAX_VMS or VAXELN.
--
package ERROR_HANDLING is

   procedure DISPLAY_ERROR_MESSAGE(
      MESSAGE_CODE : in CONDITION_HANDLING.COND_VALUE_TYPE);

end ERROR_HANDLING;
```

```
package body ERROR_HANDLING is

   procedure DISPLAY_ERROR_MESSAGE(
      MESSAGE_CODE : in CONDITION_HANDLING.COND_VALUE_TYPE) is

      -- This procedure displays the error text associated
      -- with a VAX/VMS or VAXELN condition value to the device
      -- used for error messages.
      -- For VAX/VMS this device is denoted by the logical name SYS$ERROR.
      -- For VAXELN this device is specified by the third program argument,
      -- if null, CONSOLE: is used.
      -- MESSAGE_CODE is a longword value that uniquely identifies the message.
      --
      MESSAGE_VECTOR : SYSTEM.UNSIGNED_LONGWORD_ARRAY(1 .. 2);
      PUTMSG_STATUS  : CONDITION_HANDLING.COND_VALUE_TYPE;
   begin

      -- Build the message vector that specifies the message to be
      -- written, and then call PUTMSG to display the error text.
      --
      MESSAGE_VECTOR(1) := 1;   -- One argument follows
      MESSAGE_VECTOR(2) := SYSTEM.UNSIGNED_LONGWORD(MESSAGE_CODE);
      STARLET.PUTMSG (
         STATUS => PUTMSG_STATUS,
         MSGVEC => MESSAGE_VECTOR);

   end DISPLAY_ERROR_MESSAGE;

end ERROR_HANDLING;
```

2.2.3 Square-Root Server Application Program

```
-- SQRT_SERVER_TESTER.ADA
--
with VAXELN_SERVICES;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with SYSTEM; use SYSTEM;
with TEXT_IO; use TEXT_IO;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
with ERROR_HANDLING; use ERROR_HANDLING;

-- This example program is a simple application that makes use
-- of the square-root server program (SQRT_SERVER.ADA).
-- The application starts by connecting its job port to the
-- square-root server and passing it a FLOAT'SAFE_LARGE value.
-- It then accepts the square root of that value from the server
-- and sends it right back again.  This exchange continues until
-- the value returned is as close to 1.0 as possible.  The dialog
-- with the square-root server ends when the circuit is disconnected.
--
-- Note that this program depends on the value of SYSTEM.SYSTEM_NAME,
-- which must be VAXELN.
--
procedure SQRT_SERVER_TESTER is

   INITIAL_MESSAGE_ADDRESS  : ADDRESS;
   INITIAL_MESSAGE_OBJECT   : VAXELN_SERVICES.MESSAGE_TYPE;
   INITIAL_MESSAGE_SIZE     : SYSTEM.UNSIGNED_LONGWORD;

   MESSAGE_ADDRESS          : ADDRESS;
   MESSAGE_OBJECT           : VAXELN_SERVICES.MESSAGE_TYPE;
   MESSAGE_SIZE             : SYSTEM.UNSIGNED_LONGWORD;

   SAVED_MESSAGE_DATA       : FLOAT := 0.0;

   OUR_JOB_PORT             : VAXELN_SERVICES.PORT_TYPE;
   STATUS                   : CONDITION_HANDLING.COND_VALUE_TYPE;
   VAXELN_SERVICE_ERROR     : exception;
```

```
begin

   -- Start by establishing a VAXELN circuit connection between this
   -- job's job port and the square-root server task's input port.
   -- The connection is made by using the global name -- "SQRT_SERVER_PORT".
   -- Once a server task is connected to the square-root server tester,
   -- the exchange of values and their square roots can take place between
   -- the tester and its dedicated server task.
   --
   VAXELN_SERVICES.JOB_PORT(
      STATUS  => STATUS,
      PORT    => OUR_JOB_PORT);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;

   VAXELN_SERVICES.CONNECT_CIRCUIT(
      STATUS           => STATUS,
      SOURCE_PORT      => OUR_JOB_PORT,
      DESTINATION_NAME => "SQRT_SERVER_PORT");

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;
   -- Create a message that will contain the floating-point value
   -- used in the initial square root calculation. The VAXELN CREATE_MESSAGE
   -- procedure creates the message object and an associated data
   -- buffer. The address of the data buffer is returned in the DATA_ADDRESS
   -- parameter of CREATE_MESSAGE.
   --
   INITIAL_MESSAGE_SIZE := FLOAT'SIZE/8;    -- Compute the size
                                            -- (in bytes) of the
                                            -- message; required
                                            -- by CREATE_MESSAGE.

   VAXELN_SERVICES.CREATE_MESSAGE(
      STATUS       => STATUS,
      MESSAGE      => INITIAL_MESSAGE_OBJECT,
      DATA_ADDRESS => INITIAL_MESSAGE_ADDRESS,
      MESSAGE_SIZE => INITIAL_MESSAGE_SIZE);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;
```

```
-- Use an address representation clause to obtain the
-- address of the data buffer and store the initial
-- floating-point value into that data buffer.
--
declare
   INITIAL_MESSAGE_DATA : FLOAT := FLOAT'SAFE_LARGE;
   for INITIAL_MESSAGE_DATA use at INITIAL_MESSAGE_ADDRESS;
begin

   PUT_LINE ("SQRT_SERVER_TESTER - Sending initial message to SQRT_SERVER");

   -- Send the first computable value to the square-root server.
   --
   VAXELN_SERVICES.SEND(
      STATUS           => STATUS,
      MESSAGE          => INITIAL_MESSAGE_OBJECT,
      MESSAGE_SIZE     => INITIAL_MESSAGE_SIZE,
      DESTINATION_PORT => OUR_JOB_PORT);

   if not SUCCESS(STATUS) then
      raise VAXELN_SERVICE_ERROR;
   end if;
end;

loop
   begin

      -- Wait for a response from the square-root server task
      -- to arrive in the port.
      --
      VAXELN_SERVICES.WAIT_ANY(
         STATUS       => STATUS,
         VALUE1       => OUR_JOB_PORT);

      -- The message has arrived!  Remove it from the port.
      --
      VAXELN_SERVICES.RECEIVE(
         STATUS       => STATUS,
         MESSAGE      => MESSAGE_OBJECT,
         DATA_ADDRESS => MESSAGE_ADDRESS,
         MESSAGE_SIZE => MESSAGE_SIZE,
         SOURCE_PORT  => OUR_JOB_PORT);

      if not SUCCESS(STATUS) then
         raise VAXELN_SERVICE_ERROR;
      end if;

      -- Use an address representation clause to obtain the
      -- floating-point value from the message.
      --
      declare
         MESSAGE_DATA : FLOAT;
         for MESSAGE_DATA use at MESSAGE_ADDRESS;
      begin

         PUT ("SQRT_SERVER_TESTER - Received: ");
         PUT (MESSAGE_DATA, AFT => 10);
         NEW_LINE;
         -- Exit when the square-root calculations are finished
         -- (that is, when they come as close to 1.0 as possible).
         --
         exit when SAVED_MESSAGE_DATA = MESSAGE_DATA;

         SAVED_MESSAGE_DATA := MESSAGE_DATA;

         -- Send back the message using the same message object.
         --
         VAXELN_SERVICES.SEND(
            STATUS           => STATUS,
            MESSAGE          => MESSAGE_OBJECT,
            MESSAGE_SIZE     => MESSAGE_SIZE,
            DESTINATION_PORT => OUR_JOB_PORT);

         if not SUCCESS(STATUS) then
            raise VAXELN_SERVICE_ERROR;
         end if;
      end;
   end;
end loop;
-- Clean up...
-- Delete the message object.
--
VAXELN_SERVICES.DELETE_MESSAGE(
   STATUS       => STATUS,
   MESSAGE      => INITIAL_MESSAGE_OBJECT);

if not SUCCESS(STATUS) then
   raise VAXELN_SERVICE_ERROR;
end if;

-- Disconnect the job port from the square-root server.
-- This makes the square-root server task available to
-- accept work from other requestors.
--
VAXELN_SERVICES.DISCONNECT_CIRCUIT(
   STATUS       => STATUS,
   PORT         => OUR_JOB_PORT);

if not SUCCESS(STATUS) then
   raise VAXELN_SERVICE_ERROR;
end if;

PUT_LINE ("SQRT_SERVER_TESTER - Successfully exiting");
```

```
   -- Error handling for bad message transmission or
   -- any other error returned by a VAXELN service.
   --
   exception
      when VAXELN_SERVICE_ERROR =>
         DISPLAY_ERROR_MESSAGE(STATUS);

         -- Clean up...
         -- Attempt to delete the message object,
         -- disconnect the circuit, and exit.
         --
         VAXELN_SERVICES.DELETE_MESSAGE(
            STATUS       => STATUS,
            MESSAGE      => INITIAL_MESSAGE_OBJECT);

         VAXELN_SERVICES.DISCONNECT_CIRCUIT (
            STATUS       => STATUS,
            PORT         => OUR_JOB_PORT);

end SQRT_SERVER_TESTER;
```

Chapter 3

Program Compilation and Linking

VAXELN Ada programs are compiled and linked on a VAX/VMS host system using the VAX Ada compiler and the VAX Ada program library manager (ACS). Thus, all of the information relevant to VAX Ada program development is also relevant to VAXELN Ada program development, and *Developing Ada Programs on VAX/VMS* is the best source of detailed program development information.

However, for convenience and completeness, this chapter summarizes the most important information on how to compile and link VAXELN Ada programs. It also summarizes the syntax and properties of the ACS commands, and points out features of interest to VAXELN Ada programmers in particular. Detailed information on related Ada language features is given in the *VAX Ada Language Reference Manual.*

3.1 Basic Concepts and Terminology

This section reviews the basic concepts and terminology necessary for understanding how compiling and linking work in the VAX Ada and VAXELN Ada program library (ACS) environment. These concepts are related to modular program development, which is a primary feature of the Ada language.

3.1.1 Program and Compilation Units

Program units are the functional building blocks of Ada programs. There are four kinds of program units: subprograms (procedures and functions), packages, tasks, and generic units. An Ada program generally consists of a *main program* and its related program units. A main program is always a subprogram.

To facilitate modular development, each program unit consists of a specification and a body. The specification contains only the declarations that need to be made visible to other program units; the body contains the implementation of the declarations in the specification.

The parts of Ada program units that can be compiled separately are called *compilation units.* Compilation units consist of the following:

- Subprogram specifications and bodies
- Package specifications and bodies
- Generic unit (subprogram and package) specifications and bodies
- Generic instantiations (subprogram and package) of generic units
- Subunits

NOTE

Tasks are the only program units that cannot be compiled separately. A task specification or body must be contained within a package or a subprogram before it can be compiled. A task body can be a subunit, however, and subunits can be compiled separately.

The Ada language distinguishes between two classes of compilation units:

- *Library units* are the compilation units that are essential for program compilation. They consist of library unit specifications, or *library specifications* (consisting of subprogram, package, and generic specifications), generic instantiations, and subprogram bodies that do not have separate specifications.
- *Secondary units* are the compilation units that are not essential for program compilation, but are essential for program linking and running. They consist of library unit bodies, or *library bodies* (consisting of subprogram, package, and generic bodies), and subunits.

Thus, you can begin Ada program development by designing and compiling a program consisting only of library units. Once your program's structure is established, you can add the secondary units (bodies and subunits). This process is designed to organize the development of large, complex programs, but it works efficiently for the development of programs of any size or complexity.

3.1.1.1 Compilation Unit Dependences

During and after compilation, the compiler and ACS maintain current data on the status of compilation units and the *dependences* among units. In this way, the compiler can enforce certain order-of-compilation rules, and ACS can manage the program library to support those rules.

Compilation unit dependences are derived from Ada's scope and visibility conventions:

- A library body *depends* on its library specification, if there is one.
- A subunit *depends* on its parent unit and therefore depends on its parent's associated library body and library specification.
- Each compilation unit depends on any units that are named in any context clauses. (A context clause consists of one or more **with** clauses and optional **use** clauses.) More precisely, each compilation unit depends on the library specifications of any units that are named in context clauses.

Note that target-related dependences are caused by the value of the predefined constant SYSTEM_NAME in package SYSTEM. This constant and its effects are described in Section 3.2. Order-of-compilation rules are described in Section 3.1.2.

3.1.1.2 Current and Obsolete Units

The VAX Ada compiler and ACS keep track of the date-time of the most recent compilation of a unit. Whenever a unit is compiled, any dependent unit, as defined in Section 3.1.1.1, is made *obsolete* and must eventually be compiled again before the set of units can be linked. For example, compiling a library specification makes the associated library body and any subunits obsolete; moreover, if the library specification is named in a context clause of a given unit, the given unit is also made obsolete, as are its dependent units.

You need to know about obsolete units when you link the units comprising your program. If you try to link a set of units that contains any obsolete units, ACS warns you about those units and terminates the operation. Because obsolete units are a natural consequence of Ada's compilation rules (see Section 3.1.2), ACS has a RECOMPILE command that automatically finds dependent units that have been made obsolete and compiles them in the right order. This process makes the units *current*. In keeping with this command, the verb *to recompile* is used in a very restricted sense in this chapter: it means to use the ACS RECOMPILE command to compile and make current a set of obsolete dependent units.

3.1.1.3 Unit and File-Naming Conventions

While doing program development in the ACS environment, you should be aware of the distinction between source files and units. A source file (having a default file type of .ADA) can contain several compilation units. However, after compilation of the file, ACS maintains information about the individual units, and most of the ACS commands operate on units (not source files). Thus, it is recommended that you use a separate source file for each compilation unit.

Use of a separate source file for each compilation unit also promotes efficient use of the compiler. Every time a unit is compiled, any dependent unit in the program library is made obsolete and must be recompiled. If, for example, you have placed two library specifications in the same source file, every time you modify one, you must compile both in the same compilation. Thus, you would have to recompile dependent units for both specifications rather than for only the one you modified.

The Ada language has some specific rules about compilation unit names, which you should observe when you name your compilation-unit source files. For example, although a library specification and its library body are distinct compilation units, they share the same name, called the *unit name*. All of the unit names in a program library must be unique. Similarly, all of the subunit names associated with a given ancestor library unit must be unique (every subunit mentions the name of its *parent unit*, and if the parent unit is a library unit, the parent is called the *ancestor* library unit).

Thus, the following file-naming conventions are recommended:

- The name of the source file for a *library specification* should be the name of the unit, followed by a trailing underscore character (_): for example, ERROR_HANDLING_.ADA.

- The name of the source file for a *library body* should be the name of the unit (without a trailing underscore): for example, ERROR_HANDLING.ADA.
- The name of the source file for a *library generic instantiation* should be the name of the instantiation: for example, FLOAT_MATH_LIB.ADA.
- The name of the source file for a *subunit* should be the name of the ancestor unit, followed by two underscore characters, followed by the name of the subunit: for example, ERROR_HANDLING__BAD_DATA_ERROR.ADA (if procedure BAD_DATA_ERROR were indeed a subunit of the ERROR_HANDLING package shown in Chapter 2).

These conventions are consistent with ACS and VAX/VMS file-naming conventions, and they are also used by the VAXELN Remote Debugger for the names of modules and pathnames.

3.1.2 Order-of-Compilation Rules

The VAX Ada compiler and ACS enforce the rules governing the order in which compilation units are compiled. These order-of-compilation rules stem from Ada's scope and visibility conventions, which create the dependences among units described in Section 3.1.1.1. The rules are as follows:

- You can compile a given unit only *after* compiling all library specifications named in that unit's context clause.
- You can compile a library body only *after* compiling its library specification. Note, however, that the body of a nongeneric library subprogram can also serve as its own library specification and, therefore, does not necessarily depend on a separately compiled specification.
- You can compile a subunit only *after* compiling its parent unit.

In summary, a unit must be compiled *before* any of its dependent units.

If you follow these rules, then the following are true:

- You can submit the compilation units of a program to the compiler in one or more compilations (invocations of the compiler). Also, you can submit one or more compilation units of a program at any one time. The units of any one compilation are compiled in the given order, whether submitted in one or more files. Thus, a pragma that applies to the whole of a compilation must appear before the first unit of that compilation.

- Units can be compiled in an otherwise arbitrary order relative to each other. For example, compiling a subunit affects only its subunits, if any; and compiling a library body generally does not affect any other units except its own subunits, if any. Compiling a library body affects other units in the following two cases:
  - Compiling a library body that is a generic template makes all units containing instantiations of that generic unit obsolete.
  - If a library specification specifies pragma INLINE, then compiling the library body makes a given unit that mentions the library specification in a **with** clause obsolete, if the given unit has a call that was actually expanded inline.

If these rules are not violated when you compile a unit or set of units, and no other errors are detected, then the program library is *updated*. When the program library is updated for a unit that already exists in the library, the previous versions of its associated files (see Section 3.3.1) are deleted. If the compilation is unsuccessful for any reason, no updating is done.

Note that the VAX Ada compiler always processes compilation units in a manner that is consistent with Ada's order-of-compilation rules. However, observance of the compilation rules does not ensure that the set of units in a program library is *current*. Nor does observance of the rules ensure that the set of units is *complete*. For example, a library body or a subunit may still be missing from the program library, or may have been made obsolete by the previous compilation. If you try to link an incomplete set of units, the library manager will warn you about the missing units, and terminate the operation. Obsolete units are discussed in Section 3.1.1.2; what constitutes a complete set of units is discussed in Section 3.1.3.

3.1.3 Closure

When you compile a given unit, the compiler identifies any unit that the given unit depends on, as specified in Section 3.1.1.1, and determines whether that unit is defined in the current program library. For example, if the given unit is a library body, the compiler looks for its specification.

Any unit that the given unit depends on may itself depend on yet another unit, which must also be defined in the current program library. The total set of units that the given unit depends on, directly and indirectly, is called the *compilation closure* of the given unit. Thus, the compilation closure of a given unit consists of all the units that must be defined in the current program library before you can compile the given unit.

To link a program into an executable image, it is necessary to form the *execution closure* of the main program. The execution closure consists of the compilation closure plus all associated secondary units (library bodies and subunits).

Several ACS commands operate on the execution closure of a specified set of units—for example, the CHECK, COMPILE, COPY UNIT/CLOSURE, ENTER UNIT/CLOSURE, EXPORT, LINK, RECOMPILE, and SHOW PROGRAM commands. In the remainder of this chapter, the term closure is used to signify execution closure, unless specified otherwise.

The execution closure of a specified set of compilation units is defined formally as the smallest set of units such that

- All the specified units are contained in the closure.
- For *any* given unit in the closure, the following are also contained in the closure, as applicable:
  - Its specification, if the given unit is a body
  - Its body, if the given unit is a specification
  - Its immediate subunits, if any
  - Its immediate parent unit, if the given unit is a subunit
  - All units named by the given unit in its context clause

Note that a unit that names a given unit in its context clause is not part of the execution closure of the given unit.

3.2 Setting the System Name

VAXELN Ada and VAX Ada have different run-time libraries. Also, because of differences between the VAXELN executive and the VAX/VMS operating system, VAXELN Ada has some target-specific differences (see Chapter 1). ACS, as the interface to the VAX Ada compiler and VAX/VMS Linker, is sensitive to these differences through the value of the predefined constant SYSTEM_NAME in package SYSTEM. This constant can have a value of either VAXELN or VAX_VMS.

The value of SYSTEM.SYSTEM_NAME does not cause the compiled code to differ. It is used to determine target-related compilation unit dependences, which can occur in your Ada code in the following cases:

- Use of SYSTEM.SYSTEM_NAME causes either a VAX_VMS or a VAXELN dependence.

- Use of pragma TIME_SLICE causes a VAX_VMS dependence.
- Use of pragma AST_ENTRY or the AST_ENTRY attribute causes a VAX_VMS dependence.
- Use of any of the relative or indexed input-output packages causes a VAX_VMS dependence.
- Use of the package VAXELN_SERVICES causes a VAXELN dependence.

For example, if a compilation unit uses pragma AST_ENTRY and the system name at compile time is VAXELN, you are warned that your unit depends on SYSTEM.SYSTEM_NAME and that this pragma is ignored for a VAXELN target. Similarly, if a unit uses the AST_ENTRY attribute and the system name at compile time is VAXELN, you are warned that your unit depends on SYSTEM.SYSTEM_NAME and that your use of the attribute is illegal.

When you create an ACS program library or sublibrary, the default value of SYSTEM.SYSTEM_NAME is VAX_VMS. You can use the /SYSTEM_NAME qualifier on the ACS CREATE LIBRARY or CREATE SUBLIBRARY command to explicitly determine the value of SYSTEM.SYSTEM_NAME, or you can permanently set the system name to VAXELN (or set it back to VAX_VMS) by

- Compiling the predefined Ada pragma SYSTEM_NAME.
- Executing the ACS SET PRAGMA command (ACS SET PRAGMA /SYSTEM_NAME=VAX_VMS or ACS SET PRAGMA/SYSTEM_NAME=VAXELN).

To determine the current setting for your current program library, you can use the ACS SHOW LIBRARY/FULL command; to determine system-name dependences for individual program units, you can use the ACS SHOW PROGRAM command.

You can temporarily override the current setting when you link or export units by using the /SYSTEM_NAME qualifier on the ACS LINK and EXPORT commands. For example, if you are working in a VAX/VMS environment (SYSTEM.SYSTEM_NAME=VAX_VMS), and the units you have compiled do not contain any of the VAX/VMS-specific features, you can link them for a VAXELN target with the ACS LINK/SYSTEM_NAME=VAXELN command. However, a link-time error occurs if a unit depends on the value of SYSTEM.SYSTEM_NAME and a /SYSTEM_NAME qualifier specifies a different value. The LINK and EXPORT commands are discussed in more detail in Sections 3.5 and 3.5.2.

Note that when you use pragma SYSTEM_NAME or the ACS SET PRAGMA command to change the system name (either with an argument of VAX_VMS or VAXELN), an implicit recompilation of package SYSTEM occurs. Those units that depend on the value of SYSTEM.SYSTEM_NAME are then made obsolete, and must be recompiled in the context of the new system name. For example, suppose you have the following program (dashed lines separate the individual compilation units):

```
with TASK_WORK;
procedure ALL_WORK is          -- Main program, depends on
                               -- target-dependent TASK_WORK
begin
   . . .
   TASK_WORK;
   . . .
end ALL_WORK;
-------------------------------------------------------------------
procedure TASK_WORK is         -- VAX/VMS-dependent procedure

   pragma TIME_SLICE(0.4);

   task type T;

   type TASK_FORCE_TYPE is
      array (INTEGER range 1..5) of T;

   TASK_FORCE: TASK_FORCE_TYPE;

   task body T is separate;    -- Task body is a subunit

begin
   . . .
end TASK_WORK;
-------------------------------------------------------------------
with TEXT_IO; use TEXT_IO;
separate (TASK_WORK)

task body T is                 -- Target-independent subunit depends on
                               -- target-independent package TEXT_IO
                               -- and target-dependent ancestor, TASK_WORK
begin
  PUT_LINE ("My work's just starting...");
   . . .
  delay 3.0;
   . . .
  PUT_LINE ("My work's all done!");
end T;
```

If you compile these units into a program library for which SYSTEM.SYSTEM_NAME equals VAX_VMS, and subsequently use the ACS SET PRAGMA command to set SYSTEM_NAME to VAXELN, then

- Procedure TASK_WORK becomes obsolete because it depends on SYSTEM_NAME=VAX_VMS.

- The main program, ALL_WORK, becomes obsolete because it depends on procedure TASK_WORK.
- Subunit TASK_WORK.T becomes obsolete because it depends on its ancestor, TASK_WORK.

All three units would have to be recompiled before they could be linked, and recompilation would result in a warning because pragma TIME_SLICE is ignored for VAXELN targets. Section 3.1.1.1 discusses unit dependences in more detail. Obsolete units and recompilation are described in Section 3.1.1.2.

3.3 Working with Program Libraries and Sublibraries

For compiling and linking VAXELN Ada programs—as for developing VAX Ada programs—you need to have a program library, and may wish to use sublibraries. *Developing Ada Programs on VAX/VMS* describes program libraries and sublibraries in detail. However, for convenience, Section 3.3.1 reviews program library and sublibrary concepts; Section 3.3.2 reviews how to set up a program library and summarizes the ACS program and sublibrary management commands. Section 3.3.3 briefly describes how to work in an environment containing programs that have been developed to run on both VAXELN and VAX/VMS targets.

3.3.1 VAXELN Ada Program Libraries and Sublibraries

Program libraries and sublibraries—for either VAXELN Ada or VAX Ada—are special, dedicated VAX/VMS (sub)directories that contain the following files:

- A *library index file* (ADALIB.ALB) that identifies all the other files in the program library. ACS uses the library index file to associate unit and subunit names with their related VAX/VMS file specifications. ACS also uses the library index file to record the date and time when the VAX/VMS files were created or revised.
- A set of up to three files for each compilation unit successfully compiled:
  - *Object file* (.OBJ)—to contain the machine code instructions for that compilation unit.

  - *Compilation unit file* (.ACU)—to contain the intermediate representation of a compilation unit. This file contains data that is used to support separate compilation, linking, and program library management. The data includes the name of the compilation unit; whether it is a specification, a body, or a subunit; use of certain pragmas, and so on. The compilation unit file also identifies all library specifications that the given compilation unit depends on.
  - *Copied source file* (.ADC)—to contain a copy of the source text for the given compilation unit. Copied source files are optional, and are used primarily for recompilation.

ACS uses these files to automate library operations during compilation and linking, as well as to automate library management in general. Any change to a program library (or sublibrary), such as writing or deleting files or data, is called *updating* the library. Updating is a consequence of successful compilation (see Section 3.1.2).

You can organize program libraries and sublibraries to suit the needs of your project. The compilation units of an entire Ada program can be stored in a single program library, or they can be distributed among a number of program libraries. ACS allows compilation units to be shared among program libraries, either by direct copy, using the ACS COPY UNIT command, or by reference, using the ACS ENTER UNIT command.

The difference between program libraries and sublibraries is that sublibraries exist in the context of a parent library. Units in a sublibrary are thus compiled in the context of both the sublibrary and the parent library, but only the sublibrary is updated with new files and index entries. Thus, you can use sublibraries to isolate particular compilation units for individual development. When the units in the sublibrary are stable, you can merge them into the parent library using the ACS MERGE command. A particular use of sublibraries for developing mixed VAXELN and VAX Ada programs is described in Section 3.3.3.

Note that a sublibrary's unique structure sets the following scope conventions:

- A compilation unit that has been compiled, copied, or entered into a parent library can be used in a sublibrary of that parent library as if the unit had been entered into the sublibrary.
- A unit in a parent library is hidden from a sublibrary of the parent if a unit of the same name has been compiled, copied, or entered into the sublibrary.
- The scope conventions for nested sublibraries are an extension of those for a single sublibrary.

- The visibility of unit names is determined from the bottom up; that is, the compiler starts searching in the sublibrary that is defined to be the current program library. If the unit name is not found in the sublibrary, the compiler then searches its parent library, and so on up the library tree.

A more detailed summary of these conventions, as well as illustrations of the use of sublibraries, is presented in *Developing Ada Programs on VAX/VMS*.

3.3.2 Setting Up and Working in a VAXELN Ada Development Environment

You set up a VAXELN Ada development environment just as you would set up a VAX Ada development environment: by using a series of ACS commands to create a program library and applicable sublibraries on your VAX/VMS host (see Section 3.3.1 for a summary of those commands).

First, you use the DCL CREATE/DIRECTORY command to define a VAX/VMS subdirectory to contain your VAXELN Ada source files:

```
$ CREATE/DIRECTORY [SMITH.SERVER]
```

Then, you use the ACS CREATE LIBRARY command to set up and initialize a program library:

```
$ ACS CREATE LIBRARY [SMITH.SERVER.PROGLIB]
```

Note that you should use a separate VAX/VMS subdirectory for your program library because it is wise not to alter any of the files in your program library.

If you need a sublibrary for testing or modifying particular units in your program library, you can use the DCL CREATE/DIRECTORY command to define a VAX/VMS subdirectory for containing copies of the Ada source files you intend to work with:

```
$ CREATE/DIRECTORY [SMITH.TEST]
```

Then, you use the ACS CREATE SUBLIBRARY command to set up and initialize the program sublibrary:

```
$ ACS CREATE SUBLIBRARY/PARENT=[SMITH.SERVER.PROGLIB] -
_$ [SMITH.TEST.SUBLIB]
```

Again, you should use a separate VAX/VMS subdirectory for your sublibrary, to avoid altering any of the files that the library manager creates and updates.

Note that you may want to specify the parent library when creating a sublibrary; the default parent is the current (sub)library, and it is possible to accidentally create the wrong structure. You can use the ACS SHOW LIBRARY/FULL command with your sublibrary name to determine its parent.

You can use the ACS SET LIBRARY command to determine the current library or sublibrary for the products of compilation:

```
$ ACS SET LIBRARY [SMITH.SERVER_PROGLIB]
```

or

```
$ ACS SET LIBRARY [SMITH.TEST.SUBLIB]
```

All of the ACS commands for managing program libraries and sublibraries are described in detail in *Developing VAX Ada Programs on VAX/VMS*. For quick reference, the management commands are summarized in Tables 3–1 and 3–2. Online descriptions are also available using the ACS HELP command.

Table 3–1: ACS Program Library Management Commands

Command	Function
CHECK unit-name[,...]	Forms the closure of the given units and checks the completeness and currency of all the units in the closure.[1]
COPY FOREIGN file-spec unit-name	Copies the given foreign (non-Ada) object file into the current program library as a library unit body.
COPY UNIT from-directory-spec unit-name[,...]	Copies one or more compiled units from another program library into the current program library.
CREATE LIBRARY directory-spec	Creates a VAX Ada program library.
CREATE SUBLIBRARY directory-spec	Creates a VAX Ada program sub-library, and initializes it to refer to the parent program library; allows you to isolate the development of selected units.

[1]In simple terms, "closure" is the complete set of units that a given unit depends on, plus any other units needed for its execution. *Closure* and *currency* are discussed briefly in Sections 3.1.3 and 3.1.1.2.

Table 3–1 (Cont.): ACS Program Library Management Commands

Command	Function
DELETE LIBRARY directory-spec	Deletes the given program library and its contents.
DELETE SUBLIBRARY directory-spec	Deletes the given program sublibrary and its contents.
DELETE UNIT unit-name[,...]	Deletes the given units from the current program library.
DIRECTORY [unit-name[,...]]	Alphabetically lists the units in the current program library. Displays information, such as name and date-time of the last compilation, about the given units.
ENTER FOREIGN file-spec unit-name	Enters a reference (pointer) to the given external file into the current program library. The entered file is then recognized as a foreign (non-Ada) library body for the given unit.
ENTER UNIT from-directory-spec unit-name[,...]	Enters references, in the current program library, to one or more units located in another program library. *Entered* units can be used in the current program library as if they were actually in it.
EXTRACT SOURCE unit name[,...]	Creates, in the current default directory, copies of source files for the given units.
MERGE unit-name[,...]	Merges, into the parent library, new versions of one or more units from the sublibrary where they were modified. The MERGE command replaces the older obsolete versions in the parent library.

Table 3–1 (Cont.): ACS Program Library Management Commands

Command	Function
REENTER unit name[,...]	Enters current references to the given units. Presumes that the units were compiled after they were last entered with the ENTER UNIT command.
SET LIBRARY directory-spec	Defines the given (sub)directory to be the current program library—that is, the library that is to be the compilation context and the target library for compiler output and ACS commands.
SET PRAGMA/pragma-name=value	Redefines specified values of the pragmas LONG_FLOAT, MEMORY_SIZE, and SYSTEM_NAME.
SHOW LIBRARY [directory-spec[,...]]	Displays the name and characteristics of one or more program libraries.
SHOW PROGRAM unit-name[,...]	Displays information, such as dependence on other units, about the closure of the given units. Optionally displays a portability summary.
VERIFY [directory-spec]	Performs a series of consistency checks on the given program library to determine whether the library structure and library files are in valid form. Corrects some of the inconsistencies detected.

Table 3–2: Additional ACS Commands

Command	Function
ATTACH process-name	Switches control of your terminal from your current process running ACS to another process in your job.
EXIT	Exits from ACS. You can also use CTRL/Z.
HELP [keyword ...]	Invokes the VAX/VMS HELP facility to provide information about ACS commands.
SPAWN [DCL-command]	Creates a subprocess of the current process and suspends execution of the current process.

3.3.3 Working in a Mixed VAXELN Ada and VAX Ada Environment

The key to working with mixed-target programs is that you can link units that have been compiled with a system name of either VAXELN or VAX_VMS, as long as the units do *not* involve the following target-dependent features:

- Pragma TIME_SLICE
- Pragma AST_ENTRY
- The AST_ENTRY attribute
- Relative and indexed input-output packages
- Package VAXELN_SERVICES

Thus, if you need to write programs that can run on both VAXELN and VAX/VMS targets, an efficient way to organize your code is to use target-independent specifications and target-dependent bodies, and then collect the bodies into sublibraries (see Section 3.3.1).

For example, suppose you needed to have a square-root server program that could run on both VAXELN and VAX/VMS targets, but you wanted to minimize the amount of work involved in writing it. You might organize the SQRT_SERVER program shown in Chapter 2 as follows:

- Project program library: [PROJ.SERVER.PROGLIB]

  This library contains only the parts of the program that would be common to both implementations: the specification and body of ERROR_HANDLING, as well as the specifications for the main program and the test program:

  ERROR_HANDLING_ (specification)
  ERROR_HANDLING (body)
  SQRT_SERVER_ (specification)
  SQRT_SERVER_TESTER_ (specification)

- VAXELN sublibrary: [PROJ.SERVER.VAXELN.SUBLIB]

  This sublibrary (created with the ACS CREATE SUBLIBRARY /SYSTEM_NAME=VAXELN command) contains the VAXELN versions of the main program and test program bodies:

  SQRT_SERVER
  SQRT_SERVER_TESTER

- VAX/VMS sublibrary: [PROJ.SERVER.VAXVMS.SUBLIB]

  This sublibrary (created with the ACS CREATE SUBLIBRARY command), contains the VAX/VMS versions of the main program and test program bodies:

  SQRT_SERVER
  SQRT_SERVER_TESTER

By using this organization, you can compile and link against the common units in the parent library while keeping the target-dependent units separate.

A sample series of commands for compiling and linking within this structure follows (information on compilation and linking commands appears in Sections 3.4 and 3.5):

```
$ ! Set up the initial program directory.
$ CREATE/DIRECTORY [PROJ.SERVER]
$ SET DEFAULT [PROJ.SERVER]

$ ! Edit your common source files.
$ EDIT ERROR_HANDLING_.ADA
$ EDIT ERROR_HANDLING.ADA
$ EDIT SQRT_SERVER_.ADA
$ EDIT SQRT_SERVER_TESTER_.ADA
$ ! Create your main program library. It will have
$ ! a default value of SYSTEM.SYSTEM_NAME=VAX_VMS.
$ ACS CREATE LIBRARY [PROJ.SERVER.PROGLIB]

$ ! Set the current library to your main program
$ ! library and initially compile your source files.
$ ACS SET LIBRARY [.PROGLIB]
$ ADA ERROR_HANDLING_,ERROR_HANDLING,SQRT_SERVER_,SQRT_SERVER_TESTER_

$ ! Create subdirectories and sublibraries for your
$ ! VAXELN-specific bodies.
$ CREATE/DIRECTORY [PROJ.SERVER.VAXELN]
$ ACS CREATE SUBLIBRARY/PARENT=[PROJ.SERVER.PROGLIB]-
_$ /SYSTEM_NAME=VAXELN [PROJ.SERVER.VAXELN.SUBLIB]

$ ! Edit and compile your VAXELN-specific bodies.
$ SET DEFAULT [PROJ.SERVER.VAXELN]
$ EDIT SQRT_SERVER.ADA
$ EDIT SQRT_SERVER_TESTER.ADA
$ ACS SET LIBRARY [.SUBLIB]
$ ADA SQRT_SERVER,SQRT_SERVER_TESTER

$ ! Add some code to a common unit (will require
$ ! recompilation from your sublibrary because those
$ ! units that depend on it are now obsolete).
$ EDIT [PROJ.SERVER]ERROR_HANDLING.ADA
$ ACS COMPILE SQRT_SERVER

$ ! Link the VAXELN version of your program;
$ ! The resulting executable image will be in your
$ ! VAXELN subdirectory.
$ ACS LINK SQRT_SERVER

$ ! Build and boot your program as described in
$ ! Chapters 4 and 5.

$ ! Follow a similar procedure for creating, compiling
$ ! and linking your VAX/VMS-related program, except
$ ! use the default version of ACS CREATE SUBLIBRARY
$ ! (/SYSTEM_NAME=VAX_VMS).
```

3.4 Compiling and Recompiling VAXELN Ada Programs

VAXELN Ada programs are compiled in an identical fashion to VAX Ada programs, using the VAX Ada compiler and ACS commands. The DCL-level syntax for these commands is as follows (ACS COMPILE and RECOMPILE can also be issued interactively from within ACS):

```
$ ADA file-spec[,...]

$ ACS COMPILE unit-name[,...]

$ ACS RECOMPILE unit-name[,...]
```

Developing VAX Ada Programs on VAX/VMS gives complete information on these commands. For quick reference, Table 3–3 summarizes the properties of these commands (along with the properties of the related ACS SET SOURCE and SHOW SOURCE commands); Table 3–4 summarizes the optional command qualifiers and applicable qualifier defaults.

Online descriptions of all of these commands and all of their qualifiers are also available using the ACS HELP command.

Table 3–3: Compilation Commands

Command	Function
DCL Commands	
ADA file-spec	Invokes the VAX Ada compiler to compile all compilation units in the given Ada source files. The default mode for this command is interactive. The ADA command must be used for the initial compilation of units.

Table 3–3 (Cont.): Compilation Commands

Command	Function
ACS Commands	
COMPILE unit-name[,...]	Forms the closure of the given compilation units; checks the completeness and currency of the units in the closure; and identifies units that have revised source files (it uses a date-time check of the source files). The COMPILE command then compiles the units that have revised source files and implicitly calls the ACS RECOMPILE command. The default mode for this command is batch.
RECOMPILE unit-name[,...]	Forms the closure of the given compilation units, and checks the completeness and currency of the units in the closure (it uses a date-time check of the units). The RECOMPILE command then recompiles any obsolete units in the appropriate order to make them current. The default mode for this command is batch.
SET SOURCE directory-spec[,...]	Defines a search list of the VAX/VMS directories where the COMPILE command should search for source files.
SHOW SOURCE	Displays the directory search list used by the COMPILE command.

Table 3-4: ADA, COMPILE, and RECOMPILE Qualifiers

Command Qualifier[1]	Default	Applicable Command
/AFTER=time	/AFTER=TODAY	COMPILE RECOMPILE
/[NO]CHECK	/CHECK	COMPILE RECOMPILE
/CLOSURE		COMPILE RECOMPILE
/COMMAND[=file-spec]		COMPILE RECOMPILE
/[NO]CONFIRM	/NOCONFIRM	COMPILE RECOMPILE
/[NO]COPY_SOURCE	/COPY_SOURCE	COMPILE RECOMPILE
/[NO]DEBUG[=(option[,...])]	/DEBUG=ALL	COMPILE RECOMPILE
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS	COMPILE RECOMPILE
/[NO]ERROR_LIMIT[=number]	/ERROR_LIMIT=30	COMPILE RECOMPILE
/[NO]KEEP	/KEEP	COMPILE RECOMPILE
/LIBRARY=directory-spec	/LIBRARY=ADA$LIB	ADA
/[NO]LIST[=file-spec]	/NOLIST	COMPILE RECOMPILE
/[NO]LOG	/NOLOG	COMPILE RECOMPILE
/[NO]MACHINE_CODE	/NOMACHINE_CODE	COMPILE RECOMPILE
/NAME=job-name		COMPILE RECOMPILE

[1]A command qualifier has the same effect, regardless of where it appears in the command string (whether it is appended to the command verb or to a parameter).

Table 3–4 (Cont.): ADA, COMPILE, and RECOMPILE Qualifiers

Command Qualifier[1]	Default	Applicable Command
/[NO]NOTE_SOURCE	/NOTE_SOURCE	COMPILE RECOMPILE
/[NO]NOTIFY	/NOTIFY	COMPILE RECOMPILE
/[NO]OPTIMIZE[=option]	/OPTIMIZE=TIME	COMPILE RECOMPILE
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT	COMPILE RECOMPILE
/[NO]PRINTER[=queue-name]	/NOPRINTER	COMPILE RECOMPILE
/QUEUE=queue-name	/QUEUE=ADA$BATCH[3]	COMPILE RECOMPILE
/[NO]SHOW[=option]	/NOSHOW	COMPILE RECOMPILE
/SUBMIT	/SUBMIT	COMPILE RECOMPILE
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY	COMPILE RECOMPILE
/WAIT	/SUBMIT	COMPILE RECOMPILE
/[NO]WARNINGS[=(option[,...])]		COMPILE RECOMPILE

Positional Qualifier[2]	Default	Applicable Command
/BODY		COMPILE RECOMPILE

[1]A command qualifier has the same effect, regardless of where it appears in the command string (whether it is appended to the command verb or to a parameter).

[2]A positional qualifier has a different effect depending on where it appears in the command string. A positional qualifier appended to the command verb affects the entire command string. A positional qualifier appended to a parameter affects only that parameter.

[3]If ADA$BATCH is not defined as a batch queue, then SYS$BATCH is used.

Table 3–4 (Cont.): ADA, COMPILE, and RECOMPILE Qualifiers

Positional Qualifier[2]	Default	Applicable Command
/[NO]DATE_CHECK	/DATE_CHECK	COMPILE RECOMPILE
/[NO]CHECK	/CHECK	ADA
/[NO]COPY_SOURCE	/COPY_SOURCE	ADA
/[NO]DEBUG[=(option[,...])]	/DEBUG=ALL	ADA
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS	ADA
/[NO]ERROR_LIMIT[=number]	/ERROR_LIMIT=30	ADA
/[NO]LIST[=file-spec]	/NOLIST	ADA
/[NO]MACHINE_CODE	/NOMACHINE_CODE	ADA
/[NO]NOTE_SOURCE	/NOTE_SOURCE	ADA
/[NO]OPTIMIZE[=option]	/OPTIMIZE=TIME	ADA
/[NO]SHOW[=option]	/NOSHOW	ADA
/[NO]SYNTAX_ONLY	/NOSYNTAX_ONLY	ADA
/[NO]WARNINGS[=(option[,...])]		ADA

[2] A positional qualifier has a different effect depending on where it appears in the command string. A positional qualifier appended to the command verb affects the entire command string. A positional qualifier appended to a parameter affects only that parameter.

3.5 Linking VAXELN Ada Programs

Because of the way Ada compilation units and program libraries are organized and managed, VAXELN Ada and VAX Ada do not use the VAX/VMS Linker directly. Instead, you link VAXELN Ada units with the ACS LINK command, which has the following syntax at DCL level (you can also issue this command interactively, within ACS):

```
$ ACS LINK[/qualifiers] unit-name [file-spec[,...]]
```

In particular, if you are linking units with a non-Ada main program, the syntax is

```
$ ACS LINK/NOMAIN unit-name[,...] file-spec[,...]
```

The optional qualifiers and applicable qualifier defaults for the ACS LINK command are summarized in Table 3–5; more detailed information is given in *Developing Ada Programs on VAX/VMS.*

Online descriptions of the ACS LINK command and its qualifiers are also available using the ACS HELP command.

Note in particular the qualifier /SYSTEM_NAME. This qualifier allows you to specify the name of the target on which you intend to run your program. You must specify ACS LINK/SYSTEM_NAME=VAXELN for any units or files that are to be linked as part of a VAXELN application, or you must set the value of SYSTEM.SYSTEM_NAME to VAXELN for your current program library (see Section 3.2). When SYSTEM.SYSTEM_NAME is VAXELN, the Ada units and object files named in the ACS LINK command are linked against the VAXELN-specific Ada run-time libraries and not the VAX/VMS-specific Ada run-time libraries. In other words, when the intended target is VAXELN, the ACS LINK command makes use of the DCL LINK/NOSYSLIB command to prevent searches of the standard VAX/VMS libraries.

A summary description of the ACS LINK command follows the table. Link-related operations of special interest to VAXELN Ada programmers are described in Sections 3.5.1 and 3.5.2.

Table 3–5: ACS LINK Command Qualifiers

Command Qualifier[1]	Default
/AFTER=time	/AFTER=TODAY
/BRIEF	
/COMMAND[=file-spec]	
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=file-spec]	/NODEBUG
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE
/FULL	
/[NO]KEEP	/KEEP
/[NO]LOG	/NOLOG
/[NO]MAIN	/MAIN

[1]A command qualifier has the same effect, regardless of where it appears in the command string (whether it is appended to the command verb or to a parameter).

Table 3–5 (Cont.): ACS LINK Command Qualifiers

Command Qualifier[1]	Default
/[NO]MAP[=file-spec]	/NOMAP
/NAME=job-name	
/[NO]NOTIFY	/NOTIFY
/OBJECT=file-spec	
/OUTPUT=file-spec	/OUTPUT=SYS$OUTPUT
/[NO]PRINTER[=queue-name]	/NOPRINTER
/QUEUE=queue-name	/QUEUE=SYS$BATCH
/SUBMIT	/WAIT
/[NO]SYSLIB	/SYSLIB
/[NO]SYSSHR	/SYSSHR
/SYSTEM_NAME=system	/SYSTEM_NAME=current-value
/[NO]TRACEBACK	/TRACEBACK
/WAIT	/WAIT

Parameter Qualifier[2]	Default
/LIBRARY	
/INCLUDE	
/OPTIONS	
/SHAREABLE	

[1]A command qualifier has the same effect, regardless of where it appears in the command string (whether it is appended to the command verb or to a parameter).

[2]A parameter qualifier can be used only with a specified parameter. It cannot be appended to the command verb.

The ACS LINK command goes through the following steps:

1. If the LINK/MAIN command (the default) is specified, checks that only one unit is specified and that the unit is an Ada main program.
2. Forms the closure of the main program (LINK/MAIN) or of the specified units (LINK/NOMAIN) and verifies that all units in the closure are present and current. If ACS detects an error, the operation is terminated before the VAX/VMS Linker is invoked.

3. Creates a DCL command file for the VAX/VMS Linker. The command file is deleted after the ACS LINK operation is terminated or aborted, unless the LINK/COMMAND command is specified. If the LINK /COMMAND command is specified, the command file is retained for future use, and the linker is not invoked.
4. Creates an object file (to be linked with the program) that elaborates the library units in proper order at run time. If the /MAIN qualifier is specified, the object file also contains the image transfer address. This object file is deleted after the ACS LINK operation is terminated or aborted, unless the LINK/COMMAND command is specified. If the LINK/COMMAND command is specified, the object file is retained and the linker is not invoked.
5. Unless the /COMMAND qualifier is specified, invokes the VAX/VMS Linker as follows:
   a. By default (LINK/WAIT), the linker command file generated in step 3 is executed in a subprocess. You must wait for the link operation to terminate before issuing another command.
   b. If you use the LINK/SUBMIT command, the linker command file is submitted as a batch job.

   Note that any logical names needed by subprocesses doing linking (or compilation) should at least be in the VAX/VMS job logical name table, not in the process logical name table.

ACS output originating before the VAX/VMS Linker is invoked is reported to your terminal by default, or to a file specified with the /OUTPUT qualifier. Linker diagnostics are reported to your terminal by default, or to a log file if the ACS LINK command is executed in batch mode (ACS LINK /SUBMIT). The VAX/VMS Linker is described in detail in the *VAX/VMS Linker Reference Manual.*

3.5.1 Including Object Modules from Other Libraries

The ACS LINK command can be used with the /LIBRARY, /INCLUDE, /OPTIONS, and /SHAREABLE qualifiers to allow you to link VAXELN or VAX Ada units against object library, shareable image, or options files.

The LINK/INCLUDE combination is of particular interest in writing VAXELN Ada programs, because it allows you to link specific object files, such as error-message text, from an object-module or shareable-image library with a main program. By default, error-message text is not included with a VAXELN system. Each facility—KER$, ELN$, ADA$, and so on—supplies a module in ELN$:RTL.OLB that has the message text for that facility. The module name has the following form:

```
facility-nameMSGDEF_TEXT
```

You must specifically include any error-message text when you link your main program if you want the messages to be part of the finished VAXELN system.

For example, the square-root server program (SQRT_SERVER) in Chapter 2 depends on a package ERROR_HANDLING, which contains a subprogram that displays VAXELN kernel error messages. To link SQRT_SERVER—and thus ERROR_HANDLING, because it is in the closure of the main program SQRT_SERVER—with the appropriate error message text, you would use the following command:

```
$ ACS LINK SQRT_SERVER ELN$:RTL/INCLUDE=(KER$MSGDEF_TEXT)
```

This command causes the modules KER$MSGDEF_TEXT from the object-module library ELN$:RTL.OLB to be linked with the Ada main program SQRT_SERVER. (The default file type for the external library is .OLB.)

Note that you do not need to include message text if you are running your system under the control of the remote debugger. Instead of message text, you will receive a hexadecimal message number, which you can then identify using the debugger command EVALUATE/CONDITION_VALUE. See Chapter 6 for more information on the remote debugger and general debugger commands.

3.5.2 Using the ACS EXPORT Command

The ACS EXPORT command lets you export library units from your program library so that you can link them with foreign-language (non-Ada) code. You can use the ACS EXPORT command to export units that are to be used as part of a foreign-language main program; you can also use the ACS EXPORT command to export a main (or entire) Ada program. You can also use the ACS EXPORT command to build a shareable image that consists of object modules exported from a program library. (This is the only way to build a shareable image from units in a program library.)

The command has the following syntax (DCL version shown):

```
$ ACS EXPORT unit-name[,...]
```

and its result is a concatenated object file (unit-name.OBJ) in your default directory. The object file contains the generated code for all units in the closure of the specified units.

The ACS EXPORT command takes several optional qualifiers. The three most significant are

- /OBJECT=file-spec, to provide an alternative file specification (and directory) for the generated object file.
- /[NO]MAIN, to indicate whether or not you are exporting a main program. The /NOMAIN qualifier (the default) allows you to specify more than one unit name with the EXPORT command, and the concatenated object file for the exported units includes no image transfer address (indicating a main program). The /MAIN qualifier allows you to name only one unit with the EXPORT command, and the object file for the exported unit includes the image transfer address.
- /SYSTEM_NAME, to determine the target (VAXELN or VAX_VMS) for which the specified units are to be linked. The target choices are VAXELN and VAX_VMS; the default is VAX_VMS when you create your program library or sublibrary (see Section 3.2).

For a complete list of qualifiers, see *Developing Ada Programs on VAX/VMS*.

Note that object files created by different invocations of the EXPORT command may include some code that is common—for example, if each closure includes the predefined unit TEXT_IO. In such cases, you will not be able to link those files into the same image. Whenever you think that the closures may include units in common, you should specify all the units in a single EXPORT command line. To export a set of units contained in more than one program library, create a temporary program library and enter the applicable units into that library using the ENTER UNIT command. Then, export the units from the temporary library using a single EXPORT command, and delete the library using the DELETE LIBRARY command.

Also note that the object file produced by the ACS EXPORT command is target specific. In other words, the link will fail if you export units when the value of SYSTEM.SYSTEM_NAME is VAX_VMS, and then try to link them with the VAXELN run-time libraries. Conversely, the link will fail if you export units when the value of SYSTEM.SYSTEM_NAME is VAXELN, and then try to link them with the VAX/VMS run-time libraries. See Section 3.2 for more information on SYSTEM.SYSTEM_NAME; see Section 3.5 for more information on the ACS LINK command.

Also note that when you export VAXELN Ada units that are to be used by a foreign-language main program, the ACS EXPORT command does not arrange for an automatic call of the initialization routine needed to elaborate any library packages used by the exported units. In other words, if the ACS EXPORT command is executed with an explicit or implicit

/NOMAIN qualifier and an explicit or implicit value of VAXELN for SYSTEM.SYSTEM_NAME, you must call the initialization routine from the non-Ada main program.

You call the initialization routine in the following manner. The name of the routine is ADA$ELAB_*unit*, where *unit* is the name of the first unit specified with the ACS EXPORT command. The routine has no arguments, and it must be called by the main program before calls are made or data is accessed from any of the exported units.

For example, if you wanted to export an Ada subprogram to be called by a VAXELN Pascal main program, you would declare the elaboration procedure and call it from the Pascal main program as follows:

```
MODULE main_prog;

PROCEDURE swap (VAR swap1,swap2 : INTEGER);
   EXTERNAL;
PROCEDURE ADA$ELAB_SWAP;
   EXTERNAL;

PROGRAM use_swap (INPUT,OUTPUT);
   VAR
      x,y: INTEGER;
   BEGIN
      ADA$ELAB_SWAP;
      (* Give x and y values *)
      x := 7;
      y := 5;
      WRITELN(x,y);
      swap(x,y);
      WRITELN(x,y);
   END;
END;
-----------------------------------------------------------------
procedure SWAP (A,B: in out INTEGER) is
   TEMP: INTEGER;
begin
   TEMP := A;
   A := B;
   B := TEMP;
end SWAP;
pragma EXPORT_PROCEDURE (SWAP);
```

Alternatively, for the following command

```
  ACS EXPORT/NOMAIN A,B,C
```

the main (foreign) program needs to call the procedure ADA$ELAB_A to execute the elaboration code units for A, B, and C before it calls anything or uses any data in A, B, or C.

Note that any exported Ada unit that is not a main Ada program or is not part of an exported main program must also use the export pragmas described in Chapter 13 of the *VAX Ada Language Reference Manual.*

Also note that when you link exported VAXELN Ada units (either main programs or individual units) from your VAX/VMS host system, you must specify the necessary VAXELN run-time libraries as follows (the ACS LINK command does this automatically; the DCL LINK command does not):

```
$ LINK/NOSYSLIB object-module[,...],ELN$:RTLSHARE/LIB,ELN$:RTL/LIB
```

Thus, to use the DCL LINK command to link the swap program previously shown, you would use

```
$ LINK/NOSYSLIB main_prog,swap,ELN$:RTLSHARE/LIB,ELN$:RTL/LIB
```

Chapter 4

Building a System

After you have compiled and linked your VAXELN Ada program, you build the resulting image into a system using the VAXELN System Builder. System building involves combining VAXELN Ada linked images (jobs) with other images, servers, device drivers, and the VAXELN kernel executive to form a bootable system image. The System Builder is a component of VAXELN, and provides a convenient set of menus for describing and building a VAXELN system.

The description of the System Builder in this chapter is designed to give you essential information on the EBUILD command and on the System Builder menus. More complete information is presented in the *VAXELN User's Guide*.

4.1 The EBUILD Command

The EBUILD command (plus any optional qualifiers) invokes the VAXELN System Builder with a data file that specifies the images that are to be combined into a VAXELN system. The command has the following syntax:

```
$ EBUILD [qualifier-list] data-file-specification
```

The result (when the system is built) is a file with a default file type of .SYS, and, unless otherwise specified with the /SYSTEM qualifier, the same file name as the data file. If you omit the required data file specification, you receive a prompt for a file name; the default file type for the data file is .DAT.

Table 4–1 summarized the qualifiers to the EBUILD command.

Table 4–1: EBUILD Command Qualifiers

Qualifiers	Description	Default
/[NO]BRIEF	Used with the /MAP qualifier to control the contents of a system map listing. The /BRIEF qualifier causes all the images, devices, and terminals, as well as all of the system characteristics, to be listed. The /NOBRIEF qualifier (same as the /FULL qualifier) causes all the images in the system, all program descriptions, all device descriptions, all terminal descriptions, and all system characteristics to be listed.	/BRIEF
/[NO]EDIT	Determines whether or not an interactive screen-editing mode is entered for creating or altering the data file specified with the EBUILD command. The /EDIT qualifier causes the System Builder to enter a data-file editor that is compatible with VT100- and VT200-series terminals. The /NOEDIT qualifier causes the System Builder to build a system image directly from the contents of the data file specified with the EBUILD command.	/EDIT

Table 4–1 (Cont.): EBUILD Command Qualifiers

Qualifiers	Description	Default
/[NO]FULL	Used with the /MAP qualifier to control the contents of the system map. The /FULL qualifier causes all the images in the system, including all program descriptions, all device descriptions, all terminal descriptions, and all system characteristics, to be listed. The /NOFULL qualifier (same as the /BRIEF qualifier) causes all images, devices, and terminals, as well as the system characteristics, to be listed.	/NOFULL
/KERNEL=file-spec	Specifies the name of a kernel image other than the default kernel image, ELN$:KERNEL.EXE. This feature is useful only for special applications in which the kernel is being debugged.	
/[NO]LOG	Specifies whether or not the System Builder displays the size of the finished system image.	/LOG
/[NO]MAP[=file-spec]	Enables or inhibits the production of a system map listing. If the /MAP qualifier is used without its optional file specification, the listing has the same name as the specified data file, with a file type of .MAP. The contents of the map listing are controlled by the /BRIEF and /FULL qualifiers, which are mutually exclusive.	/NOMAP
/SYSTEM=file-spec	Specifies an alternate file name for the file to which the system image is written. By default, the system image file has the same name as the specified data file, with a file type of .SYS.	

4.2 Using the System Builder Menus

The easiest way to create or alter the data file required by the System Builder is to work with the menus that appear when you execute the EBUILD command in EDIT mode (the default mode for this command). Each menu supplies a set of defaults for each of its options. You either use the existing defaults, or you can edit the default values to suit your needs.

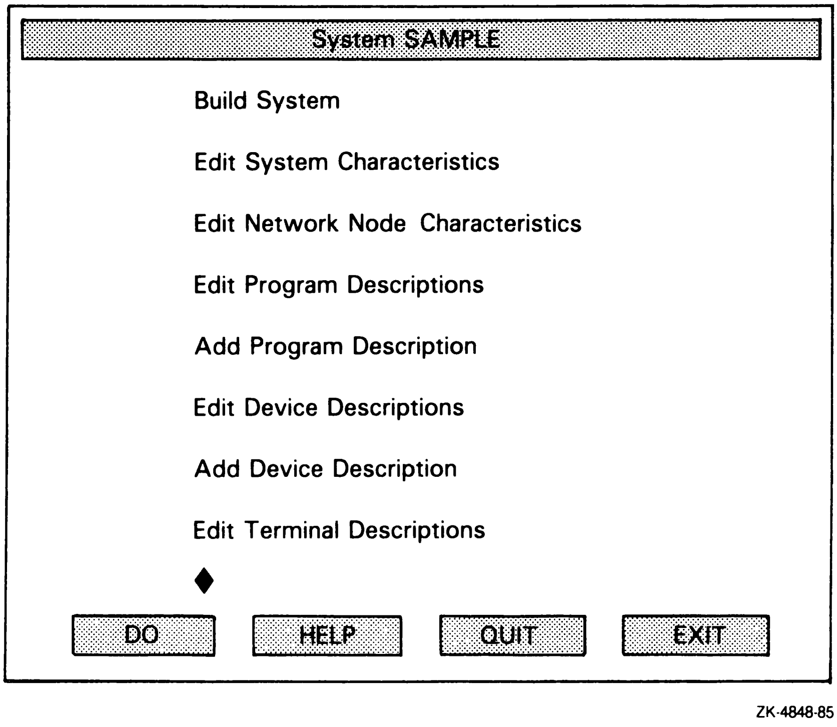
When you invoke the System Builder in EDIT mode, you receive a Main menu that lists the following choices:

- *Build System*—uses the EBUILD data file to combine programs with the VAXELN kernel and run-time software to create a bootable system image.
- *Edit System Characteristics*—presents a set of menu options that describe particular system characteristics.
- *Edit Network Node Characteristics*—presents a set of menu options that describe particular network and network node characteristics.
- *Edit Program Descriptions*—lists the existing program descriptions as choices for editing.
- *Add Program Description*—presents a set of menu options for an individual program. If the program has never been characterized, you can use this menu to add the program to the set of program descriptions.
- *Edit Device Descriptions*—lists the existing device descriptions as choices for editing.
- *Add Device Description*—presents a set of menu options for an individual device. If the device has never been characterized, you can use this menu to add the device to the set of device descriptions.
- *Edit Terminal Descriptions*—lists the terminals (except for the console terminal) that have been characterized.
- *Add Terminal Description*—presents a set of menu options for an individual terminal. If the terminal has never been characterized, you can use this menu to add the terminal to the set of terminal descriptions.
- *Edit Console Characteristics*—presents a set of menu options that describe particular console terminal characteristics.

All menu editing functions are controlled with the PF1, PF2, PF3, and PF4 keys; the four arrow keys; and the control sequences CTRL/E, CTRL/H, CTRL/R, and CTRL/U.

The four PF keys correspond, left to right, with four legends at the bottom of each menu. For reference, a sample menu (in this case, the Main menu) is shown in Figure 4–1.

Figure 4–1: Sample System Builder Menu



On all menus, the PF1 key corresponds to the DO function. When you press the PF1 key, you activate the currently selected entry or add a set of edited characteristics to the system. Similarly, on all menus, the PF2 key corresponds to the HELP function. The definitions of the PF3 and PF4 keys vary, depending on the activity represented by the menu:

- On the Main menu, the PF3 key corresponds to the QUIT function, which aborts the System Builder session without altering the data file. The QUIT function requires confirmation with the DO function (bound to the PF1 key).
- On the Main menu, the PF4 key corresponds to the EXIT function, which ends the System Builder session but incorporates any changes.
- On the Edit Program Descriptions, Edit Device Descriptions, and Edit Terminal Descriptions menus, the PF3 key corresponds to the DELETE function, which deletes the current set of descriptions from the data file. The DELETE function requires confirmation with the DO function (bound to the PF1 key).
- On all menus except the Main menu, the PF4 key corresponds to the BACK function, which returns you to the previous menu, without incorporating any edits.

NOTE

The VT200-series keyboard keys, F15 and F16, usually labeled HELP and DO, are not recognized by the System Builder.

The arrow keys move up, down, left, and right among menu choices.

Four control sequences are predefined for working with input text (you cannot use them for manipulating menu choices):

- CTRL/E moves to the end of a line of text.
- CTRL/H (backspace) moves to the beginning of a line of text.
- CTRL/R refreshes the screen.
- CTRL/U deletes text from the cursor back to the beginning of the current line.

CTRL/E, CTRL/H, and CTRL/U are particularly useful for editing long strings of input text.

Note that the display for input text scrolls to the left of the column allocated for that value if you enter more information than will fit on the screen (such as when you enter information for the *Arguments* option of the Program Characteristics menus). A diamond symbol appears at the top, bottom, or edge of a menu when text—either text that you have entered or menu options—is off the screen.

4.2.1 Build System

When chosen from the Main menu and activated with the DO function (bound to the PF1 key), the Build System option causes the System Builder to create a new system image file using the data file specified with the EBUILD command. You generally make this choice when you have finished specifying your system with the System Builder menus.

By default, the image file has the same file name as the data file and a default file type of .SYS. By using the /SYSTEM qualifier on the EBUILD command, you can explicitly control the naming of the system image file. (See Section 4.1 and Table 4–1.)

This menu choice is comparable to the EBUILD/NOEDIT command.

4.2.2 System Characteristics

The system characteristics define the general properties of the VAXELN system you are building. The Edit System Characteristics Menu options and their default values are presented in Table 4–2.

Table 4–2: Edit System Characteristics Menu

Menu Option	Description	Default
System image	Specifies the name of the system image file to be created by the System Builder.	data-file-spec.SYS
Debug Local Remote Both None	Specifies which debuggers are to be built into the system. *Local* means that the EDEBUGLCL debugger image is included. *Remote* means that the EDEBUGREM debugger image is included. *Both* means that both EDEBUGDCL and EDEBUGREM are included: in that case, EDEBUGREM is the primary debugger, and EDEBUGLCL gets control only in the event of a system error. Note that the VAXELN Remote Debugger communicates with EDEBUGREM, so that in order to perform remote debugging you must specify either *Remote* or *Both*.	Remote

Table 4–2 (Cont.): Edit System Characteristics Menu

Menu Option	Description	Default
Console Yes No	Specifies that communication with the console terminal on the target machine is desired. *Yes* means that a VAXELN Console Driver and the device description for the console terminal are included automatically when the system is built. Note that when the *Debug* option is *Local* or *Both,* then the driver and device description are included implicitly, independent of the value of the Console option. If you select *No* and the *Debug* option is *Remote,* the remote VAX/VMS terminal behaves as the console terminal while you are using the remote debugger. The name for the system's console terminal is CONSOLE:.	Yes
Instruction emulation String Float Both None	Selects emulation software for instructions that are present in the full VAX architecture, but that are not included in the MicroVAX architecture. *None* indicates a full VAX target. *Float* includes emulation software for the floating-point instructions. *String* includes emulation for the other instructions in the subset. *Both* includes both floating-point and string emulation software. On a MicroVAX I, *Both* is recommended; on a MicroVAX II, *String* is sufficient, unless H_floating types (like LONG_LONG_FLOAT) are involved.	String

Table 4–2 (Cont.): Edit System Characteristics Menu

Menu Option	Description	Default
Boot method Disk ROM Downline	Selects the method by which the finished system will be booted on the target machine, and determines the type of image header used in the system: Disk and tape—No header ROM—MicroVAX ROM header Downline—VAX/VMS image header If you select *Disk* or *ROM* but did not specify a node address in the Edit Network Node Characteristics menu, a warning message is issued.	Downline
Disk/volume names	Supplies the device specifications and volume names for disks present on the target machine in the following format: "device-specification volume_name". Programs can then refer to a volume by prefixing the given name with DISK$. For example, "DUA0 TEST" can be referred to as DISK$TEST. Multiple names must be separated by commas; the first specification in the list identifies the default disk volume. When the system is bootstrapped, the VAXELN File Service automatically mounts the indicated volumes. Because the volume name is optional, the file service will attempt to mount whichever disk is present in the indicated drive if the volume name is omitted from this menu.	
Guaranteed image list	Lists the 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.	

Table 4–2 (Cont.): Edit System Characteristics Menu

Menu Option	Description	Default
Page table slots	Specifies 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 of 64 thus accommodates a system with 32 simultaneous jobs (if they do no multitasking). The minimum number of page table slots is 2; the maximum is 32,767.	64
Ports	Specifies 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.	256
Pool size	Specifies the approximate number of VAXELN system objects that can be in simultaneous use (Ada types for these objects are predefined in package VAXELN_SERVICES). One pool block (128 bytes) is needed for each system object in use, processes require a total of 3, and a few additional blocks are needed for each job. Essentially, you can use one block per system object plus three times the number of jobs and processes in simultaneous use. The minimum number of blocks is 16; the maximum is 32,764.	384 blocks

Table 4–2 (Cont.): Edit System Characteristics Menu

Menu Option	Description	Default
Virtual size	Specifies the maximum size, in 512-byte pages, of each P0 and P1 region in the system. The value is used by the VAXELN 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 minimum number of pages is 128; the maximum is 32,640.	1024 blocks
Interrupt stack	Specifies the maximum number of pages required for the system interrupt stack. The minimum number of pages is 2; the maximum number is 8,192.	2 pages
I/O region size	Specifies the maximum number of 512-byte pages required by all interrupt service communication regions. The value is used by the VAXELN kernel to allocate system page table entries during the startup of the system. The minimum number of pages is 0; the maximum is 32,767.	128 pages
Dynamic program space	Specifies the number of 512-byte memory pages that can be allocated for dynamically loading programs into the running system (the pages are not actually allocated until they are needed). The minimum number of pages is 0; the maximum is 32,767.	0 pages

Table 4–2 (Cont.): Edit System Characteristics Menu

Menu Option	Description	Default
Time interval	Specifies the interval (in microseconds) between interval-timer interrupts. In other words, this option specifies 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 minimum time is 1 microsecond; the maximum time is 120,000,000 microseconds (2 minutes). Note that on some processors (including the MicroVAX), the default of 10,000 microseconds cannot be altered.	10000 micro-seconds
Connect time	Specifies the time (in seconds) that is allowed to elapse before a VAXELN circuit connection must be accepted. The minimum time is 1 second; the maximum is 3,599 seconds (59 minutes, 59 seconds).	45 seconds
Memory limit	Specifies the maximum amount of physical memory (in 512-byte pages) that is available for use by the system. A value of 0 means that the system should use all the memory available on the target configuration. Thus, a limit needs to be specified only for special applications. The minimum is 0 pages (no limit); the maximum is 65,535 pages.	0 pages

4.2.3 Network Node Characteristics

The Edit Network Node Characteristics menu defines characteristics for the VAXELN Network Service and Authorization Service. The menu options and default values are presented in Table 4–3.

Table 4–3: Edit Network Node Characteristics Menu

Menu Option	Description	Default
Network service Yes No	Determines whether or not the VAXELN Network Service is included automatically.	Yes
Name server Yes No	Determines whether or not the VAXELN Network Service running on this target machine can volunteer to be the name service in network applications. If *Yes* is specified, then a value of *Yes* must also be specified for the *Network service* option.	Yes
File access listener Yes No	Determines whether or not the VAXELN File Access Listener is included automatically. The System Builder issues an informational message if you include the File Access Listener without also specifying *Yes* for the *Network service* option.	Yes
Network device UNA QNA Other	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* is the QBUS-to-Ethernet adapter (DEQNA). The necessary device driver program and device description are included automatically if you also specify *Yes* for the *Network service* option.	QNA
Node name	Specifies the node name by which a VAXELN node is identified in a network. It can have a maximum of 6 characters, and must be unique in the network. You do not need to specify the node name if your system will be downline loaded from your development system.	

Table 4–3 (Cont.): Edit Network Node Characteristics Menu

Menu Option	Description	Default
Node address	Specifies the address for a VAXELN node in a network. The address can have one of three forms: nnn is a DECnet node number, aaa.nnn is a DECnet area and node number, and nn-nn-nn-nn-nn-nn is a 48-bit Ethernet address (where a and n are digits). You do not need to specify the node address if your system will be downline loaded from your development system.	0
Authorization required Yes No	Determines whether or not the VAXELN Network Service can authorize inbound circuit connections by communication with the VAXELN Authorization Service.	No
Authorization service Local Network None	Determines whether or not an Authorization Server is included in the system, and, if it is included, whether the service should serve only the local node or the entire local area network. Some nodes can have local services of their own, but there should be only one network Authorization Server.	None
Authorization file	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 it must exist on a node that the service is authorized to access.	AUTHORIZE.D
Default UIC	Specifies the default user identification code (UIC) for users that are not explicitly authorized.	[1,1]
Node triggerable Yes No	Specifies whether or not downline load triggers are enabled. *Yes* means that the system will allow itself to be remotely triggered, and should be the setting during development, so that developers can remotely load the system.	Yes

Table 4–3 (Cont.): Edit Network Node Characteristics Menu

Menu Option	Description	Default
Network segment size	Specifies the size (in bytes) of the largest segment that can be sent over the network. 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 seent size minus 32. The segment size has a minimum value of 192 bytes; the maximum value is 1,470.	576 bytes

4.2.4 Program Descriptions

Each image in the system (except the kernel, debugger, device drivers, shareable run-time library images, and network and authorization service) has a program description. The Edit Program Descriptions menu lists all of the program descriptions characterized in the data file; the Add Program Description menu presents a menu of options for the individual program chosen from the Edit Program Descriptions menu, or for a new program that is to be added to the system. The options and default values for the Add Program Description menu are presented in Table 4–4.

Note that the Edit Program Descriptions menu is empty if there are no program descriptions yet in the system. When you select the Edit Program Descriptions menu and it is empty, you can use the DO function (bound to the PF1 key) to switch directly to the Add Program Description menu without returning to the Main menu.

Also note that some program descriptions are added for you automatically by the System Builder. For example, if you specify *Yes* for the *File Access Listener* option on the Edit Network Node Characteristics menu, the File Access Listener's image and program description are automatically added. Most device drivers are also added automatically. (See the *VAXELN User's Guide* for more information.)

Table 4–4: Add Program Description Menu

Menu Option	Description	Default
Program	Specifies the program image name (without the file type, device, or directory).	
Debug Yes No	Determines whether or not any job that runs this program gives control to the debugger instead of executing immediately.	No
Run Yes No	Determines whether or not a job running the program image is started automatically when the system itself is started.	Yes
Init required Yes No	Determines whether or not an initialization procedure is done for the program. More specifically, *Yes* means that the program is run automatically and will run to completion before any other program starts unless it calls the VAXELN_SERVICES.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.)	No
Mode Kernel User	Determines the mode in which the program is run. Kernel mode is required for device drivers (programs calling CREATE_DEVICE), as well as for programs using the VAXELN_SERVICES.ALLOCATE_MAP, SYSTEM.MFPR, SYSTEM.MTPR, and VAXELN_SERVICES.FREE_MAP subprograms. User mode is recommended unless a program definitely requires kernel mode.	User

Table 4–4 (Cont.): Add Program Description Menu

Menu Option	Description	Default
User stack (initial)	Determines the initial stack size of the user stack (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 the job). The minimum size is 1 page; the maximum is 32,767 pages. (Note that the user stack is not used by VAXELN Ada, so the default is sufficient for VAXELN Ada programs.)	1 page
Kernel stack	Determines the actual stack size of your kernel stack (the kernel stack is used by all programs for kernel procedure calls, and by kernel-mode programs for all execution; it is fixed in size and is thus not automatically extended during execution of the program). Most kernel-mode programs require a larger kernel stack than the default of 1 page. The minimum size is 1 page; the maximum is 32,767 pages.	1 page
Job priority	Determines the priority of the job in which your program is executing (0 is the highest priority; 31 is the lowest).	16
Process priority	Determines the initial priority of the master process and any subprocesses it creates (0 is the highest; 15 is the lowest). (Note that this value is ignored for all VAXELN Ada tasks, including the main task. Also note that the priority of VAXELN Ada tasks is calculated as 15–P, where P is the process priority.)	8
Job port message limit	Determines the maximum number of messages that can reside at one time in the job port. The minimum number of messages is 0; the maximum is 16,384.	16,384 messages

Table 4–4 (Cont.): Add Program Description Menu

Menu Option	Description	Default
Powerfailure exception Yes No	Determines whether or not the program receives an exception (VAXELN_SERVICES.KER_POWER_SIGNAL) when the processor restarts after a power failure. You should select this option only if the program has an exception handler established for it; such a handler allows you to take general, system-wide action when power has failed. 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 menu option.	No
Argument(s)	Specify additional information needed for building the program, for example, device names or file specifications. Arguments must be strings, and, if they have embedded spaces, must be enclosed in double quotation marks (not apostrophes). Multiple arguments must be separated by commas.	

4.2.5 Device Descriptions

Each device that is part of the target machine's hardware configuration should be specified and built into the system. The Edit Device Descriptions menu lists all of the device descriptions currently characterized in the data file; the Add Device Description menu presents a menu of options for the individual device chosen from the Edit Device Descriptions menu, or for a new device that is to be added to the system. The options and default values for the Add Device Description menu are presented in Table 4–5.

Note that the Edit Device Descriptions menu is empty if there are no device descriptions yet in the system. When you select the Edit Device Descriptions menu and it is empty, you can use the DO function (bound to the PF1 key) to switch directly to the Add Device Description menu without returning to the Main menu.

Also note that no device description is needed for the target machine's console terminal or Ethernet adapter; these descriptions are provided for you automatically when you select the corresponding options on the Edit System Characteristics menu.

Table 4–5: Add Device Description Menu

Menu Option	Description	Default
Name	Describes a device controller. For terminals, individual lines are described by terminal descriptions and named with the controller name and a line number (for example TTA1). Note that the device name must be used consistently across various contexts; however, the actual choice of the device name is up to you.	
Register address	Gives the physical 18-bit address of the device's first device control register (18-bit values are also used for the QBUS). Valid values range from %O000000 to %O777777, and can be specified in decimal, octal (%O), or hexadecimal (%X). To configure a device on the second UNIBUS adapter of a VAX–11/750 target system, you must prefix the 18-bit register address with a one bit, implying UNIBUS adapter 1, rather than the default adapter 0. For example, for a DLV11-type device on the first adapter, you might specify a CSR (control status register) address of %O776500, but on the second you would specify %O1776500.	%O000000
Vector address	Gives the address of the device's first interrupt vector. Valid values range from %O000 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).	%O000

Table 4–5 (Cont.): Add Device Description Menu

Menu Option	Description	Default
Interrupt priority	Determines the device's bus-request priority, from 4 (high) to 7 (low). These values correspond to the VAX interrupt priority levels 14 (hex) to 17 (hex), respectively.	5
Autoload driver Yes No	Determines whether or not the appropriate device driver image is included automatically in the system. For a detailed description of this option, see the *VAXELN User's Guide*.	Yes

You specify the control status register (CSR) addresses, interrupt vector addresses, and priorities for bus devices exactly as they are described in the appropriate device hardware manual or in the *Microcomputer Products Handbook*.

You can also use the VAX/VMS System Generation Utility (SYSGEN) CONFIGURE command to calculate UNIBUS and QBUS CSR and vector addresses if you know the device name. For example, to determine the CSR and vector addresses for two MSCP disk controllers, an Ethernet controller, and a TK50, you would execute the following series of commands:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> CONFIGURE
DEVICE> UDA,2
DEVICE> QNA
DEVICE> TU81
DEVICE> DHV11
CTRL/Z
```

The resulting output (when you press CTRL/Z) would be

```
Device: UDA      Name: PUA      CSR: 772150     Vector: 154     Support: yes
Device: TU81     Name: PTA      CSR: 774500     Vector: 260     Support: yes
Device: QNA      Name: XQA      CSR: 774440     Vector: 120     Support: yes
Device: UDA      Name: PUB      CSR: 760334*    Vector: 300*    Support: yes
Device: DHV11    Name: TXA      CSR: 760500*    Vector: 310*    Support: yes
```

A complete discussion of the SYSGEN Utility and a list of the devices that SYSGEN configures automatically are presented in the *VAX/VMS System Generation Utility Reference Manual*.

Table 4–6 lists common QBUS device names (for MicroVAX systems) and their SYSGEN equivalents. Table 4–7 lists common UNIBUS device names (for VAX–11/730 and VAX–11/750 systems). In both tables, where device names are grouped and indented, the main device name is the name of the controller, and the indented device names are common devices that can be attached to that controller. Names spelled with a lowercase n or x represent a family of individual devices (the n or x designates an appropriate integer).

Table 4–6: Common QBUS Device Names and Their SYSGEN Equivalents

Device	SYSGEN Name
RQDXn	UDA
RD5x	
RX50	
KDA50	UDA
RA6x	
RA8x	
RC25	UDA
TK50	TU81
DEQNA	QNA
DZV11	DZ11
DZQ11	DZ11
DHV11	DHV11
LPV11	LP11

Table 4–7: Common UNIBUS Device Names and Their SYSGEN Equivalents

Device	SYSGEN Name
DMF-32	DMF32
RB730	RB730
R80	
RL02	

Table 4–7 (Cont.): Common UNIBUS Device Names and Their SYSGEN Equivalents

Device	SYSGEN Name
UDA50	UDA
RA6x	
RA8x	
TU81	TU81
DEUNA	UNA
TU58	TU58[1]

[1]If the console TU58 is on a VAX–11/730 or VAX–11/750, leave the register address blank and use %O360 for the vector.

Note that devices with multiple interrupt vectors require only one device description; the other vectors are obtained from the procedure parameters of VAXELN_SERVICES.CREATE_DEVICE.

4.2.6 Terminal Descriptions

Each terminal connected to an asynchronous serial controller line should be described and built in your system, and the terminal descriptions must include the characteristics of the asynchronous controller (for example, the DMF-32 or DZV11). The Edit Terminal Descriptions menu lists all of the program descriptions characterized in the data file; the Add Terminal Description menu presents a menu of options for the individual terminal chosen from the Edit Terminal Descriptions menu, or for a new terminal that is to be added to the system. The options and default values for the Add Terminal Description menu are presented in Table 4–8.

Note that the Edit Terminal Descriptions menu is empty if there are no terminal descriptions yet in the system. When you select the Edit Terminal Descriptions menu and it is empty, you can use the DO function (bound to the PF1 key) to switch directly to the Add Terminal Description menu without returning to the Main menu.

Also note that the console terminal has a separate menu, Edit Console Characteristics (see Section 4.2.7).

Table 4–8: Add Terminal Description Menu

Menu Option	Description	Default
Terminal	Specifies the terminal name, which is the controller device name suffixed with a unit number. For example, terminal TTA0 is the first terminal line on controller TTA.	
Terminal controller type DMF DZ DH	Specifies the type of controller used for terminals. *DMF* applies to asynchronous lines on a DMF-32 controller. *DZ* designates the DZV11 or DZQ11 interface for the MicroVAX. *DH* designates 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 that designates a terminal in programs.	DZ
Speed	Specifies the baud rate that applies to the individual line, as well as specifying the speed for both input and output. Possible values are 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200, 38400.	9600
Parity Yes No	Determines whether or not parity checking is enabled on this line.	No
Parity type Odd Even	Determines the kind of parity checking (if checking is enabled).	Even
Display type Scope Hardcopy	Specifies the kind of output display. *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 terminal that prints on paper rather than displaying output on a screen. The setting of this option is ignored on a DDCMP-specified line.	Scope

Table 4-8 (Cont.): Add Terminal Description Menu

Menu Option	Description	Default
Escape recognition Yes No	Determines whether or not the terminal driver program checks the format of escape sequences to see if they conform to ANSI format. The setting of this option is ignored on a DDCMP-specified line.	Yes
Echo Yes No	Determines whether or not input characters are echoed on the terminal. The setting of this option is ignored on a DDCMP-specified line.	Yes
Pass all Yes No	Determines whether or not all control characters are passed to the user's program as ordinary input, instead of being interpreted by the driver program. The setting of this option is ignored on a DDCMP-specified line.	No
Eight-bit Yes No	Determines whether or not the attached terminal uses 8-bit ASCII characters. The setting of this option is ignored on a DDCMP-specified line.	No
Modem Yes No	Indicates whether or not a modem is attached to the line. Modems are supported only on the DHV11 and DMF-32 controllers.	No
DDCMP Yes No	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* means that the line is a regular terminal line.	No

4.2.7 Console Characteristics

The console terminal on the target machine must be characterized separately, and a separate menu—Edit Console Characteristics—exists for that purpose. The menu options and their default values are presented in Table 4–9.

Table 4–9: Edit Console Characteristics Menu

Menu Option	Description	Default
Display type Scope Hardcopy	Identifies the kind of display. *Scope* means that the attached console terminal is a CRT terminal. *Hardcopy* means that the console terminal displays its output on paper.	Hardcopy
Escape recognition Yes No	Determines whether or not the console terminal driver program checks the format of escape sequences to see whether they conform to ANSI format.	Yes
Echo Yes No	Determines whether or not input characters are echoed on the terminal.	Yes
Pass all Yes No	Determines whether or not all control characters are passed to the user's program as ordinary input, instead of being interpreted by the driver program.	No
Eight-bit Yes No	Determines whether or not the attached console terminal uses 8-bit ASCII characters.	No

Chapter 5

Booting and Running a System on a VAXELN Target

In order to run a system image that has been produced by the VAXELN System Builder (a file with the file type .SYS), the image must be booted on the target system.

The three methods available for transferring your application to the target are

- Disk and TU58 or TK50 tape
- Downline loading over Ethernet
- Programmable read-only memory (PROM)

This chapter describes how to use each of these methods.

5.1 Booting Systems from Disks and Tape

The system image can be booted from any Files–11 disk that is supported as a boot device by the target processor; the disk need not be one that is supported by the VAXELN executive. In addition, the TK50 cartridge tape drive can be used for booting a MicroVAX II. In this case, the procedure is the same as for a disk. The TU58 cartridge tape drive can also be used on the VAX–11/725, VAX–11/730, and VAX–11/750.

VAXELN provides a COPYSYS.COM command procedure in the ELN$ directory on the development system that is used to create a disk or tape containing a bootable system image. When executed, the COPYSYS procedure prompts for the system image file name and the output device name. It then asks if the disk or tape should be initialized.

NOTE

> System images using the COPYSYS procedure described in this section must be built with the System Builder System Characteristics menu *Boot method* option set to Disk. (See Chapter 4 for more information on the System Builder and the EBUILD command.)

The following example shows the use of the COPYSYS procedure to place the system image called TEST on a TU58 tape cartridge:

```
$ @ELN$:COPYSYS
System image file:  TEST
Output disk:  DDAO
Initialize the disk? (Y/N) [N]:  Y
$
```

This example assumes that a separate TU58 drive named DDA0 is available on the host system. If you are instead using the console device, CSA1:, and you receive the error message "No such device" when executing COPYSYS, the console device has not been made available for use. To make the console device available, have your system manager connect it with the following command:

```
$ MCR SYSGEN CONNECT CONSOLE
```

You have now created a disk containing a VAXELN system image. 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 TEST DDAO
```

Here, the default "no initialization" is chosen.

The cartridge containing the application system can be transferred to the target machine, (in this example, a VAX-11/750), and booted with the following console boot command:

```
>>> B DDO
```

The system responds by displaying the following information:

```
%%

                VAXELN V2.1-03
```

For a VAX-11/730, the console boot command takes the name of the device to be booted. For example:

```
>>> B DQ1
```

Here, presumably DQ1 contains an RL02 cartridge that you have prepared with the COPYSYS command procedure.

For a TU58 cartridge in the external VAX-11/730 drive, the console command is

```
>>> B DDO
```

The command for booting from an RX50 diskette (DUA1) in the first floppy disk drive on a MicroVAX is

```
>>> B DUA1
```

5.2 Downline 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.

Downline loading of a VAXELN system uses a downline load and bootstrap loader, which resides on the target machine, as well as 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 stored 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 downline. 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 the *VAXELN User's Guide*.)

Perhaps the easiest method of downline loading is with the VAXELN Remote Debugger (whether or not you intend to debug your system). This method is described in Section 5.2.8, as well as in Chapter 6.

5.2.1 Preliminary Steps

You need to perform the following preliminary steps to set up your host and target machines before downline 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 database.
5. Configure or install the downline load bootstrap loader on the target machine.

The following sections describe these steps in more detail.

Note, however, that before continuing with the setup procedures, you should 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 *Guide to Networking on VAX/VMS.*

5.2.2 Installing Communication Hardware on the Target Machine

The communication hardware should be installed at the default input-output bus address on the target processor. Table 5–1 lists the address assumed by the downline load bootstrap loader for each particular hardware device.

Table 5–1: Datalink Device Default Addresses

Device	Address (Octal)
DEUNA	774510
DEQNA	774440

5.2.3 Configuring a Host for Downline Loading

The following commands must be issued to configure your VAX/VMS host for downline loading. These commands enable the host's recognition of boot-request messages from the target system.

NOTE

These commands are valid for a VAX/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.

```
$    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
```

5.2.4 Adding the Target Machine to the Host Node Database

The target VAXELN machine needs to be described in the host system's network node database. To enter the machine in the database, 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 database using the DEFINE command. For example, for a node named "ARTHUR":

```
$    RUN SYS$SYSTEM:NCP
NCP>    DEFINE NODE ARTHUR ADDRESS 42 SERVICE_CIRCUIT UNA-0
NCP>    DEFINE NODE ARTHUR HARDWARE_ADDRESS AA-00-03-00-00-E1
```

NOTE

These commands are valid for a VAX/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.

The node address and name may have already been specified when your network was installed, but you should 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, which connects the host system to the target machine.

The hardware address is required for downline loading through 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 DIGITAL field service representative, who can provide the address by running the controller's diagnostic package.

Once the target machine had been added to the host system's permanent database, the information should be copied to the current volatile database using the SET command. For example:

```
NCP>  SET NODE ARTHUR ALL
```

After the DEFINE and SET ALL commands have been used, the target machine's description remains permanently in both databases; that is, the target machine description remains even 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 the OPER privilege.

5.2.5 Configuring the Bootstrap Loader

The downline 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 downline loader is contained in the boot read-only memory (ROM).

To install the downline 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 files 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 the CMKRNL privilege. It is recommended that you use the NEWBOOT procedure from the fully privileged system manager account.

Since the MicroVAX downline 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, and thus enable unattended downline loading of the target machine. (See the system configuration section of the *MicroVAX I Owner's Manual* for details.)

5.2.6 Downline Loading Procedure

To downline load a target machine, the VAXELN system image file must be available to the network software on the host development system, and the downline load bootstrap loader must be running on the target machine.

When you build the system with the System Builder, be sure to specify *Downline* for the *Boot method* option 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 database using NCP. For example:

```
NCP>
```

The same operation can be performed by the remote debugger as described in Section 5.2.8.

Once the system image file has been made known to the network software, the downline load bootstrap loader can be started using the console boot command ("B") on the target machine. For example, to start the DEUNA loader on a VAX-11/730, type

```
>>>
```

For a VAX-11/750, type

```
>>>
```

For a MicroVAX I or II, type

```
>>>
```

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 that reads the specified VAXELN system image file and sends the image file to the target bootstrap loader.

When you downline load a machine (in contrast to bootstrapping it from a disk or read-only memory), you do not need to use the System Builder to set the node name or node address; as part of the load procedure, the target machine receives its proper node name and address. Thus, 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.

5.2.7 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 downline load bootstrap loader by setting the default bootstrap selection switches to the correct read-only loader. On the VAX–11/730, this setting is performed by the NEWBOOT command procedure. On a VAX–11/750, set the Default Boot Device switch to "A"; on the MicroVAX I, set the CPU configuration DIP switch number 1 to "on".

To trigger a target machine, use the NCP TRIGGER command. (You must have also built the system with a value of *Yes* for the *Node triggerable* option of the Edit Network Node Characteristics menu.) For example:

```
NCP>   TRIGGER NODE ARTHUR
```

The trigger function sends a "boot-request" message to the target machine, which causes the VAXELN datalink device driver to halt execution of the VAXELN executive and begin execution of the default bootstrap, the downline load bootstrap loader.

NOTE

If desired, the DEUNA controller on a VAX–11/730 or VAX–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 VAX–11/730 or VAX–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 a 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.

5.2.8 Downline Loading Using the VAXELN Remote Debugger

During the VAXELN programming and development cycle, the target machine is likely to be loaded downline and remotely debugged many times. To facilitate this operation, the remote debugger can load machines downline. In fact, using the remote debugger is one of the easiest methods for downline loading, whether or not you intend to remotely debug your system.

The /LOAD qualifier on the DEBUG/REMOTE command performs two functions:

- It causes the specified VAXELN system image file name to be stored in the network node database.
- It triggers the target machine's downline load bootstrap loader.

For example, the system TEST.SYS is loaded during a remote debugger session as follows (you must have the OPER privilege to execute this command):

```
$ DEBUG/REMOTE /LOAD=DISK$WORK:[ROBOT]TEST.SYS ARTHUR
```

and the effect of this command is equivalent to the effect of the following pair of commands (see Section 5.2.7 for information on triggering):

```
NCP> SET NODE ARTHUR LOAD FILE DISK$WORK:[ROBOT]TEST.SYS
NCP> TRIGGER NODE ARTHUR
```

You can use the /NOCONNECT qualifier on the DEBUG/REMOTE command if you just want to load without debugging.

See Chapter 6 for a complete description of the remote debugger.

5.2.9 Reloading Production Machines Downline

Once a VAXELN application has been debugged and is installed in production use, you can continue to use the downline load facilities to load the target machines. The host's node database 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 downline load bootstrap loader, as described in Section 5.2.6. Whenever a target machine is rebootstrapped (for example, after a power failure or a serious hardware or software failure), it will be reloaded by the host system.

Chapter 6

Debugging VAXELN Ada Programs

This chapter presents information on debugging VAXELN Ada programs with the VAXELN Remote Debugger. The remote debugger allows you to debug a program running on a target machine "remotely," that is, from a host VAX/VMS system. The host and target must be connected by Ethernet, as described in the *VAXELN Ada Installation Guide*. Remote debugger features include support for most VAX/VMS Debugger (DEBUG) features, including Ada-specific commands with additional support for features that allow you to

- Get information about job and system resources
- Create jobs
- Delete jobs
- Halt and continue jobs
- Change job priorities

You may also wish to refer to *Developing Ada Programs on VAX/VMS* for detailed information on debugging VAX Ada programs.

6.1 The Debugging Environment

You need to correct your program when any of the following happens:

- The compiler flags errors.
- The linker detects errors.
- The VAXELN System Builder detects errors.
- The run-time library detects errors.
- You determine, based on receiving incorrect output during a program's execution, that a logic error exists.

The VAXELN Remote Debugger lets you control the execution of your program so that you can monitor specific locations, change the contents of locations, check the sequence of program control, and otherwise locate and correct errors as they occur. In the process, the remote debugger lets you use the same symbolic names that appear in your program to refer to variables, compilation units, labels, and so on. After you track down the errors, you can edit your source program and compile, link, EBUILD, downline load, and execute the corrected version.

6.1.1 VAXELN Remote Debugger Support for VAX/VMS Debugger Features

You can use most VAX/VMS Debugger commands and features with the VAXELN Remote Debugger with the exception of

- SHOW AST
- ENABLE AST
- DISABLE AST
- CTRL/Y DEBUG
- SHOW EXIT_HANDLER

Appendix B provides command descriptions for remote debugger commands and a summary of VAX/VMS Debugger commands used with VAXELN Ada programs.

6.1.2 VAXELN Terms and Concepts Related to Debugging

This section lists and defines the terms used in this chapter and presents a brief discussion of VAXELN concepts used in debugging. Some terms, such as job and process, have different meanings in VAX/VMS and VAXELN environments. See Chapter 8 for further discussion.

Ada tasks in the VAXELN environment

In the VAXELN Ada environment, each Ada task is a subprocess of the Ada main program (the main process) within a VAXELN job. It is also possible for a VAXELN Ada program to create processes that are not Ada tasks with the CREATE_PROCESS procedure (for example, a Pascal process block). See Chapter 8 for further information on the use of tasks in VAXELN Ada.

In the debugger (and Ada) context, %TASK 0 is the null task and %TASK 1 is the main program. Each task introduced thereafter by the Ada program is given the next sequential task number. The main (environment) task is equivalent to the master process.

debug-wait state

A debug-wait state is the state that a job or process is in when it is waiting for the attention of the debugger. A process enters this state when it hits a breakpoint, tracepoint, or watchpoint or raises an unhandled exception. Once a process in a job is in a debug-wait state, process-switching is disabled for the job, so the entire job is considered to be in a debug-wait state.

A process can be placed into a debug-wait state using the SET JOB/HALT command.

job

Within a VAXELN system, programs are executed by jobs. Within each job, a master process executes the main program and zero or more subprocesses. In the VAXELN Ada environment, each Ada task is a subprocess. For any job (regardless of how many subprocesses it may create), there is one and only one image (.EXE). It is also possible for a VAXELN Ada job to have one or more subprocesses that are not Ada tasks.

All processes in a job share the same P0 region. Each process has its own P1 region.

The "family" of a master process and any of its subprocesses is a job in VAXELN terminology. Jobs can be created dynamically to execute a specific program that was included with the System Builder. You can create jobs with the CREATE_JOB procedure or with the debugger command CREATE JOB.

One copy of a program's code is shared by any number of jobs executing that program. A System Builder option also allows you to specify which jobs should be created by the kernel when the system is started, to run designated programs in the system and, as an additional option, to perform initialization sequences before other jobs are allowed to execute. The options are detailed in Chapter 4.

job specifier

A job specifier identifies a VAXELN job, a set of jobs, or a process within a job. The job specifier may be a job ID or program name and may contain wildcards. The syntax for a job specifier for a job can be

```
job_ID        (4)
job_ID.n      (4.1)
job_name      (server)
job_name.n    (server.1)
```

where *n* is a process number in that job.

If multiple jobs are running the same program, then the program name will not identify a unique job. Use the job ID to uniquely identify a particular job.

process

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 Ada, this thread is called the environment task.

command session

A command session is used to enter commands that are directed to a specific job. This session is a dialog between you and a specific job running on the target system. Command sessions are started with the SET JOB/CURRENT command. The command session prompt is

```
RDBG>
```

Only one command session may be active at a time. To end a command session and begin a new one, use the SET JOB/CURRENT command. Note that when a command session is ended, the context of that command session is discarded, including watchpoints, breakpoints, and tracepoints, and the job is left in whatever state it was in prior to the termination of the command session.

system session

The system session is used to enter commands when the process being debugged is not waiting for debugging input. This session does not permit a debugging dialog with a specific job; that is, commands entered in the system session are not directed at any particular job. The system session prompt is

```
RDBG*>
```

The system session is the current session when there is no command session, or when CTRL/C was pressed during a command session. Only general debugging commands are allowed in the system session. Many commands, such as EXAMINE, DEPOSIT, STEP, and GO are not allowed.

6.2 Compiling and Linking Programs for Debugging

In VAXELN Ada, as in VAX Ada, to execute a program with the debugger, you first compile and link the program with the /DEBUG qualifier. The /DEBUG qualifier is a default to the DCL ADA command and ACS COMPILE and RECOMPILE commands, so normally you should only have to link your program with the /DEBUG qualifier. In the following example, the /DEBUG qualifier is appended to the ADA command only for emphasis.

For example,

```
$  ADA /DEBUG MY_PROGRAM
```

The /DEBUG qualifier on the ADA and any of the compilation commands requests the compiler to write the symbol records associated with the units being compiled into the resulting object modules. These records permit you to specify in debugging commands the names and other identifiers declared in your program—that is, the names of any variables, subprograms, packages, and so on.

If you use the ADA or compilation commands with the /NODEBUG qualifier, only symbol records for global symbols are included in the object modules. Global symbols consist of any names that the program exports to modules in other languages by means of an export pragma: EXPORT_PROCEDURE, EXPORT_FUNCTION, EXPORT_OBJECT, EXPORT_EXCEPTION, and PSECT_OBJECT.

While debugging, you will probably want to refer to the source code of the program, either in a debugger-generated display (see the screen mode description in the *VAX/VMS Debugger Reference Manual*) or in a listing file (.LIS). To obtain a listing file, use the /LIST qualifier with your compilation command.

After compiling a program, you must link it using the /DEBUG qualifier ACS LINK command. If the system name has not been specified with the pragma SYSTEM_NAME or with the ACS SET PRAGMA /SYSTEM_NAME= command, then you must specify the /SYSTEM_NAME=VAXELN qualifier to the ACS LINK command as well. This links the program against the VAXELN Ada run-time library and the VAXELN system libraries.

For example,

```
$ ACS LINK/DEBUG/SYSTEM_NAME=VAXELN MY_PROGRAM
Invoking VAX/VMS Linker
```

The /DEBUG qualifier on the ACS LINK command requests the linker to include in the executable image all symbol information that is contained in the object modules.

6.3 Setting up the Remote Debugger Environment

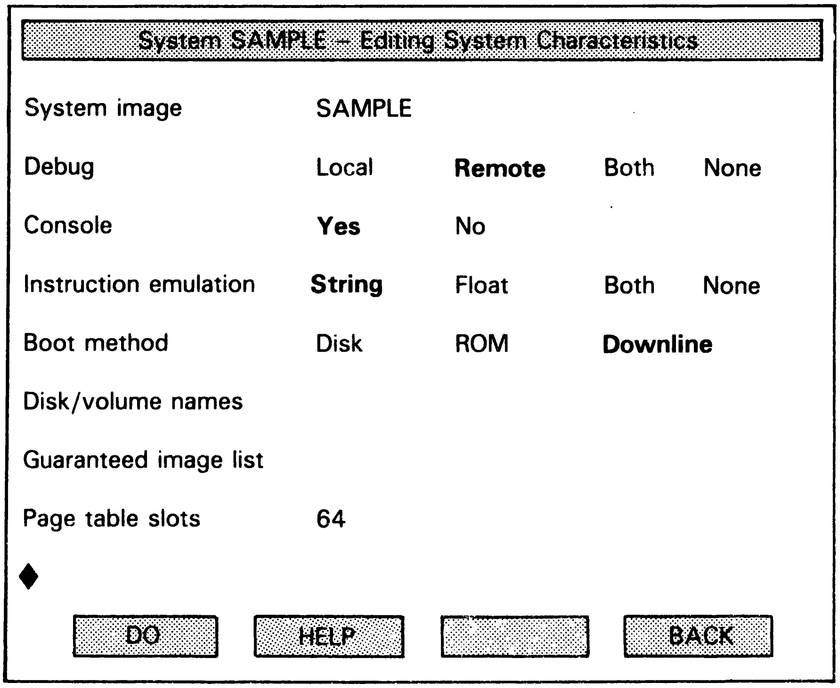
After you have developed the programs needed by a VAXELN system, you create the system image using the VAXELN System Builder Utility (EBUILD). To use the remote debugger, you must build the VAXELN system image to include remote debugging capabilities. This section describes how various system characteristics affect the VAXELN Remote Debugger, and how to select debugging options for programs in the system. For more information on using the VAXELN System Builder, see Chapter 4 or the *VAXELN User's Guide*.

You create a VAXELN system image with the VAXELN System Builder Utility, using the DCL command EBUILD. The format for this command is

```
EBUILD [/qualifier(s)] data_file
```

Issuing the EBUILD command places you at the Main menu (see Figure 6–1) for the VAXELN system you are creating. From this menu, you may choose options to describe or modify system contents and characteristics.

Figure 6–1: Main System Menu



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			

♦

ZK-4858-85

6.3.1 System Characteristics

Characteristics that control the overall properties of the system can be selected from the System Characteristics menu. To access this menu, choose the Edit System Characteristics option from the Main menu.

Remote debugging is affected by the following system characteristics:

- Debug—The options specify which debuggers are built into the system, if any. Including a debugger during the development of a system is useful even if no program description has the debug option. This ensures that the debugger gains control in the event of an exception that is not handled by a program.

Menu Option	Description
Local	Designates the local debugger image EDEBUGLCL is included. This allows you to enter EDEBUG commands from the hardware console device.
Remote	Designates the remote debugger image (EDEBUGREM), for use with either the EDEBUG utility or the remote debugger.
Both	Designates that both the local debugger image and remote debugger image are included for use with either the EDEBUG utility or the remote debugger.

- Console—The console parameter specifies whether a console driver should be included in the VAXELN system image. Note that the console driver is included implicitly if the Local or Both debug option is in effect.

Menu Option	Description
Yes	Specifies that a console driver should be included in the system. Console input-output is directed to the hardware console device.
No	Specifies that a console driver should not be included in the system. Console input-output is directed to the remote debugger.

If the console driver is not included in the VAXELN system image, the remote debugger directs all output to the output display and the log file, if logging is enabled. The remote debugger uses a special prompt to indicate that a remote console input request needs to be satisfied.

- Boot Method—Selects the method by which the finished system is booted on the target machine. The default is downline.

Menu Option	Description
Disk	Specifies that the target system will be booted from disk.
ROM	Specifies that the target system will be booted from read-only memory. The ROM method is normally only used when an application has been fully debugged.
Downline	Specifies that the target system will be loaded downline.
Node triggerable	Specifies whether the downline load triggers are enabled. Selecting Yes, the default, means that the system can be remotely triggered.

- Virtual Size—Selects the maximum size, in 512-byte pages, of each P0 and P1 region on the system. The value is used by the kernel to allocate process page tables for each job and process.

Value	Description
default value	The default value for a job's P0 region and for each process's P1 region is 1024 pages.
maximum value	The maximum value is 32,640 pages for a job's P0 region and an equal amount for each process's P1 region.

The remote debugger uses this value to determine how much of its P0 space must be reserved to map the VAXELN process' P0 space. The default Virtual Size values will allow the remote debugger to run on the VAX/VMS system without changing the default VAX/VMS VIRTUALPAGECNT SYSGEN parameter (VAXELN systems with higher Virtual Size values may require the SYSGEN parameter to be increased so that the remote debugger may allocate enough virtual memory to map both the remote virtual memory and virtual address space needed to contain the debugger).

Note that the Virtual Size value is used by the VAXELN kernel when allocating page tables for *each* job and process on the system. Therefore, if this value is large to satisfy the requirements of a single program, the remote debugger must allocate large amounts of virtual memory to debug *any* program on the system even though the program being debugged only uses very little virtual memory.

6.3.2 Program Descriptions

Each executable image (.EXE) included in the system must be described in a program description. To access the Program Description menu, select the Edit Program Description or Add Program Description option from the Main menu. Both options allow you to select a debug option that determines whether this program should begin execution under debugger control.

- Debug—Choose "Yes" or "No" to select whether this program will begin execution under control of the debugger. "Yes" is the default.

Menu Option	Description
Yes	Select "Yes" if you want any VAXELN job that runs this program to give control directly to the debugger.
No	Select "No" if you want any VAXELN job that runs this program to execute immediately without debugger intervention. (Even if the program is executed without having given control first to the debugger, a subsequent unhandled exception can give control to the debugger; this includes asynchronous exceptions that may be raised using the debugger command SET JOB/HALT.)

6.4 Using the VAXELN Remote Debugger

The remote debugger allows you to use most standard VAX/VMS Debugger commands, with the addition of a command set designed for use in remote debugging. The remote debugger also lets you use Ada-specific debugging commands, such as SHOW TASK. These commands are described fully in *Developing Ada Programs on VAX/VMS* and summarized in Appendix B of this book.

The remote debugger uses the logical names RDBG$INPUT and RDBG$OUTPUT for input-output. They are usually assigned to SYS$INPUT and SYS$OUTPUT respectively. This allows remote debugger commands to be entered with VAX/VMS command procedures.

The following sections present an overview on starting and terminating sessions with the remote debugger. Topics include

- The DEBUG/REMOTE command
- Downline loading files
- Using HELP
- Exiting a remote debugger session
- Interrupting and reinvoking the remote debugger
- Invoking and exiting command and system sessions

6.4.1 The DEBUG/REMOTE Command

The DCL command DEBUG/REMOTE invokes the remote debugger. To use the remote debugger the host system must be connected to the target machine by Ethernet. You can specify any node connected to the host system as the node from which the session is to be conducted. You can also use the DEBUG/REMOTE command to load new systems from the host to the target node, to start systems with or without the debugger in control, and to reconnect to nodes that were loaded previously. Section 6.9 illustrates use of this command.

The DEBUG/REMOTE command has the format

```
DEBUG/REMOTE [/optional_qualifier(s)] node_name
```

The optional qualifiers are

Option	Description
/[NO]CONNECT	Controls whether you can enter debugging commands. /CONNECT is the default.
/[NO]GO	Controls whether jobs that initially require debugging continue in their execution or are left suspended. /NOGO is the default.
/LOAD=file_specification	Controls whether the file specification that names a VAXELN system image should be downline loaded to the target node.

For example,

```
$ DEBUG/REMOTE ARTHUR
```

invokes the remote debugger in debugging programs of the target node ARTHUR. The prompt

```
RDBG*>
```

indicates that you are in a remote debugging system session and can enter remote debugging commands.

Use of the /LOAD qualifier alone and in combination with other qualifiers is presented in the next section.

6.4.2 Downline Loading During Debugging

During the VAXELN programming and development cycle, the target machine is likely to be loaded downline and remotely debugged many times. To facilitate this operation, the remote debugger can load machines downline.

The downline load operation may be performed using the DCL command

```
   DEBUG/REMOTE  /LOAD=file_specification  node_name
```

The /LOAD qualifier stores the specified system image file name in the host system's network node database and triggers the target machine's downline load bootstrap loader.

The following table describes the effect of using the /LOAD qualifier alone and in combination with other DEBUG/REMOTE qualifiers.

Format	Function
DEBUG/REMOTE /LOAD=file_spec node_name	Sets the load file for the target node and triggers it with the remote debugger in control.
DEBUG/REMOTE /LOAD=file_spec /GO node_name	Sets the load file for the target node and triggers it with the remote debugger in control. Issues the equivalent of the GO command for each job in a debug-wait state.
DEBUG/REMOTE /LOAD=file_spec /NOCONNECT node_name	Sets the load file for the target node and triggers it. Returns immediately to DCL. No debug dialog takes place.

6.4.3 Using HELP

At any time during a remote debugging session, you can get on line HELP by using the HELP command. For example, if you type HELP STEP at a remote debugger prompt, the debugger displays information about its STEP command. Type the HELP HELP command to obtain information on using the HELP facility.

6.4.4 Exiting from the Remote Debugger

You can leave a remote debugging session by typing the EXIT or QUIT commands, or by pressing CTRL/Z at either the command or system session prompt.

The EXIT or QUIT commands, and CTRL/Z all cause the orderly termination of the debugging session.

When you leave a remote debugging session, any tracepoints, breakpoints, and watchpoints from that command session are lost and the program is left in its current state from the time you terminated the command session.

Pressing CTRL/Y interrupts the remote debugger and brings you back to the DCL prompt.

6.4.5 Interrupting and Reinvoking the Remote Debugger

If you wish to temporarily exit the remote debugger without losing the session's context, use the SPAWN or ATTACH command to create a subprocess or switch to an existing process. For example:

```
RDBG> SPAWN MAIL
MAIL> EXIT
%RDEBUG-I-RETURNED, control returned to process...
```

Note that when you leave the subprocess and recommence the remote debugging session, you return to the previous point in the command session and that any watchpoints, breakpoints, or tracepoints that were set still exist.

6.4.6 Interrupting and Restarting a Remote Debugger Command Session

If the command session is not waiting for input (for example, if the job attached to the command session is running), you can press CTRL/C to enter the system session. For example

```
RDBG> GO
       CTRL/C
RDBG*> SET JOB/HALT
RDBG>
```

6.4.7 Exiting from a Remote Debugger System Session

There are three ways to leave a system session.

- To return to DCL level from a system session, type the EXIT or QUIT commands, or press CTRL/Z.
- To begin a new command session, type the SET JOB/CURRENT command.
- To return to an active command session, press the RETURN key at the RDBG*> prompt.

6.4.8 Console Input-Output

Remote console input and output requests are only processed when there is no debugger prompt displayed. Console output is prefixed with the node name enclosed in square brackets. Console input uses a prompt that consists of the node name followed by an angle bracket. For example,

```
RDBG> GO
[ALLUDE] Enter today's date
ALLUDE> 09-AUG-1986
[ALLUDE] Today's date is 09-AUG-1986
```

6.5 Monitoring Job and System Activity

In addition to support for DEBUG commands, the remote debugger provides commands that allow you to monitor system activity, create and delete jobs, control job execution, and select jobs for debugging.

6.5.1 Monitoring System Activity

In order to monitor the overall execution of the system, you can use the SHOW SYSTEM command. This provides a list of the jobs running on the system, and their current status. For example:

```
RDBG*> SHOW SYSTEM

  Job   Program    Priority   State    Shared Size  Readonly size
   2    XQDRIVER       1     waiting      31232         30208
   3    EDEBUGREM      3     running       5120         11264
   4    EXAMPLE_B      5     waiting      35832         26894
   5    EXAMPLE_A      5     waiting      32768         30208
```

To get information about specific jobs, use the SHOW JOB command. To get information about each process that makes up the job, use the SHOW JOB/FULL command.

For example:

```
RDBG*> SHOW JOB EXAMPLE_A

      Job    Program     Priority   State    Shared Size  Readonly size
        5    EXAMPLE_A      5      waiting      32768        30208

RDBG*> SHOW JOB/FULL EXAMPLE

      Job    Program     Priority   State    Shared Size  Readonly size
        5    EXAMPLE_A      5      waiting      32768        30208

      Process         Priority   State   Stack Size    CPU time
         1                1     waiting       2560     00:00:11.21
         2                8     waiting       2560     00:00:00.17
         3                2     waiting       2560     00:00:00.06
         4                4     waiting       2560     00:00:02.32
        10                8     waiting       2560     00:00:00.16
```

6.5.2 Creating Jobs

It is often useful to introduce new jobs to the running system while debugging a VAXELN system. The remote debugger provides commands to create jobs. Jobs may be created from programs that are already present in the VAXELN system, or from program images loaded into the VAXELN system prior to creating the job.

To create a job running a program that was included in the VAXELN system image, use the CREATE JOB command. This command is useful when a job exits and you wish to rerun the program, perhaps with tracepoints and breakpoints set to monitor execution.

To create a job running a program that was not included in the VAXELN system image, use the CREATE JOB/LOAD command. The /LOAD qualifier allows you to load a program image from a file that is accessible from the target node prior to creating a job running that image. This command is useful when you wish to try a new version of a program without rebooting the VAXELN system.

For example, if you were to find a bug in a program running on the target system, you could edit the source program, compile and link the program, and create a new job running the corrected program.

When using the /LOAD qualifier with the CREATE JOB command, you may also specify the kernel stack size (/KERNEL_STACK) and starting job priority (/PRIORITY) that will be used for jobs running the program.

Note that once a program image is loaded using the CREATE JOB/LOAD command, you may create other jobs running that program without loading new copies of the program image. In this case, the /LOAD qualifier may be omitted unless you deleted the program from the system using the DELETE JOB/UNLOAD command.

For example:

```
RDBG*> CREATE JOB EXAMPLE /LOAD=DUA0 [EXES]EXAMPLE EXE /PRIORITY=5
Job 4 (EXAMPLE) is waiting for your attention
```

The program image is loaded from the file DUA0:[EXES]EXAMPLE.EXE on the target system. The job is created with a starting job priority of 5. Note that the file is opened in the context of the target system.

```
RDBG*> CREATE JOB EXAMPLE
Job 5 (EXAMPLE) is waiting for your attention
```

The job is created using the program image loaded in the previous example. Since a priority of 5 was specified when the program was loaded, this job is also created with a starting job priority of 5.

6.5.3 Deleting Jobs from a VAXELN System

You can delete jobs that are running on a target system with the DELETE JOB command. All objects owned by the process, including subprocesses, are deleted; all open files are closed.

To unload a program image that was loaded by using the CREATE JOB /LOAD command or by the LOAD_PROGRAM procedure, described in Appendix A, the /UNLOAD qualifier may be used. Note that the program is only unloaded when all jobs running that program have exited.

For example:

```
RDBG*> DELETE JOB EXAMPLE /UNLOAD
```

This command deletes the job EXAMPLE from the system. The /UNLOAD qualifier indicates that this program, which was loaded using the CREATE JOB/LOAD command, should be unloaded when the program exits.

You must be careful not to delete jobs such as EDEBUGREM (the remote debugger nucleus), or XQDRIVER (the network driver), because the VAXELN Remote Debugger will then not be able to communicate with the target system.

6.5.4 Selecting a Job to Be Debugged

When the remote debugger is invoked, it displays a list of all jobs that are in a debug wait state. A job is in a debug wait state when it has a process that:

1. Is starting under debugger control
2. Raised an unhandled exception
3. Is waiting at a breakpoint
4. Is waiting at a tracepoint

Under normal circumstances, the last two states, at a breakpoint, or at a tracepoint, occur only if a command session was left in that state previously. This may happen if you exit from the remote debugger when the command session is stopped at a breakpoint.

The SET JOB/CURRENT command may be used to begin a command session with any job that is in a debug wait state. If the program is not in a debug wait state, then the SET JOB/HALT command may be used to put that job in a debug wait state.

For example:

```
RDBG*>  SHOW JOB EXAMPLE*

        Job   Program     Priority   State     Shared Size  Readonly size
          4   EXAMPLE_A       5      waiting      3584          66560
          6   EXAMPLE_B       8    debug-wait     3584          66560

RDBG*>  SET JOB/CURRENT EXAMPLE_B
%RDEBUG-I-SESSION_INIT, Loading symbols for Job 6. (EXAMPLE_B)
%RDEBUG-I-FROM, from file TESTD:[EXE]EXAMPLE_B.EXE;1
%RDEBUG-I-INITIAL, language is ADA, module set to 'EXAMPLE'
%RDEBUG-I-NOTATMAIN, type GO to get to start of main program
RDBG>
```

Of the jobs listed previously, only job EXAMPLE_B is currently ready to begin a command session. If you wanted to begin a command session with job EXAMPLE_A, then you would have to use the SET JOB/HALT command first.

The program image file that was supplied to the System Builder must be available to the remote debugger so that it can read in the symbol table information from it. The file must be local to the host system. The target system provides the remote debugger with the program image file specification that the System Builder used when building the system image. If the file has moved, then the /IMAGE qualifier must be used with the SET JOB/CURRENT command to indicate where the file may currently be found. If the program was loaded using the CREATE JOB /LOAD command, then the /IMAGE qualifier must be used to indicate where the file may be found.

For example:

```
RDBG*> CREATE JOB EXAMPLE /LOAD=DUA0:[EXES]EXAMPLE.EXE
Job 5 (EXAMPLE) is waiting for your attention
```

Here, since the file specified with the /LOAD qualifier is opened in the context of the target system, the file is loaded from DUA0:[EXES]EXAMPLE.EX on the target node.

```
RDBG*> SET JOB/CURRENT/IMAGE=[LOCAL_EXES]EXAMPLE.EXE EXAMPLE
%RDEBUG-I-SESSION-_INIT, Loading symbols for Job 6. (EXAMPLE_B)
%RDEBUG-I-FROM, from file TESTD:[LOCAL_EXES]EXAMPLE.EXE;1
%RDEBUG-I-INITIAL, language is ADA, module set to 'EXAMPLE'
%RDEBUG-I-NOTATMAIN, type GO to get to start of main program
RDBG>
```

The /IMAGE qualifier may then be used with the SET JOB/CURRENT command to specify the local copy of the executable image that the remote debugger may use.

You should take great care that the file specified with the /IMAGE qualifier is the same as the one running on the target node. Otherwise, the symbol table information used to debug the program will not match the actual program that is running on the target system.

6.5.5 Controlling Execution of Jobs

It is often useful to debug a particular job in isolation from other jobs in the system. You can use the SET JOB/HALT command to freeze the state of jobs in the system. This causes the specified jobs to raise an asynchronous debug exception, thus placing the job or jobs in a debug wait state. By using wildcards and job lists you can stop the execution of several jobs using one command. For example:

```
RDBG> SET JOB/HALT JOB_A,JOB_B,*SERVER,7
```

You must be careful not to halt jobs such as EDEBUGREM (the remote debugger nucleus), or XQDRIVER (the network driver), as the remote debugger will then not be able to communicate with the target system.

In order to continue execution of jobs that are in a debug wait state, the SET JOB/CONTINUE command may be used. This has the effect of executing a GO command for the process that put the job in a debug wait state. If the job is associated with the command session, then the command session is ended and the system session is reentered.

By using wildcards and job lists, it is possible to continue the execution of several jobs using one command. For example:

```
RDBG>  SET JOB/CONTINUE JOB_A,JOB_B,*SERVER,7
```

6.5.6 Changing Job Priorities

The SET JOB/PRIORITY command may be used to assign new priorities to jobs running on the target system. This allows you to try different job priority combinations without rebuilding or rebooting the VAXELN system each time a change is made. The job must be in a debug wait state to use this command. For example:

```
RDBG*>  SHOW JOB 4

  Job  Program     Priority     State     Shared Size   Readonly size
    4  SQRT_SERVER    16      debug-wait     3584          66560

RDBG*>  SET JOB/PRIORITY=14 4
RDBG*>  SHOW JOB 4

  Job  Program     Priority     State     Shared Size   Readonly size
    4  SQRT_SERVER    14      debug-wait     3584          66560
```

The priority of Ada tasks (which are implemented as VAXELN processes) may be changed using the SET TASK/PRIORITY command.

6.6 Controlling Program Execution

You can enter commands by typing in the command name or by using the keypad. When you use the keypad, you can use the predefined VAX/VMS Debugger functions that are set when you invoke the debugger, or you can redefine the commands associated with the keys.

Note that VAX/VMS Debugger commands that control program execution can only be used in a remote debugger command session.

6.6.1 The GO command

Use the GO command to start program execution after invoking the remote debugger with the DEBUG/REMOTE command and beginning a command session, and to resume program execution any time thereafter—that is, whenever the program has been suspended for any reason and the remote debugger command session prompt (RDBG>) is displayed.

When you run a VAXELN Ada program under debugger control, execution is suspended initially before the elaboration of library packages. This gives you the option to control and observe the package elaboration phase by using the techniques described in the next sections.

6.6.2 Breakpoints, Tracepoints, and Watchpoints

The sequence of events when your program encounters a breakpoint is as follows: the debugger suspends program execution, displays the line number and source line (or instruction) where the breakpoint occurred, executes any DO command sequence (if specified), and prompts for a command. For example:

```
RDBG>  SET BREAK %LINE 37 DO (EXAMINE TEST_RESULT)
RDBG>  GO
break at COUNTER.%LINE 37
    37: end loop;
COUNTER.TEST_RESULT:   253.02
RDBG>
```

As with breakpoints, every time a tracepoint is reached, the debugger issues a message and displays the source line. However, the program continues executing past the tracepoint.

To display the currently active breakpoints or tracepoints, use the SHOW BREAK or SHOW TRACE command, respectively. For example:

```
RDBG>  SHOW BREAK
breakpoint at COUNTER.%LINE 37
RDBG>
```

To cancel a breakpoint or tracepoint, use the CANCEL BREAK or CANCEL TRACE command, respectively, specifying the program location exactly as you did when setting the breakpoint or tracepoint. The CANCEL BREAK/ALL command cancels all breakpoints. The CANCEL TRACE/ALL command cancels all tracepoints.

A breakpoint suspends execution at the first byte of the specified location, so that the instruction beginning at that location does not execute. If you set a breakpoint on a subprogram, the call to the subprogram is executed before the debugger takes control and issues the message "routine break at routine NAME". Note that all breakpoints are lost when exiting the remote debugger and the program is left in a suspended state.

The SET WATCH command lets you specify program variables that the debugger will monitor as your program executes. This process is called setting watchpoints. If the program modifies the value of a "watched" variable, the debugger informs you by suspending execution and displaying information. The debugger monitors watchpoints continuously during program execution. Note that the SET BREAK/MODIFY command is identical to the SET WATCH command.

When using the SET WATCH command, the address expressions you specify as parameters are *variable names*. This is in contrast to using the SET BREAK or SET TRACE commands, where the address expressions you specify as parameters are *program locations* (line numbers, subprograms, labels, and so on). For example, the following command sets a watchpoint at the variable TEST_RESULT:

```
RDBG>  SET WATCH TEST_RESULT
RDBG>
```

Subsequently, every time the program modifies the value of TEST_RESULT, the watchpoint is activated.

Like the SET BREAK and SET TRACE commands, the SET WATCH command accepts optional DO and WHEN clauses.

The sequence of events when your program modifies the contents of a watched variable is as follows: The debugger suspends program execution; displays the location of the watched variable (line number plus byte offset), the source line, and the old and new values of the variable; executes any DO command sequence (if specified), and prompts for a command. The following example sets a watchpoint on the variable X in the module SCREEN_IO and starts execution:

```
RDBG>  SET WATCH X
RDBG>  GO
watch of SCREEN_IO.X.%LINE 13+3
     13:   X := X + 1;
    old value: 16
    new value: 17
break at SCREEN_IO.%LINE 14
     14:   SWITCH(X,Y);
RDBG>
```

As with breakpoints and tracepoints, you use the SHOW WATCH and CANCEL WATCH commands to display and cancel the currently active watchpoints.

Note the following restriction when setting watchpoints. As for VAX Ada, you can set watchpoints only on statically allocated variables. The only variables that are statically declared are those declared in library packages. Thus, you can set watchpoints only on variables that are declared in a library package specification or in the declarative part of that package's body. You cannot set watchpoints on variables that are declared in subprograms or tasks, or in packages nested within subprograms or tasks. Such variables are dynamically allocated, either on the stack or in registers. If you try to set a watchpoint on a variable whose storage is dynamically allocated, the debugger issues a message such as the following:

```
%RDEBUG-W-BADWATCH, cannot watch protect address 0000E898
```

6.7 Using the Kernel Debugger

This section describes how to invoke the kernel debugger when the system is booted and how to invoke the kernel debugger on a running system. You must set the system characteristic Debug to "local" or "both" to use the kernel debugger.

The kernel debugger is of use in cases where you need to set breakpoints and examine locations in the VAXELN kernel image. You must do kernel image debugging from the VAX hardware console terminal. See the *VAXELN User's Guide* for more information.

6.7.1 Invoking the Kernel Debugger at Boot Time

Type the following at the VAX hardware console to boot the system under control of the kernel debugger:

```
>>> B/4 <device>
```

The following example shows how to downline load and boot a VAXELN system on a MicroVAX I, with the kernel debugger getting control during the system initialization sequence. The GO command is used to leave the kernel debugging session.

```
>>>  B/4 XQAO
ATTEMPTING BOOTSTRAP

Kernel Edebug V2.1-00

Kernel Breakpoint.
8000122B: NOP
Kernel Edebug>  GO

         VAXELN V2.1-03
```

6.7.2 Invoking the Kernel Debugger on a Running System

Place the VAX processor in hardware console mode (on the MicroVAX, this is accomplished by pressing the HALT button twice; for other VAX processors, the appropriate hardware manual should be consulted).

Type the following at the VAX hardware console:

```
>>>  D/I 14 5
>>>  C
```

This invokes the kernel debugger and causes the following message to be displayed on the hardware console device:

```
Kernel Breakpoint
 8000122B: NOP
Kernel Edebug>
```

Note that if the local debugger was not included when the system was built, (that is, if debug=none or debug=remote was specified), the commands listed previously are ignored.

6.8 Ada Tasking and Remote Debugging

This section discusses considerations in Ada tasking and remote debugging. The remote debugger performs an automatic stack check for Ada tasks whenever the debugger is in control and checks the amount of stack space in use in any task, thus immediately detecting stack overflow.

This is true only for the job associated with the command session.

6.8.1 Stack Storage

As pointed out in the *VAX Ada Programmer's Run-Time Reference Manual,* an undetected stack overflow can occur in certain circumstances, and can lead to unpredictable execution. To help you detect these kinds of stack problems, the remote debugger performs automatic stack checks as you use it.

If the stack pointer is out of bounds, the remote debugger displays an error message. The stack check is performed for the active task after a STEP or BREAKPOINT event triggers (except if /SILENT is used). The stack check is also performed for each task whose state is displayed in a SHOW TASK command—hence, a SHOW TASK/ALL automatically checks the stack of all tasks.

The following are samples of the error messages that the remote debugger displays when a stack check fails. Note that a warning is issued when most of the stack has been used up, even if the stack has not yet overflowed.

```
warning: %TASK 2 has used up over 90% of its stack
  SP: 0011194C  Stack top at: 00111200   Remaining bytes:  1868

error: %TASK 2 has overflowed its stack
  SP: 0010E93C Stack top at: 00111200 Remaining bytes: -10436
```

6.8.2 Monitoring Tasking Performance

Information derived by using the debugging command SHOW TASK /STATISTICS/FULL can be useful in measuring the effects of any changes you may make to improve task performance. For example,

```
RDBG> SHOW TASK/STATISTICS/FULL

task statistics
     Entry calls       = 5      Accepts = 2     Selects = 2
     Tasks activated   = 3      Tasks terminated   = 2
     ASTs delivered    = 0      Hibernations       = 1
     Locks tested      = 86     Locks that blocked = 21, 24%
     Total schedulings = 24
         Due to readying a higher priority task = 1
         Due to task activations                = 3
         Due to suspended entry calls           = 5
         Due to suspended accepts               = 2
         Due to suspended selects               = 1
         Due to waiting for a DELAY             = 1
         Due to scope exit awaiting dependents  = 0
         Due to exception awaiting dependents   = 1
         Due to waiting for I/O to complete     = 8
         Due to delivery of an AST              = 0
         Due to task terminations               = 2
         Due to shared resource lock contention = 0
```

The SHOW TASK/FULL command provides detailed information about each task selected for display. For example,

```
RDBG> SHOW TASK/FULL %TASK 1, %TASK 2

  task id     pri hold state   substate          task object
* %TASK 1      7        SUSP                   1690144

        Task type:      MAIN$TASK
        Created at PC:  19800
        Parent task:    %TASK 0
        Start PC:       19800
        Task control block:                Stack storage (bytes):
          Task value:   159824               RESERVED_BYTES:       10640
          Entries:      0                    TOP_GUARD_SIZE:        5120
          Size:         1440                 STORAGE_SIZE:         30720
        Stack addresses:                     Bytes in use:           632
          Top address:  0001F800
          Base address: 00026FFC           Total storage:          47920
```

```
task id     pri hold state   substate          task object
%TASK 2      4       SUSP  Accept            100352

      Awaiting rendezvous at: accept SQRT_ENTRY
        having do part at address 0000211C

      Task type:       SQRT_TASK
      Created at PC:   SQRT_SERVER.LOOP$397.%LINE 424+71
      Parent task:     %TASK 1
      Start PC:        SQRT_SERVER.SQRT_TASK$TASK_BODY
      Task control block:              Stack storage (bytes):
        Task value:    161424            RESERVED_BYTES:        10640
        Entries:       1                 TOP_GUARD_SIZE:         5120
        Size:          1453              STORAGE_SIZE:           6144
      Stack addresses:                   Bytes in use:            792
        Top address:   0002BC00
        Base address:  0002D3FC        Total storage:          23357
```

6.9 Sample Remote Debugging Session

This example shows a typical debugging session for a VAXELN Ada program. It illustrates the purpose of system sessions and command sessions, and shows the use of remote debugger commands and VAX/VMS Debugger commands in obtaining VAXELN and VAX Ada information. It also illustrates system loading from the remote debugger.

The program in this example is the square-root server presented in Chapter 2. It shows how a simple network-wide multithread server can be implemented using VAXELN Ada and VAXELN kernel service routines.

The server uses a global VAXELN job port to receive VAXELN circuit requests for square-root calculations from other VAXELN jobs. When the server receives a request, it connects the requestor (using a VAXELN circuit) to a dedicated Ada server task that computes one or more square root values, depending on the requestor's needs. When a server task detects that its service is ended (the circuit with the requestor is disconnected), the task makes itself available for another requestor and more computations.

The square-root calculations are passed as VAXELN messages between the requestor and its Ada server task. Each message contains one floating-point value.

Example 6–1: Debugging a VAXELN Ada Program

```
❶ $DEBUG/REMOTE ARTHUR /LOAD=SQRT_SERVER.SYS

         VAXELN Remote Debugger  Version V1.0-00

  %RDEBUG-I-ATTEMPT_LOAD, Setting load file for node ARTHUR          1
          to TESTD:[ADA.EXE]SQRT_SERVER.SYS;
  %RDEBUG-I-TRIGGER, Triggering node ARTHUR
  %RDEBUG-I-ATTEMPT_CONNECT, Connecting to node ARTHUR               2
  %RDEBUG-I-RETRY_CONNECT, Retrying connect to node ARTHUR
  %RDEBUG-S-CONNECTION, Connected to node ARTHUR
  Job 4.1 (SQRT_SERVER) is waiting for your attention                3
  %RDEBUG-S-SET_TIME, System time set on node ARTHUR                 4
  RDBG*>                                                             5

❷ RDBG*>SHOW SYSTEM

    Job   Program     Priority   State     Shared Size  Readonly size
      2   XQDRIVER       1      waiting       31744        30720
      3   EDEBUGREM      3      running        4608        11776
      4   SQRT_SERVER   16      debug-wait     4608        93696
❸ RDBG*>SET JOB/CURRENT SQRT_SERVER
  %RDEBUG-I-SESSION_INIT, Loading symbols for Job 4. (SQRT_SERVER)
  -RDEBUG-I-FROM, from file TESTD:[ADA.EXE]SQRT_SERVER.EXE;1
  %RDEBUG-I-INITIAL, language is ADA, module set to 'SQRT_SERVER'
  %RDEBUG-I-NOTATMAIN, type GO to get to start of main program
  RDBG>
  RDBG>GO
  break at routine SQRT_SERVER
      53: procedure SQRT_SERVER is

❹ RDBG>STEP
  stepped to SQRT_SERVER.%LINE 66
      66:        entry SQRT_ENTRY(TASK_ARRAY_INDEX : TASK_INDEX);
  RDBG>STEP 9
  stepped to SQRT_SERVER.%LINE 330
     330: begin
❺ RDBG>EXAMINE NUM_ACTIVE_SERVER_TASKS
  SQRT_SERVER.NUM_ACTIVE_SERVER_TASKS:     0
❻ RDBG>SET BREAK /EVENT=ACTIVATING
❼ RDBG>GO                                                      1
  CTRL/C                                                       2
  RDBG*>CREATE JOB SQRT_SERVER_TESTER                          3
  RDBG*> RETURN                                              4
  break on ADA event ACTIVATING                                5
    Task %TASK 2 is about to begin its activation
    Its task body is at: SQRT_SERVER.SQRT_TASK$TASK_BODY

  RDBG>
```

(Continued on next page)

Example 6–1 (Cont.): Debugging a VAXELN Ada Program

```
❽ RDBG>

    task id     pri hold state  substate          task object
    %TASK 1      7       SUSP  Activating tasks  1740056
  * %TASK 2      4       SUSP  Activating        103424
❾ RDBG>

    Job   Program    Priority   State   Shared Size  Readonly size
  *  4   SQRT_SERVER   16      debug-wait   151040        93184

      Process          Priority   State   Stack Size   CPU time
         1                8     waiting       3584     00:00:00.68
    *    2               11     debug-wait    3072     00:00:00.07

❿ RDBG>
  RDBG*>
  [ARTHUR] SQRT_SERVER_TESTER - Sending initial message to SQRT_SERVER
  [ARTHUR] SQRT_SERVER_TESTER - Received:  1.3043814304E+19
  [ARTHUR] SQRT_SERVER_TESTER - Received:  3.6116221440E+09
  [ARTHUR] SQRT_SERVER_TESTER - Received:  6.0096773438E+04

      .
      .
      .
  [ARTHUR] SQRT_SERVER_TESTER - Received:  1.0000001192E+00
  [ARTHUR] SQRT_SERVER_TESTER - Successfully exiting
  [ARTHUR] SQRT_SERVER_TASK - Partner Exited.

⓫ RDBG*>
  _RDBG*>
  Job 6.1 (NEW_TESTER) is waiting for your attention
  RDBG*>
  Job 7.1 (NEW_TESTER) is waiting for your attention
⓬ RDBG*>
  RDBG*>
  [ARTHUR] SQRT_SERVER_TESTER - Sending initial message to SQRT_SERVER
  [ARTHUR] SQRT_SERVER_TESTER - Received:  1.3043814304E+19
  [ARTHUR] SQRT_SERVER_TESTER - Received:  3.6116221440E+09
  [ARTHUR] SQRT_SERVER_TESTER - Received:  6.0096773438E+04
  [ARTHUR] SQRT_SERVER_TESTER - Received:  2.4514643860E+02

      .
      .
      .
  [ARTHUR] SQRT_SERVER_TESTER - Received:  1.0000001192E+00
  [ARTHUR] SQRT_SERVER_TESTER - Successfully exiting
  [ARTHUR] SQRT_SERVER_TASK - Partner Exited.
```

(Continued on next page)

Example 6–1 (Cont.): Debugging a VAXELN Ada Program

```
⓭ RDBG*>  SET JOB/CURRENT -
  _RDBG*>  /IMAGE=[ADA.EXE]SQRT_SERVER_TESTER.EXE 6
  %RDEBUG-I-SESSION_INIT, Loading symbols for Job 6. (NEW_TESTER)
  -RDEBUG-I-FROM, from file TESTD:[ADA.EXE]SQRT_SERVER_TESTER.EXE;1
  %RDEBUG-I-INITIAL, language is ADA, module set to 'SQRT_SERVER_TESTER'
  %RDEBUG-I-NOTATMAIN, type GO to get to start of main program

⓮ RDBG>  SET JOB/CURRENT SQRT_SERVER
  %RDEBUG-W-NOTWAITING, Job 4 is not in a debug-wait state

⓯ RDBG>  SET JOB/HALT SQRT_SERVER
  Job 4.1 (SQRT_SERVER) is waiting for your attention

⓰ RDBG>  SET JOB/CURRENT SQRT_SERVER
  %RDEBUG-I-SESSION_INIT, Loading symbols for Job 4. (SQRT_SERVER)
  -RDEBUG-I-FROM, from file TESTD:[ADA.EXE]SQRT_SERVER.EXE;1
  %RDEBUG-I-INITIAL, language is ADA, module set to 'ADA$ELAB_SQRT_SERVER'

⓱ RDBG>  EXIT
  $
```

The following notes explain the events in Example 6–1.

❶ The DEBUG/REMOTE command invokes the remote debugger. The VAXELN system image SQRT_SERVER.SYS is loaded to the target node ARTHUR and then booted.

When you invoke the remote debugger, a series of messages appears. The following list explains these messages in the order in which they occur:

1. The /LOAD qualifier causes the specified VAXELN system image SQRT_SERVER.SYS to be loaded on ARTHUR. The system is booted after the system image is loaded.
2. Once the trigger request has been sent to the target node, the remote debugger attempts to connect to the target node. Depending on the size of the VAXELN system image, network traffic, and host-system activity, the remote debugger may have to repeat the connection attempt several times before the system boots.
3. The remote debugger reports the job number, name, and state of each job in the system that is in a debug-wait state. In this case, SQRT_SERVER has started under control of the debugger and is therefore in a debug-wait state.
4. If the system time on the target node has not been set, then it is set to the current time on the host.

5. The current session is the system session, as indicated by the prompt RDBG*> .

❷ The SHOW SYSTEM command displays an overview of the current state of the jobs in the system.

The XQDRIVER job is the network driver. The EDEBUGREM job is the remote debugger nucleus. The SQRT_SERVER job is running the application that is to be debugged. Since the SQRT_SERVER job is in a debug wait state, it is ready to start a command session.

❸ The SET JOB/CURRENT command begins a command session with the job SQRT_SERVER. The remote debugger reads the symbol table information from the image file associated with the job and prepares to accept commands that may be directed at that job.

The GO command begins job execution.

❹ The STEP command executes the program a line at a time. As shown in the second STEP command, it is possible to step over several lines with one command.

❺ The EXAMINE command displays the current value of one or more variables during program execution.

❻ The SET BREAK command selects locations for program suspension. You may also set breakpoints that will be triggered by various Ada events. In this example, the SET BREAK/EVENT=ACTIVATING command causes a breakpoint to be triggered when a task is about to begin its activation.

❼ The breakpoint may be triggered by the following sequence of commands:

1. The GO command causes the program to continue execution. When the program is running, the command session is not waiting to accept commands.
2. CTRL/C gains the attention of the system session. Once in the system session, you may enter commands.
3. The CREATE JOB SQRT_SERVER_TESTER command creates a job running a test program for the SQRT_SERVER.
4. Pressing the RETURN key while in the system session returns control to the command session.
5. Once SQRT_SERVER_TESTER begins execution, it connects to SQRT_SERVER and SQRT_SERVER creates a new task. So, SQRT_SERVER triggers a breakpoint on the Ada event ACTIVATING.

❽ The SHOW TASK/ALL command displays information about all currently existing tasks. In this example, there are currently two existing tasks, the environment task, %TASK 1, and the task that is currently being activated, %TASK 2.

Note that the SHOW TASK command presents information using the Ada categories of task and task priorities. You can obtain information on VAXELN processes and process priorities by means of the SHOW JOB command.

❾ The SHOW JOB/FULL command displays information about all existing processes in a VAXELN job. In this example, process 1 is the VAXELN process that is running the Ada task %TASK 1, and process 2 is the VAXELN process that is running the Ada task %TASK 2. The priorities listed are their VAXELN process priorities.

❿ The SET JOB/CONTINUE command causes a job this is in a debug-wait state to continue execution. This command causes Job 4 to continue execution as if a GO command were executed. Because the job was associated with the command session, the command session is ended and control is returned to the system session.

⓫ The CREATE JOB/LOAD command creates jobs running programs that were not part of the VAXELN system image.

The first CREATE JOB command loads the program image for the program SQRT_SERVER_TESTER from the node "180" to the target node and then creates a job running that program. The second CREATE JOB command creates another job running that same program; the /LOAD qualifier is not needed in this case because the program was already loaded on the system.

⓬ The SET JOB/CONTINUE command causes a job that is in a debug-wait state to continue execution.

This command causes Job 7 to continue execution as if a command session was started with that job and a GO command was executed. Since the VAXELN system was created without a console driver, all console output generated by the job SQRT_SERVER_TESTER is directed to your debugging terminal, as illustrated in the previous step.

⓭ The SET JOB/CURRENT command begins a new command session.

This command causes the command session with job SQRT_SERVER to end and a new command session with Job 6 (NEW_TESTER) to begin. The /IMAGE qualifier is used to direct the remote debugger to a local copy of the program image file for the program that the target job is running.

⓮ To begin a command session with a job that is not in a debug-wait state, you must first put the job in a debug-wait state using the SET JOB/HALT command.

The SET JOB/CURRENT command demonstrates that you cannot begin a command session with a job that is not in a debug-wait state.

⓯ The SET JOB/HALT command causes the SQRT_SERVER task to raise an asynchronous debugging exception, thus putting the job into a debug-wait state.

⓰ You may then begin a command session with the job using the SET JOB/CURRENT command.

⓱ Use the EXIT command to exit from the remote debugger.

This command causes an orderly termination from the remote debugger. All breakpoints, tracepoints, and watchpoints in the job associated with the command session are canceled. All jobs are left in whatever state they were in prior to executing the EXIT command.

Part III Run-Time Related Topics

This section presents information on run-time related topics. It discusses input-output, Ada tasking, and using VAXELN system services.

Chapter 7
Input-Output

This chapter explains the kinds of VAXELN files that are available for input-output operations. It also describes the use of file specifications in VAXELN Ada, and gives an explanation of how to control the characteristics of external files.

The VAXELN Ada predefined packages and their operations are implemented using VAXELN file organizations and facilities. VAXELN Ada supports the following VAX Ada predefined input-output packages for operations on sequential files:

- SEQUENTIAL_IO
- DIRECT_IO
- SEQUENTIAL_MIXED_IO
- DIRECT_MIXED_IO
- TEXT_IO

VAXELN Ada does not support the following VAX Ada predefined input-output packages:

- RELATIVE_IO
- RELATIVE_MIXED_IO
- INDEXED_IO
- INDEXED_MIXED_IO

All package specifications, as well as explanations of the operations provided by each package, are presented in the *VAX Ada Language Reference Manual*.

Before reading this chapter, you should be familiar with the Ada language and the VAX Ada predefined input-output packages, as described in the *VAX Ada Language Reference Manual*. You should also have some familiarity with VAX/VMS Record Management Services (RMS) file organizations and access methods. You should know how to work with VAX/VMS file specifications and directories, and have some familiarity with the VAX/VMS File Definition Language (FDL). The *VAX Ada Programmer's Run-Time Reference Manual* also contains more detailed information on input-output.

7.1 Files and File Access

VAXELN Ada supports only sequential files. Thus, VAX Ada packages that deal specifically with relative and indexed files (RELATIVE_IO, RELATIVE_MIXED_IO, INDEXED_IO, and INDEXED_MIXED_IO) are not supported. A VAXELN Ada program that attempts to use any of these packages will not link. An attempt to access a nonsequential file with the supported packages will raise the USE_ERROR exception.

Input-output operations can be performed on terminals, printers, and other devices. The VAX Ada predefined input-output packages can also be used to communicate with other nodes in a DECnet network, as well as with other VAXELN jobs using the VAXELN interjob communication services.

The following sections describe how external files can be accessed from a VAXELN Ada program.

7.1.1 Disk Files

The VAXELN Disk File Service is used whenever a program accesses a file on a disk device. The Disk File Service supports Files–11 On-Disk Structure Level 2 (ODS-2), which is compatible with the VAX/VMS file system. Therefore, disks and files can be exchanged between VAXELN and VAX/VMS systems.

When the application system boots, you can specify that certain disks be mounted automatically. The System Characteristics menu of the System Builder Utility (EBUILD) allows you to list the device names and gives you the option of listing the volume names. The first disk specified in the list of disks to be mounted automatically is considered the default disk. The default disk is used if a device name is not given in a file specification. If a disk is not specified to be mounted automatically, the first disk mounted by the application becomes the default disk. See Section 7.2.2 for more information on disk device names. You can also initialize, mount, and dismount disks by calls to the VAXELN Disk Utility procedures. See Chapter 9 for more information on the Disk Utility procedures.

You must explicitly define the disk devices to be used by your application by means of one or more Device Description entries in the System Builder menu. See Chapter 4 for more information on Device Description entries.

The Disk File Service is automatically included with your application unless you specify that it should not be included in the System Characteristics menu of EBUILD. The disk device drivers supplied with the VAXELN system are already linked with the Disk File Service. If you are using your own disk driver and want to use the Disk File Service, you must link your driver with the File Service. For more information, see the *VAXELN User's Guide.*

7.1.2 Tape Files

The VAXELN Tape File Service is used whenever a program accesses a file on a tape device. The Tape File Service supports Level 3 of the ANSI standard for magnetic tapes, and is compatible with the VAX/VMS system.

Tape volumes can be initialized, mounted, and dismounted by calls to the VAXELN Tape Utility procedures. See Chapter 9 for more information on the Tape Utility procedures.

You must explicitly define the tape devices to be used by your application by means of one or more Device Description entries in the System Builder menu. See Chapter 4 for more information on Device Description entries.

The Tape File Service is included with your application unless you specify that it should not be included in the System Characteristics menu of EBUILD. The tape device drivers supplied with the VAXELN system are already linked with the Tape File Service. If you are using your own tape driver and want to use the Tape File Service, you must link your driver with the File Service. For more information, see the *VAXELN User's Guide.*

7.1.3 Terminals and Printers

The VAXELN system provides device drivers for terminals and printers. These can be accessed just like files by specifying the appropriate device name in a file specification. You must explicitly include support for the terminal or printer interface device by means of one or more Device Description entries in the System Builder menu. You can also specify characteristics of individual terminal lines by means of Terminal Description entries in the System Builder menu. See Chapter 4 for more information on the System Builder Utility. See the *VAXELN User's Guide* for more information on terminal and printer devices.

Console support is included by default unless you specify that it should not be included in the Systems Characteristics menu of EBUILD. However, if console support is not present and the remote debugger is included, console input-output is automatically directed through the remote debugger. If you specify debug=both or debug=local, then console support is included, (even if you specified that it should not be included). See Chapter 6 for more information.

7.1.4 Network Files

A VAXELN system can function as an end node in a DECnet Phase IV network using Ethernet as the communications medium. The Network Service is included with your application by default unless you specify that it should not be included in the Network Node Characteristics menu of the System Builder Utility.

If the Network Service is present, you can access files on other DECnet nodes and perform DECnet task-to-task communication with other nodes. If the File Access Listener (which is the default), is also included, other DECnet nodes can access files on your VAXELN system. Section 7.2.1 provides information on how to specify remote files and DECnet task objects.

It is not necessary to define device support for the Ethernet interface device; support is automatically included if you include the Network Service. For more information on the Network Service, see the *VAXELN User's Guide*.

7.1.5 VAXELN Circuits

VAXELN circuits allow you to communicate with other jobs on VAXELN nodes and with DECnet tasks on other nodes in a DECnet network. While you can call VAXELN services directly to create and use circuits, VAXELN Ada provides a simple way of using Ada input-output to communicate by means of circuits. For more information on circuits, see the *VAXELN User's Guide*.

If the VAXELN_CIRCUIT attribute in the FILE section of a FORM string parameter to the CREATE or OPEN procedure is specified as YES, a circuit is used rather than a normal file. The CREATE procedure creates a circuit and waits for other processes to connect to the circuit. The OPEN procedure connects to an existing circuit. For both the CREATE and OPEN procedures, the name of the circuit is specified by the NAME parameter. The following example creates a circuit named MY_CIRCUIT.

```
CREATE  (FILE => CIRCUIT_FILE,
         MODE => INOUT_FILE,
         NAME => "MY_CIRCUIT",
         FORM => "FILE; VAXELN_CIRCUIT YES");
```

Once the connection is established, you can use any of the operations provided by the input-output packages to send and receive information over the circuit. The following packages support circuits:

- SEQUENTIAL_IO
- SEQUENTIAL_MIXED_IO
- TEXT_IO

The TEXT_IO package sends or receives each line as an individual message over the circuit. To connect a circuit to a job on another node, the Network Service must be included with your application. For more information on the capabilities of VAXELN interjob communication, see the *VAXELN User's Guide*. For an example of using VAXELN circuits, see the sample program file ELN$:SQRT_SERVER_TESTER.ADA on your system.

7.1.6 Other Devices

You can use the VAXELN Ada predefined input-output packages to access other devices if your device driver supports calls from the File Service.

VAXELN also provides device drivers for serial and parallel communications devices, which can be accessed using calls to VAXELN services. The following devices are supported by VAXELN:

- Serial devices

  DMF-32
  DHV11
  DZV11
  DLVJ1
- Parallel devices

  DMF-32 parallel port
  DRC11
- Real-time devices

  ADV11C interface
  AXV11C interface
  KWV11C programmable clock

For more information on device drivers, see the *VAXELN User's Guide*.

7.2 Naming External Files

VAXELN Ada supports the VAX/VMS file specification syntax except for hyphens that are not allowed in unquoted file specifications. However, there are a few differences in the use of some of the file specification fields. These differences are described in the following sections.

7.2.1 Node Names

If you have included the Network Service in your application, you can access files on other nodes connected to yours in a DECnet network. However, the VAXELN system does not support the use of node names for non-VAXELN nodes. You must use the node number for non-VAXELN nodes. To find your node number on a VAX/VMS node, type the following command at the DCL prompt ($):

```
$ SHOW NETWORK
```

The node number is specified as a decimal integer, for example:

```
$ 110::USER_DISK:[SMITH]ANALOG_DATA.DAT
```

If the remote node is not in the same DECnet area as your VAXELN node, specify the node address in the form "a.n" where "a" is the DECnet area number and "n" is the DECnet node number in that area. For example:

```
$ 2.110::USER_DISK:[SMITH]ANALOG_DATA.DAT
```

If you are accessing a file on another VAXELN node connected to yours by the Ethernet, you do not need to specify the node name if the remote node has created a universal name for a device on another VAXELN node. See Section 7.2.2 for more information.

7.2.2 Device Names

The first time a disk or tape device is mounted, the Name Service creates a universal name for the device. This name has the form DISK$name or TAPE$name, where "name" is the volume name of the disk or tape that was mounted. For example, if a disk with the volume name TEST was mounted, the universal name DISK$TEST would be created.

A universal name is available to all VAXELN nodes connected to your node by the Ethernet, so you do not need to specify a node name to access disks or tapes mounted on other nodes. If another disk or tape of the same volume name is mounted, a local name is created; a local name is only visible on the node where it was created. You can use universal names as device names in file specifications in the NAME parameter to the OPEN and CREATE procedures. For example:

```
OPEN (FILE => F,
     MODE => IN_FILE,
     NAME => "DISK$TEST:[TEST_DIR]TEST_DATA.DAT");
```

You can supply a list of disk device names and volume names in the Systems Characteristics menu of EBUILD. Devices in this list are mounted automatically by the File Service. If you include a volume name for a device in this list, it must match the actual volume name. In addition, the name DISK$DEFAULT_VOLUME is created for the first disk in this list. This disk is then used if you do not specify a device in a file specification, and is also used for temporary files. If a disk is not specified to be mounted automatically, then the first disk mounted by the application is considered the default volume. See the *VAXELN User's Guide* for more information.

You can also use the physical device name in file specifications. The physical device name is composed of the controller name (as specified in a Device Description menu in EBUILD) and the unit number. For example, if you have a device description for device DUA, the device name for unit 0 on DUA would be DUA0:. The device name for the console is CONSOLE:. For more information on device descriptions, see Chapter 4 of this manual, as well as the *VAXELN User's Guide*.

7.2.3 Directory Name

If you do not include a directory name in a file specification, [000000] is the default directory.

7.2.4 File Type

VAXELN Ada does not provide a default for the file type. The DEFAULT_NAME attribute in the FILE section of a FORM string is not supported. If a file type is not specified in a file argument, then a default is not given. For example, if you create the file [SMITH]TEST, TEST. would be the file name; it would not have a type.

7.3 Standard Input-Output Files

The default file specification for the standard input and standard output files is "CONSOLE:". You can supply an alternate file specification for standard input by means of the first program argument in the Program Description menu of EBUILD. If you specify a nonnull string as the first program argument, that string is used as the file specification for standard input. Similarly, the second program argument, if nonnull, is used as the file specification for standard output.

If a VAXELN Ada program generates an unhandled exception, the message for that exception is displayed by the VAXELN Ada run-time library. By default, the file specification used for exception messages is "CONSOLE:"; however, if you specify a nonnull string as the third program argument, that string is used instead.

The convention of using the first, second, and third program arguments for the standard input, output, and error files is also observed by VAXELN Pascal and VAXELN C programs.

7.4 Specifying External File Attributes with the File Definition Language (FDL)

VAXELN Ada programs can specify the system-dependent attributes of an external file using the FORM parameter of the CREATE and OPEN procedures. VAXELN Ada uses a subset of the VAX RMS File Definition Language (FDL), which is used by VAX Ada programs under the VAX/VMS system. VAXELN Ada also supports an additional attribute, VAXELN_CIRCUIT, which is not supported by VAX Ada. See Section 7.4.2 for more information on the VAXELN_CIRCUIT attribute.

VAXELN Ada does not support the ability to specify the name of a separate file containing FDL statements by the use of an initial at sign character (@) in the value of the FORM parameter.

When using FDL to specify the attributes of an Ada external file, you should observe the following FDL rules:

- The primary attributes must appear in the order shown in Table 7–1.
- Each attribute string (primary or secondary) constitutes an FDL statement, and must be terminated with a semicolon. For example:

```
-- Create SOME_FILE.DAT with fixed record format and
-- have it submitted to the print queue when the file
-- is closed.
--
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT",
       FORM => "RECORD; FORMAT FIXED; SIZE 120;");
```

  Note that the exclamation point is the comment character in FDL, and anything following it is ignored. For example:

```
-- Create SOME_FILE.DAT with fixed record format
--
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT",
       FORM => "RECORD; FORMAT FIXED; !fixed-size records");
```

- Each FDL statement can represent only one primary or secondary attribute and its associated value. Each statement can have no more than a total of 132 characters (including blanks). To format your program without adding extra blanks to the form string, you can use the Ada concatenation operator (ampersand) to break up the form string into individual statement strings. Thus, the first example could be rewritten as follows:

```
CREATE(FILE => MY_FILE,
       MODE => OUT_FILE,
       NAME => "SOME_FILE.DAT",
       FORM => "RECORD;"          &
               "FORMAT FIXED;"    &
               "SIZE 120;"         );
```

- Keywords can be truncated to their shortest unique abbreviations. Strings must be enclosed in either a pair of apostrophes or a pair of double quotation marks. Note that Ada-based integers or integers with underscores are not legal FDL syntax.

The following sections include a table describing the primary and secondary attributes, a detailed description of the VAXELN_CIRCUIT attribute, and a list of the most commonly used FDL attributes. For more information on the use of the FORM parameter, see the *VAX Ada Programmer's Run-Time Reference Manual.*

7.4.1 FDL Primary and Secondary Attributes

FDL statements in a VAXELN Ada form string specify predefined VAXELN file attributes. *Primary attributes* take a single value or represent a group of related or *secondary attributes,* which also take values. (Most of the primary attributes that have secondary attributes do not themselves take values.) Table 7–1 lists the available primary and secondary attributes.

Table 7–1: FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
TITLE	Gives a title to the FDL file; for comment purposes only.	None
IDENT	Gives the date and time of creation of the FDL file, and specifies the name of the creating utility; treated as a comment.	None
SYSTEM	Takes no value. Secondary attributes give system identification information.	DEVICE, SOURCE, TARGET
FILE	Takes no value. Secondary attributes determine file characteristics: owner, organization, protection, and revision; what will happen when the file is opened or closed; whether data checking will be done when the file is read or written; what kind of processing is allowed; how much space is allocated for the file and whether the space is contiguous; and so on. Secondary attributes also allow specification of magnetic tape file operations. The value given for NAME in a form string is ignored by VAXELN Ada.	ALLOCATION, BEST_TRY_CONTIGUOUS, CONTIGUOUS, CREATE_IF, DELETE_ON_CLOSE, DIRECTORY_ENTRY, EXTENSION, MAXIMIZE_VERSION, MT_CLOSE_REWIND, MT_CURRENT_POSITION, MT_NOT_EOF, MT_OPEN_REWIND, NAME, ORGANIZATION, OWNER, PROTECTION, READ_CHECK, SEQUENTIAL_ONLY, SUPERSEDE, TEMPORARY, TRUNCATE_ON_CLOSE, USER_FILE_OPEN, VAXELN_CIRCUIT, WRITE_CHECK
RECORD	Takes no value. Secondary attributes specify the following characteristics of records in the file: size, kind of carriage control, and format.	BLOCK_SPAN, CARRIAGE_CONTROL, CONTROL_FIELD, FORMAT, SIZE

Table 7–1 (Cont.): FDL Primary and Secondary Attribute Descriptions

Primary Attribute	Function	Secondary Attributes
ACCESS	Takes no value. Secondary attributes specify the file-processing operations.	GET, PUT, TRUNCATE
SHARING	Takes no value. Secondary attributes specify whether multiple readers or writers can concurrently access the file.	GET, PROHIBIT, PUT, USER_INTERLOCK
CONNECT	Takes no value. Secondary attributes specify run-time features and operations related to record access and performance.	END_OF_FILE, NOLOCK, READ_AHEAD, TRUNCATE_ON_PUT, UPDATE_IF, WRITE_BEHIND

7.4.2 VAXELN_CIRCUIT Attribute

VAXELN Ada supports an additional attribute called VAXELN_CIRCUIT in the FILE section of the FDL language. Any two ports (usually in different jobs) can be bound into pairs called *circuits*. The VAXELN_CIRCUIT attribute specifies whether or not the program wants to access a VAXELN circuit rather than an actual file. After the OPEN or CREATE procedure, all READ operations read from the circuit, and all WRITE operations write to the circuit. Circuits can be used only with packages SEQUENTIAL_IO, SEQUENTIAL_MIXED_IO, and TEXT_IO.

You can specify the following values for VAXELN_CIRCUIT:

YES	Specifies that the OPEN or CREATE procedure is to access or create a named circuit, where the name is supplied by the NAME parameter. Circuits are given universal names, and have a limit of four messages waiting in the circuit at any one time; an attempt to exceed that limit causes the sender to wait until space is available.
NO	Specifies that the OPEN or CREATE procedure is to access a file, not a circuit. No is the default.

7.4.3 Commonly Used FDL Attributes

Table 7–2 describes the primary and secondary FDL attributes that you are most likely to use in a VAXELN Ada program, and gives their default values. The table provides a quick reference and summarizes information presented in the *VAX/VMS File Definition Language Facility Reference Manual*.

As shown in the table, the value assigned to an attribute can take one of the following forms:

Switch	A logical value, set to TRUE, YES, FALSE, or NO. TRUE (or YES) sets the attribute; FALSE (or NO) clears it. (You can also use the abbreviations T, Y, F, and N.)
Keyword	A word that you must type (in either uppercase or lowercase) after the attribute name. You can truncate a keyword to its shortest unique abbreviation.
String	A character string (enclosed in either a pair of apostrophes or a pair of double quotation marks) that you must type after the attribute name. The null string is a valid string value.
Integer	A decimal integer (based integers or underscores are not allowed).

Table 7–2: Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
TITLE	String of up to 132 characters, including the TITLE keyword. No default value.	Comment

Table 7-2 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
IDENT	String of up to 132 characters, including the IDENT keyword. Default value is the date, time of creation, and name of the creating utility if created with EDIT/FDL or ANALYZE/RMS_FILE; otherwise, no default value.	
SYSTEM		
DEVICE	String. Default value is null.	Comment (names the disk model on which the file will reside).
FILE		
ALLOCATION	Integer in the range of 0 to 4294967295. Default value is 0.	Sets the number of blocks that will be initially allocated for the file. If 0, the system will not preallocate space for the file.
BEST_TRY_CONTIGUOUS	Switch. Default value is NO.	Controls whether the file will be allocated contiguously if there is enough space. If set to YES, and there is enough space for the file, the file will be allocated contiguously; if there is not enough space, the file will not be allocated contiguously. If set to NO, this attribute is ignored.
CONTIGUOUS	Switch. Default value is NO.	Controls whether the file must be allocated contiguously. When set to YES and there is not enough space for the file's initial allocation, an error message results. When set to NO, the attribute is ignored.

Table 7–2 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
EXTENSION	Integer in the range of 0 to 65535. Default value is 0.	Sets the number of blocks for the default extension value for the file. Each time the file is extended, the specified number of blocks is added. If 0, the extension size is determined by the system each time the file must be extended.
ORGANIZATION	Keyword. Default value is SEQUENTIAL.	Defines the type of file organization. The keyword must be SEQUENTIAL for VAXELN Ada.
PROTECTION	String. Default value is the process default.	Defines the levels of file protection. Its value can take one of two forms (SYSTEM=code, OWN=code, GROUP=code, WORLD=code) or (SYSTEM:code, OWN:code, GROUP:code, WORLD:code) where the code is a protection specification for READ, WRITE, EXECUTE, and DELETE in the form RWED. To deny a specific access right, you omit it from the code. To give no access rights to a user classification, you omit the classification from the list. For example, the following string gives all access rights to SYSTEM and OWNER, gives only READ access to GROUP, and gives no access rights to WORLD: (SYSTEM=RWED, OWNER=RWED, GROUP=R).

Table 7–2 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
SEQUENTIAL_ONLY	Switch. Default value is NO.	Indicates that the file can be processed only sequentially, thus allowing certain processing optimizations. Any attempt to perform random access results in an error.
RECORD		
CARRIAGE_CONTROL	Keyword. Default value is CARRIAGE_RETURN.	Specifies the type of carriage control for the records in the file. Must be one of the keywords CARRIAGE_RETURN, FORTRAN, NONE, or PRINT. See the *Vax Ada Programmer's Run-Time Reference Manual* and the *VAX/VMS File Definition Language Facility Reference Manual* for more information.
FORMAT	Keyword. Default value is VARIABLE.	Sets the record format for the data file. Must be one of the keywords FIXED, STREAM, STREAM_CR, STREAM_LF, UNDEFINED, VARIABLE, VFC. See the *VAX/VMS File Definition Language Facility Reference Manual* for more information.
SIZE	Integer. No default value.	Sets the maximum record size in bytes. With fixed-length records, this value is the length of every record in the file. With variable-length records, this value is the length of the longest record that can be placed in the file.

Table 7–2 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
		You can specify 0 and the system will not impose a maximum record length. If the records are variable with fixed-control (VFC), the fixed control portion of the record is not included in the SIZE calculation; only the data portion is set by this attribute. The fixed area is the size in bytes of the fixed-control portion of VFC records. Regular variable-length records have a fixed-control size of 0. See the *VAX/VMS File Definition Language Facility Reference Manual* for the maximum sizes allowed for the various record organizations and formats.
ACCESS		
GET	Switch. Default value is GET when using an OPEN procedure and if no other ACCESS secondary attribute has been specified.	Permits VAXELN GET or FIND operations.
PUT	Switch. Default is PUT when using a CREATE procedure.	Permits VAXELN PUT or EXTEND operations.
TRUNCATE	Switch. Default value is FALSE.	Permits VAXELN TRUNCATE operations.
SHARING		
GET	Switch. Default is TRUE if ACCESS GET has also been specified.	Allows other users to read the file.

Table 7–2 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
PROHIBIT	Switch. Default is YES if ACCESS DELETE, ACCESS PUT, ACCESS TRUNCATE, or ACCESS OPERATE has been specified.	Prohibits any type of file sharing by other users. When set to YES, this attribute takes precedence over all other ACCESS secondary attributes.
PUT	Switch. No default value.	Allows other users to write records to the file.
CONNECT		
READ_AHEAD	Switch. No default value.	Indicates read-ahead operations; to be used with multiple buffers. When one buffer is filled, the next record is read into the next buffer while the input-output operation takes place for the first buffer. Since the system does not have to wait for input-output completion, input and computing can overlap. This attribute is ignored for DECnet operations. See the *VAX/VMS File Definition Language Facility Reference Manual* for more information.
TRUNCATE_ON_PUT	Switch. No default value.	Specifies that a VAXELN PUT or WRITE operation can occur at any point in a file, truncating the file at that point. A WRITE operation causes the end of file mark to immediately follow the last byte written. This attribute can be used only with VAXELN sequential files.

Table 7–2 (Cont.): Commonly Used FDL Attributes

FDL Attributes	Type of Value and Default	Function
UPDATE_IF	Switch. No default value.	Indicates that if a PUT operation is specified for a record that already exists in the file, the operation is converted to an update.
WRITE_BEHIND	Switch. No default value.	Indicates that write-behind operations are to occur when multiple buffers are used. When one buffer is filled, the next record is written into the next buffer while the input-output operation takes place for the first buffer. Since the system does not have to wait for input-output completion, computing and output can overlap. See the *VAX/VMS File Definition Language Facility Reference Manual* for more information.

Certain FDL attributes can significantly improve application performance; that is, if the files used by the application are designed and tuned properly, the application will run more efficiently, often because a minimum number of input-output operations occur. File design and tuning are important for large files. The file characteristics specified when a file is created often have a significant effect on application performance at run time.

The following attributes can affect application performance:

FILE ALLOCATION
FILE BEST_TRY_CONTIGUOUS
FILE CONTIGUOUS
FILE EXTENSION
FILE SEQUENTIAL_ONLY
CONNECT READ_AHEAD
CONNECT WRITE_BEHIND
ACCESS and SHARING attributes

For additional information on file applications, refer to the *Guide to VAX/VMS File Applications.*

Chapter 8 Tasking

The use of Ada tasks in the VAXELN environment is very similar to the use of Ada tasks in the VAX/VMS environment. However, there are differences in the way VAXELN Ada tasks are implemented. This chapter provides information on those differences.

Note that the structure of this chapter parallels the structure of the tasking information provided in the VAX Ada documentation set. If you are not familiar with VAX Ada tasking, you should first read Chapter 9 of the *VAX Ada Language Reference Manual* and review the tasking information given in the *VAX Ada Programmer's Run-Time Reference Manual.* For more information on VAXELN concepts, see the *VAXELN User's Guide.*

8.1 Introduction to Using Tasks on VAXELN

Because both VAXELN Ada and VAX Ada conform to the Ada language standard, they both provide the same tasking mechanisms, and those mechanisms involve the same principles:

- Tasks are defined as entities whose execution proceeds (conceptually) in parallel.
- A main, or environment, task is automatically created whenever a main program is run; this task first elaborates any library packages associated with the program, and then calls the main program. When execution of the main program is completed, and all tasks that depend on its library packages terminate, the main task is terminated, and the job is deleted.

- Any task is said to depend on a number of masters (blocks, tasks, subprograms, or library packages). An immediate master is the master that immediately contains the declaration of a task object, or that immediately contains the definition of the access type whose designated type is a task type. Control cannot leave a master until all of its dependent tasks have terminated.
- Whenever a task is created, the VAXELN Ada run-time library creates a task control block to manage the task; whenever a task is activated, the VAXELN Ada run-time library creates a process and a stack for the statements, which the task will execute.
- Synchronization (in Ada) is accomplished with the rendezvous mechanism.

The SQRT_SERVER example in Chapter 2 shows the use of Ada tasks in a VAXELN application; other tasking examples are given in the *VAX Ada Language Reference Manual* and the *VAX Ada Programmer's Run-Time Reference Manual.*

Both VAXELN Ada and VAX Ada implement parallel tasks with interleaved execution on a single physical processor. However, other task implementation details are different, and they mirror the differences between the VAX/VMS operating system and the VAXELN kernel. Table 8–1 presents these conceptual differences.

Table 8–1: Comparison of VAX/VMS and VAXELN Ada Task Implementations

Ada Entity	VAX/VMS Equivalent	VAXELN Equivalent
Ada program	Image executing in the context of a VAX/VMS process.[1]	A job, set up by a call to the VAXELN CREATE_JOB procedure.[2] (The call can be either implicit, when the system is booted, or explicit.)

[1]When a VAX Ada program terminates, control returns to the command line interface (CLI), if there is one.

[2]When a VAXELN Ada program terminates, the job terminates.

Table 8–1 (Cont.): Comparison of VAX/VMS and VAXELN Ada Task Implementations

Ada Entity	VAX/VMS Equivalent	VAXELN Equivalent
Main (environment) task	Created and managed by the Ada run-time library task scheduler in the context of the process in which the program is executing.	A VAXELN master process, created and managed by the VAXELN kernel.
	Stack expands automatically as the program executes, and is allocated in P1 space of the process in which the program is executing.	Stack has a fixed size, and is allocated in P0 space.[3]
Task	Created and managed by the Ada run-time library task scheduler in the context of the process in which the program is executing.	A VAXELN process within the job (subprocess). Created by a VAXELN Ada run-time library call to the VAXELN CREATE_PROCESS procedure, and managed by the VAXELN kernel.
	Stack has a fixed size, and is allocated in P0 space.	Stack has a fixed size, and is allocated in P0 space.[3]

[3]VAXELN process stacks are normally allocated in P1 space. However, when a VAXELN process is used to implement an Ada task, a stack is immediately created in P0 space for the task, and a stack switch is made. (This switch permits each task to address variables in any other task's stack in the same Ada program.)

8.2 Task Storage Allocation

In the VAXELN environment, as in the VAX/VMS environment, each task created in your Ada program requires a certain amount of storage. The following sections explain how to control the amount of storage allocated for VAXELN Ada tasks.

You should note that, unlike the VAX/VMS operating system, the VAXELN executive is a nonpaging virtual memory system: every virtual memory page is associated with a physical memory page. Thus, all VAXELN Ada tasks are always memory resident, and the number of tasks you can create is limited by the amount of physical storage available on your system.

You should also note that the number of tasks you can create may be further limited by the amount of virtual address space available. You can select the amount of available virtual address space with the *Virtual size* option on the System Builder Edit System Characteristics menu when you build your system image; see Chapter 4 for more information.

8.2.1 Storage Created for a Task Object—the Task Control Block

In VAXELN Ada, as in VAX Ada, a task control block is allocated whenever a task object is created. This task control block keeps track of its task's execution, and is deallocated when control leaves the task's immediate master (not when the task terminates).

The size of a VAXELN Ada task control block depends on the characteristics of the task's type. In other words, the size increases in proportion to the number of single entries in the task type and the total number of members of all its entry families (note that a main task has no entries, so main task control blocks have a constant size).

Estimating the size of a task control block is the same in VAXELN Ada as it is in VAX Ada. However, because VAXELN does not have an asynchronous trap mechanism, AST entries do not have to be accounted for. You can do the estimation with the following formula:

```
TCB_SIZE (pages) = (FIXED_AMOUNT + E*12.2)/512
```

where

FIXED_AMOUNT	=	3000
E	=	the number of single entries plus the number of members in all entry families

For most task types (that is, those having fewer than a few hundred total entries), the storage consumed by the task control block is relatively small. You can reduce the size of the task control block by reducing the number of entries and the number of entry family members. You can also reduce the total accumulated amount of task storage by arranging for control to leave the immediate masters of any terminated tasks. In so doing, you cause the storage consumed by the terminated tasks' control blocks to be released. See the *VAX Ada Programmer's Run-Time Reference Manual* for an example and for further discussion.

8.2.2 Storage Created for a Task Activation—the Task Stack

In both VAX Ada and VAXELN Ada, a task stack is allocated each time an Ada task is activated; the task stack is deallocated as soon as the task is terminated.

In VAXELN Ada, a separate VAXELN process is created for a task when the task is activated; all task stacks, including the stack for the main task, are allocated in P0 space. The main task stack has a fixed size. (This implementation differs from the VAX Ada implementation, where multiple tasks in an Ada program are created in the context of a single VAX/VMS process. The stack for a VAX Ada main task is adjustable and is allocated in P1 space; the stacks for all other tasks have fixed sizes and are allocated in P0 space.)

The task stack allocated for any VAXELN Ada task (including the main task) has two areas: a *working storage area* and a *top guard area*. The working area is used during normal task execution to store variables, call frames, and so on. The top guard area is a set of pages (512 bytes per page) at the top of the stack. These pages are inaccessible to your program—that is, attempts to read or write them will cause a hardware access violation (ACCVIO), which will usually terminate the process (task) immediately. The purpose of these guard pages is to help you detect accidental overflow of the working area of the stack. Accidental stack overflow can occur, for example, when a task executes non-Ada code for which stack checking is not performed or when storage size checks are suppressed when you compile the program.

Unless you specify otherwise, the size of the working area and the top guard area of all task stacks are set by the VAXELN Ada run-time library. By default, the working area is set to 60 pages, and the top guard area is set to 10 pages.

The default stack allocation for all tasks allows an additional 10 pages of stack for calls to non-Ada routines, which is adequate for most routines, including VAXELN kernel routines.

As in VAX Ada, you may need to specify the sizes of a VAXELN Ada task's stack areas for a number of reasons:

- You may find that the task or main program is raising the exception STORAGE_ERROR, and you want to increase its working area.
- You may find that the task does not need all of its default stack allocation, and you may wish to reduce the working area so that the unused storage can be put to other use by your program.
- You have not called any non-Ada routines, and you are not having any stack overflow (you have not suppressed checks and STORAGE_ERROR is not being raised). Thus, you may wish to decrease the top guard area and put the storage to other use.
- You may suspect that some non-Ada routine might be overflowing the stack, and you may wish to increase the top guard area in an attempt to detect the overflow.

To control the storage allocated for main task stacks, you can use pragma MAIN_STORAGE; to control the storage allocated for all other task stacks, you can use the STORAGE_SIZE length representation clause and pragma TASK_STORAGE. Note, however, that because VAXELN virtual pages are always memory-resident, you should be careful about setting excessive working storage and top guard areas.

The following sections describe how to work with task stacks.

8.2.2.1 Controlling the Size of a Main Task Stack

Main task stacks have a fixed size in the VAXELN environment. However, VAX Ada provides the pragma MAIN_STORAGE to allow you to control the size of a main task stack. (Note that in the VAX/VMS environment, pragma MAIN_STORAGE forces the main stack to be allocated as it is in the VAXELN environment: with a fixed size and in P0 space. Thus, you can use this pragma to simulate the behavior of a VAXELN main task when you are working with VAX Ada on a VAX/VMS target.)

The following example shows the use of pragma MAIN_STORAGE to decrease the stack allocation for a main task (TRY_MAIN).

```
procedure TRY_MAIN is
   -- Cut down the stack size by allocating four pages
   -- of working storage, and no guard pages
   --
   pragma MAIN_STORAGE(WORKING_STORAGE => 4,
                       TOP_GUARD => 0);

   type NUM_ARRAY_TYPE is array (1..10) of INTEGER;
   NUM_ARRAY: NUM_ARRAY_TYPE := (others => 0);
   I: INTEGER;
begin
   for I in NUM_ARRAY'RANGE loop
       . . .
   end loop;
end;
```

The syntax and rules for using pragma MAIN_STORAGE are given in Appendix C.

8.2.2.2 Controlling the Size of Other Task Stacks

To control the size of other tasks' working storage areas, you can use the STORAGE_SIZE length representation clause. For example, the attribute

```
for SQRT_TASK'STORAGE_SIZE use 12*512;
```

sets the working storage size for the task type SQRT_TASK to 12*512 bytes (that is, 12 physical pages).

If you specify a size of zero (bytes), the default stack size is used. Regardless of what size you specify, at least 10 pages of additional space are allocated for task management purposes. See Chapter 13 of the *VAX Ada Language Reference Manual* for a description of the syntax and use of STORAGE_SIZE.

You can determine the amount of storage you need for the stack working area by executing your program with the VAXELN Remote Debugger, and using the task debugging features for examining changes in stack storage as the program executes (see Chapter 6 for more information).

To control the stack's top guard area, you can use pragma TASK_STORAGE. For example, the statement

```
pragma TASK_STORAGE(SQRT_TASK, 0)
```

sets the top guard area of the task type SQRT_TASK to zero. See Chapter 13 of the *VAX Ada Language Reference Manual* for a description of the syntax and use of pragma TASK_STORAGE.

8.2.3 Stack Overflow and Non-Ada Code

In both VAXELN Ada and VAX Ada, you are protected from stack overflow in your Ada program: the exception STORAGE_ERROR is raised when an attempt is made to overflow either the main stack or an Ada task stack. In addition, the default 10-page stack storage allocated for each non-Ada call should be adequate protection against stack overflow for most non-Ada routine calls. However, you should be aware that non-Ada routines and kernel routines do not check for stack overflow. Thus, when you call a non-Ada routine from an Ada program, it could be possible that the stack of the main task or an individual task would overflow, and the Ada program would not be able to detect it because the exception STORAGE_ERROR would not be raised. Such an undetected stack overflow could result in random changes to various locations beyond the storage allocated for the stack. Because the correct operation of the Ada program may depend on such locations, undetected stack overflow could make your program erroneous.

Thus to be safe, you should not mix VAXELN Ada and non-Ada programs without checking for stack overflow. You can use the top guard areas of tasks in your program to detect if a non-Ada routine causes the stack to overflow. (See Section 8.2.2 for information about the top guard area.) If you make the size of the guard pages in the top guard area large enough, then undetected overflows that are not larger than the guard pages will raise a hardware access violation (ACCVIO) exception. In most cases, this exception will terminate your VAXELN process (task) immediately.

The VAXELN Remote Debugger can be used to detect stack overflow. The remote debugger performs an automatic stack check for you, and you can have it display the amount of stack space in use in any task, including the main task. For further information, see Chapter 6.

8.3 Task Switching

In the context of a single Ada program, or job, VAXELN Ada tasks are implemented to meet the following language requirement: when two tasks are eligible for execution, and they have different priorities, the lower priority task will not be executing while the higher priority task is ready and waiting to execute. Switching between tasks of equal priority normally takes place only when the running task becomes suspended (for example, a delay, an entry call, an input-output request, an interaction with the VAXELN service that causes a wait, and so on). In other words, Ada task scheduling is first-in-first-out (FIFO).

In the context of a VAXELN system composed of Ada programs (or Ada programs mixed with programs written in VAXELN Pascal or C), tasks are treated as any other VAXELN processes. VAXELN process scheduling is primarily FIFO. (See the *VAXELN User's Guide* for detailed information on how process scheduling works.)

VAXELN does not have a time-slicing mechanism, and therefore VAXELN Ada does not provide the VAX Ada pragma TIME_SLICE. Thus, the only means of modifying task switching behavior in VAXELN Ada is to change individual task priorities. You can use the Ada pragma PRIORITY to change task priorities, as in the following example:

```
task type SQRT_TASK is
   entry SQRT_ENTRY(TASK_ARRAY_INDEX : TASK_INDEX);
   pragma PRIORITY(4);
end;
```

The range of possible VAXELN Ada task priorities is the same as for VAX Ada: from 0 to 15 (low to high). Because VAXELN processes have priorities ranging from 15 to 0 (low to high), the relationship between the priority of an Ada task and the priority of the VAXELN process that implements it is defined by the following formula:

```
process_priority = 15 - task_priority
```

Tasks specified without pragma PRIORITY have a default midrange priority of 7 (and a VAXELN process priority of 8).

Note that you should use only pragma PRIORITY to set the priorities of VAXELN Ada tasks because the following VAXELN methods of setting priorities do not apply to Ada tasks:

- The *Process priority* option in a System Builder program descriptions menu (this option is ignored for VAXELN Ada tasks, including main tasks).
- The SET_PROCESS_PRIORITY procedure (the VAXELN Ada run-time library tends to reset task priorities according to Ada rules).

The syntax and use of pragma PRIORITY are described in detail in Chapter 9 of the *VAX Ada Language Reference Manual.*

8.4 Special Tasking Considerations

Use of tasks in an Ada program requires some care, because, like any other language construct, tasking has its own characteristic set of programming pitfalls: deadlock, busy waiting, abort statements, shared variables, reentrancy, and so on. Information on these general topics is given in the *VAX Ada Programmer's Run-Time Reference Manual*. You should bear in mind the following differences:

- There is no CTRL/Y mechanism in the VAXELN environment, so all cautions about interrupting an Ada program with CTRL/Y can be ignored (and the VAX Ada predefined package CONTROL_C_INTERCEPTION is not provided for VAXELN Ada).
- Because the VAXELN executive does not use the asynchronous system trap (AST) mechanism, VAXELN Ada does not provide the AST_ENTRY attribute and pragma.
- All VAXELN kernel services and utilities are fully reentrant.
- Use of VAXELN semaphores, mutexes, events, and areas from Ada tasks can introduce potential waits not under the control of Ada.

8.5 Calling VAXELN Kernel Services from Tasks

All VAXELN kernel services and utility routines are available from tasks (see Chapter 9 for information on how to call them). In general, these services and routines do not interfere with program execution, or the execution of other tasks in the program. In other words, if a wait is involved in a call from a VAXELN Ada task to a VAXELN service or routine, the wait does not block the job in which the task is executing: only the calling task waits until the service or routine call is completed.

8.6 Measuring and Tuning Tasking Performance

When you use tasks in your program, you must frequently trade off between responsiveness and throughput. Responsiveness is how fast a task responds to an asynchronous event, such as a user typing at a keyboard. Throughput is how much useful work, as measured by CPU time, a program accomplishes in a given amount of elapsed time (time spent switching tasks is overhead and takes CPU cycles that could be used for useful work).

Because time-slicing is not available with VAXELN Ada, the only way to increase responsiveness (rather than throughput) is to assign a higher priority to a task. Assigning a higher priority to some task invariably means that the program will perform more task switches—every time the high priority task becomes eligible for execution, Ada rules require that it displace a currently running lower priority task.

To help you measure the effects of any changes you may make to improve tasking performance, the VAXELN Remote Debugger provides the commands SHOW TASK/STATISTICS and SHOW TASK/FULL. See Chapter 6 for more information.

Chapter 9

Calling VAXELN Services

VAXELN system services and utilities can be called using the VAX Ada package VAXELN_SERVICES. Package VAXELN_SERVICES provides type definitions for VAXELN types and specifications for VAXELN kernel services. It also defines interfaces to VAXELN system services, utility routines, and device drivers and can be considered analogous in function to package STARLET in VAX Ada.

You can obtain the complete package specification by using the ACS EXTRACT SOURCE command.

The specification of data types is presented in Appendix A along with Ada-language procedure specifications, which are the interfaces for calling VAXELN services, and detailed argument descriptions for each procedure.

This chapter discusses VAXELN objects, the use of strings with VAXELN Ada, and presents a summary of VAXELN services grouped by function.

9.1 Objects in VAXELN Services

VAXELN 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' execution.

VAXELN objects are AREA, DEVICE, EVENT, MUTEX, MESSAGE, NAME, PORT, PROCESS, and SEMAPHORE. The following sections summarize the function and properties of each of these objects. See the *VAXELN User's Guide* for a complete discussion of objects.

9.1.1 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 has the following associated properties:

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

Operations with AREA Values

AREA values are used in the following operations:

CREATE_AREA	Creates an area or maps an existing area into the creating job's virtual address space.
DELETE_AREA	Deletes an area from an existing application and unmaps the data from its address space.
SIGNAL_AREA	Signals that a referencing process is finished with its exclusive access to an area. This allows the next waiting process to gain explicit access. Use this only if an area is "locked" by a process.
WAIT_ALL or WAIT_ANY	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.

9.1.2 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.

A DEVICE object has the following associated properties:

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

Operations with DEVICE Values

DEVICE values are used in the following operations:

CREATE_DEVICE	Establishes a connection between a physical device, a program, and an interrupt service routine.
DELETE_DEVICE	Removes a 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 become invalid.
SIGNAL_DEVICE	Signals a DEVICE object from an interrupt service routine.
WAIT_ALL or WAIT_ANY	Causes a wait until the device is signaled from an interrupt service routine. The DEVICE object is cleared when the wait is satisfied.

9.1.3 EVENT Object

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

An EVENT object has the following associated properties:

- A state of being either signaled or cleared

- A list of processes waiting for the event to be signaled

Operations with EVENT Values

EVENT values are used in the following operations:

CLEAR_EVENT	Sets the state of an EVENT object to cleared.
CREATE_EVENT	Creates and initializes an event with an initial state of signaled or cleared returning the EVENT value that identifies the event.
DELETE_EVENT	Removes an EVENT object from the system. When an event is deleted, any waiting processes are removed from their wait states immediately.
SIGNAL_EVENT	Sets the state of an EVENT object to signaled.
WAIT_ALL or WAIT_ANY	Causes a wait until an EVENT object is signaled. Waiting for an event causes no modification of the object and all waiting processes continue if their wait conditions are otherwise satisfied.

9.1.4 MESSAGE Object

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

The MESSAGE object describes a block of memory that can be moved from one job's virtual address space to another's. The block of memory is called a message's test variable and is allocated dynamically by the kernel from physically contiguous, page-aligned blocks of memory. A MESSAGE object and its associated text variable are both created by calling the CREATE_MESSAGE kernel service.

A MESSAGE object has the following associated properties:

- A data buffer to hold the message text
- The message length

Operations with MESSAGE Values

MESSAGE values are used in the following operations:

CREATE_MESSAGE	Creates a MESSAGE object that allocates and maps its associated message data.
DELETE_MESSAGE	Removes the MESSAGE 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.
RECEIVE	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).
SEND	Sends a message to a port. This 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.

9.1.5 MUTEX Object

A MUTEX object represents a mutually exclusive semaphore that monitors access to a resource. A mutex function is identical to a binary semaphore's function except that a mutex saves resources by not calling a kernel service unless contention occurs.

Operations with MUTEX Values

MUTEX values are used in the following operations:

CREATE_MUTEX	Initializes a mutex (initially unlocked) and creates its associated semaphore.
DELETE_MUTEX	Removes a MUTEX object from the system. When a mutex is deleted, any waiting processes are removed from their wait states immediately.

LOCK_MUTEX	Locks a mutex (used instead of WAIT_ANY or WAIT_ALL).
UNLOCK_MUTEX	Unlocks a mutex (used instead of SIGNAL).

9.1.6 NAME Object

A NAME object is an entry in a name table that associates character-string names with message ports. The local name table is used only within a node. The universal name table 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 two properties, either local or universal

Operations with NAME Values

NAME values are used in the following operations:

CREATE_NAME	Creates a NAME object and associate it with a message port.
DELETE_NAME	Deletes a specified message port name.
TRANSLATE_NAME	Finds the value of a port associated with the specified name.

9.1.7 PORT Object

A PORT object (or, informally, message port) is a destination for messages. Each port belongs to a particular job, but can be referenced from any job in the local area network. Unlike 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

Operations with PORT Values

PORT values are used in the following operations:

CONNECT_CIRCUIT	Connects a port to a specified destination port. This causes the invoking process to wait for the connection request to be accepted.
DISCONNECT_CIRCUIT	Breaks the circuit connection between two ports.
CREATE_PORT	Creates a message port.
JOB_PORT	Returns a PORT value identifying the caller's job port. A unique job port is created whenever a job is started.
RECEIVE	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.
WAIT_ALL or WAIT_ANY	Causes a process to wait 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.

9.1.8 PROCESS Object

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)

Operations with PROCESS Values

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.

PROCESS values are used in the following operations:

CREATE_PROCESS	Creates a new subprocess.
CURRENT_PROCESS	Returns the PROCESS value identifying the process from which it is called.
EXIT_PROCESS	Exits the calling process.
RAISE_PROCESS_EXCEPTION	Raises the asynchronous exception KER_PROCESS_ATTENTION in the specified process.
RESUME	Resumes the execution of a suspended process.
SET_PROCESS_PRIORITY	Sets the scheduling priority of the specified process to an integer in the range of 0 to 15. Priority 0 is the highest.
SUSPEND	Suspends the execution of a process.

9.1.9 SEMAPHORE Object

A SEMAPHORE object is used to protect a resource, including other data, from simultaneous access. It is also used to control the execution of processes that require some limited resource.

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. If the maximum allowed count equals one, use of a MUTEX object is recommended.

- A list of processes waiting for the semaphore to be signaled.

Operations with SEMAPHORE Values

SEMAPHORE values are used in the following operations:

CREATE_SEMAPHORE	Creates a semaphore and initializes it with an initial count and a maximum count.
DELETE_SEMAPHORE	Removes a SEMAPHORE object from the system. When a semaphore is deleted, any waiting processes are removed from their wait states immediately.
SIGNAL_SEMAPHORE	Increments and tests the semaphore count.
WAIT_ALL or WAIT_ANY	Causes a process to wait to be signaled by a semaphore. When the wait is satisfied, the semaphore count is decremented.

9.2 VAXELN Types

The identifying values for each object for VAXELN systems are defined as types in the package VAXELN_SERVICES. Each VAXELN object is represented by an Ada type as described in the following table.

Object Name	VAXELN Ada Type
AREA object	AREA_TYPE
DEVICE object	DEVICE_TYPE
EVENT object	EVENT_TYPE
MESSAGE object	MESSAGE_TYPE
Mutual-exclusion (MUTEX) object	MUTEX_TYPE
NAME object	NAME_TYPE
PORT object	PORT_TYPE
PROCESS object	PROCESS_TYPE
SEMAPHORE object	SEMAPHORE_TYPE

9.3 Using Strings with VAXELN Services

Many VAXELN services and utility procedures have one or more arguments that are of a string type. However, there are several kinds of strings that are used, and you must be aware of which kind is needed when passing an argument to a VAXELN procedure.

VAXELN procedure string arguments may be either fixed-length strings or varying-length strings. These are discussed in the next two sections.

9.3.1 Fixed-Length Strings

Fixed-length strings are the same as the Ada type STRING, and the usual Ada language features dealing with type STRING can be used. To identify a fixed-length string argument, refer to the argument type's definition in the first section of Appendix A. If the type is defined as STRING, then it is a fixed-length string, as in the following example:

```
subtype AREA_NAME_TYPE is STRING;
```

You can pass character and string literals, constants, or variables where the base argument type is STRING.

9.3.2 Varying-Length Strings

Most of the utility procedures require varying string arguments. The structure of a varying string is one that is compatible with the varying string types present in VAX Pascal and VAX PL/I. The structure consists of a 16-bit unsigned word integer that is the current length of the string, followed by an array of characters whose length is the maximum length of the string; this array contains the value of the varying string. The remainder of the value array past the "current length" is considered undefined. You can modify the contents of the string by changing the current length and the appropriate characters in the value array.

Ada does not define a varying string type, therefore type definitions for varying strings are provided in package VAXELN_SERVICES. The VAXELN procedures use varying strings in a variety of maximum lengths; the package contains a separate definition for each maximum length required. The name of each type is VS_n, where n is the maximum length.

Thus, to determine if a particular argument type is a varying string, examine the declaration of that type in Appendix A. For example:

```
        subtype FILE_NAME_TYPE is VS_255;
```

This declares FILE_NAME_TYPE to be a varying string of maximum length 255.

The following is an example of the definition of a varying string. This describes the type VS_16, a varying string of maximum length 16:

```
type VS_16 is
    record
        LENGTH: NATURAL range 0..16 := 0;
        VALUE : STRING (1..16);
    end record;

for VS_16 use
   record
       LENGTH at 0 range 0..15;
   end record;
```

The LENGTH and VALUE component in the previous example are common to all definitions of varying strings.

The LENGTH component

- is constrained to the maximum length of the varying string
- has a representation clause that causes the component to be the first 16 bits of the record
- has an initialization value to cause variables of this type to be initialized as zero-length strings

The VALUE component is the size of the maximum length of the string.

9.3.2.1 Uses of Varying Strings

The following examples show typical uses of varying strings in VAXELN Ada:

1. To declare an object of a varying string type, use the argument type name.

```
        LOG_FILE : FILE_NAME_TYPE
```

2. To assign a value to a varying string from a STRING variable and a separate current length:

   ```
   LOG_FILE.LENGTH := CURRENT_LENGTH;
   LOG_FILE.VALUE (1..CURRENT_LENGTH) := FIXED_STRING (1..CURRENT_LENGTH);
   ```

3. To use the value of a varying string as an item of type STRING:

   ```
   PUT_LINE (LOG_FILE.VALUE (1..LOG_FILE.LENGTH));
   ```

9.3.2.2 Varying String Descriptors

Some procedures have an argument of type VS_DESCR_TYPE. This is a descriptor of a varying string and is used when the maximum length of the string argument is not constant. An example of declaring and initializing such a descriptor is

```
MY_STRING : VS_255;
MY_STRING_DESCR : VS_DESCR_TYPE;
MY_STRING_DESCR.MAXLEN := MY_STRING.VALUE'LAST;
MY_STRING_DESCR.POINTER := MY_STRING'ADDRESS;
```

9.4 VAXELN Services

This section discusses VAXELN services by the type of function these services can be used to perform.

See Appendix A for a complete description of each system service and utility. These descriptions include the VAXELN Ada procedure specification for calling each service or utility.

9.4.1 Authorization Utility Procedures

The following procedures are used to set or return the user identity of processes:

Procedure	Purpose
GET_USER	Returns the user identity of a process.
SET_USER	Sets the user identity of the current process.

9.4.2 Authorization Service Utility Procedures

The following procedures are used to maintain the authorization database:

Procedure	Purpose
AUTH_ADD_USER	Adds a new user record to the authorization database.
AUTH_MODIFY_USER	Modifies an existing user record in the authorization database.
AUTH_REMOVE_USER	Removes an existing user record from the authorization database.
AUTH_SHOW_USER	Returns authorization database information.

9.4.3 Device Driver Utility Procedures

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.

9.4.3.1 Analog-to-Digital Converter Procedures

The following procedures are used in programs that access ADV11C or AXV11C real-time devices.

Procedure	Purpose
AXV_INITIALIZE	Readies an ADV or AXV device for input-output and creates all needed data structures.
AXV_READ	Reads analog data from the specified channels, converts it to binary form, and stores it in a data array.
AXV_WRITE	Writes a value to an analog-to-digital conversion output register on an AXV device.

9.4.3.2 Real-Time Clock Procedures

The following procedures are used in programs that access KWV11C real-time devices.

Procedure	Purpose
KWV_INITIALIZE	Readies a KWV device for input and creates all needed data structures.
KWV_READ	Reads time values from a KWV device and stores them in a data array.
KWV_WRITE	Sets up the KWV11C device to generate the clock-overflow signal.

9.4.3.3 Asynchronous Serial Line Controller Procedures

The following procedures are used in programs that access DLVJ1 serial line controller devices.

Procedure	Purpose
DLV_INITIALIZE	Readies a DLV device line for input-output and creates all needed data structures.
DLV_READ_BLOCK	Reads characters from a serial line until the specified number of characters are read.
DLV_READ_STRING	Reads characters from a serial line until a carriage return character is encountered.
DLV_WRITE_STRING	Writes the specified character string to a serial line.

9.4.3.4 Parallel Line Interface Procedures

The following procedures are used in programs that access DRV11-J parallel line interface devices.

Procedure	Purpose
DRV_INITIALIZE	Readies a DEV device controller for input-output and creates all needed data structures.
DRV_READ	Reads data words from the specified parallel port.
DRV_WRITE	Writes data words to the specified parallel port.

9.4.4 DMA Device Handling Procedures

The following direct memory mapping access (DMA) device handling procedures are used in memory mapping for UNIBUS and QBUS devices. These procedures can only be called from programs running in kernel mode.

Procedure	Purpose
ALLOCATE_MAP	Allocates a contiguous block of UNIBUS or QBUS map register.
ALLOCATE_PATH	Allocates a UNIBUS adapter buffered datapath.

Procedure	Purpose
FREE_MAP	Frees a set of previously allocated UNIBUS or QBUS map registers.
FREE_PATH	Frees a previously allocated UNIBUS adapter buffered datapath.
LOAD_UNIBUS_MAP	Loads UNIBUS or QBUS map registers.
PHYSICAL_ADDRESS	Returns the physical address of the variable supplied as its argument.
UNIBUS_MAP	Maps memory buffers for direct memory access by UNIBUS or QBUS devices.
UNIBUS_UNMAP	Unmaps memory buffers previously mapped for direct memory access by a UNIBUS or QBUS device.

9.4.5 Exception Handling Procedures

The following procedures relate to VAXELN exception handling and accessing the message database.

Procedure	Purpose
DISABLE_ASYNCH_EXCEPTION	Prevents the delivery of asynchronous exceptions.
ENABLE_ASYNCH_EXCEPTION	Allows the delivery of asynchronous exceptions.
GET_STATUS_TEXT	Returns the text associated with a status code.
RAISE_EXCEPTION	Causes a software exception in the calling process.
RAISE_PROCESS_EXCEPTION	Raises the KER$_PROCESS_ATTENTION exception.
UNWIND	Unwinds the call stack to a new location.

9.4.6 Exit Utility Procedures

The following procedures establish and delete an exit handler to perform cleanup operations following the termination of a job with the EXIT procedure.

Procedure	Purpose
CANCEL_EXIT_HANDLER	Deletes a specific exit handler routine.
DECLARE_EXIT_HANDLER	Calls an exit handler routine defined by the program.

9.4.7 File Service Procedures

9.4.7.1 Disk Utility Procedures

The following procedures are performed by the VAXELN disk File Service.

Procedure	Purpose
DISMOUNT_VOLUME	Dismounts a File Service volume on the specified disk drive.
INIT_VOLUME	Initializes a File Service disk for use as a file-structured volume.
MOUNT_VOLUME	Mounts a File Service disk for use as a file-structured volume.

9.4.7.2 File Utility Procedures

The following procedures are performed by the VAXELN File Service for disk and tape volumes.

Procedure	Purpose
COPY_FILE	Makes an exact duplicate of the specified file.
CREATE_DIRECTORY	Creates a directory on the specified disk volume.
DELETE_FILE	Deletes a file from a mounted disk volume.
DIRECTORY_CLOSE	Closes an existing directory on a mounted disk volume.
DIRECTORY_LIST	Obtains the next file name from a mounted disk directory.
DIRECTORY_OPEN	Opens an existing directory on a mounted disk volume in preparation for a directory listing.
PROTECT_FILE	Changes the protection of a disk file.
RENAME_FILE	Renames a disk file.

9.4.7.3 Tape Utility Procedures

The following procedures are performed by the VAXELN tape File Service.

Procedure	Purpose
DISMOUNT_TAPE_VOLUME	Dismounts a File Service tape on the specified tape drive.
INIT_TAPE_VOLUME	Initializes a File Service tape for use as a file-structured volume.
MOUNT_TAPE_VOLUME	Mounts a File Service tape for use as a file-structured volume.

9.4.8 Interrupt Priority Level Procedures

The following procedures raise or lower the processor's interrupt priority levels.

Procedure	Purpose
DISABLE_INTERRUPT	Prevents interrupts from a device by raising the interrupt priority level.
ENABLE_INTERRUPT	Allows interrupts from a device by lowering the interrupt priority level to 0.

9.4.9 Memory Allocation Procedures

The following procedures relate to allocating and freeing memory.

Procedure	Purpose
ALLOCATE_MEMORY	Allocates physical VAX memory pages into the virtual address space of the job that calls it.
FREE_MEMORY	Frees a region of physical VAX memory pages previously allocated
MEMORY_SIZE	Scans the kernel database and returns the value for the initial main memory, current free memory, and current largest, physically contiguous block of free memory.

9.4.10 Message Transmission Procedures

The following procedures relate to transmitting messages between processes, jobs, and ports.

Procedure	Purpose
ACCEPT_CIRCUIT	Establishes a circuit between two ports.
CONNECT_CIRCUIT	Connects a port to a specified destination port.
CREATE_MESSAGE	Creates a message and its associated message data.
CREATE_NAME	Creates a name for a port.

Procedure	Purpose
CREATE_PORT	Creates a message port.
DELETE_MESSAGE	Deletes the specified message object.
DELETE_PORT	Deletes the specified port.
DELETE_NAME	Deletes the specified name.
DISCONNECT_CIRCUIT	Breaks the circuit connection between two ports.
JOB_PORT	Returns the current job port.
RECEIVE	Receives a message from a port.
SEND	Sends a message from a port.
TRANSLATE_NAME	Returns a value identifying a named port.

9.4.11 Program Argument Procedures

The procedures summarized in this section obtain arguments, argument list lengths, and argument counts.

Procedure	Purpose
PROGRAM_ARGUMENT	Returns the character string passed as an argument.
PROGRAM_ARGUMENT_COUNT	Returns the number of arguments passed to the program.

9.4.12 Program Loader Procedures

The following procedures dynamically load and unload program images into a running VAXELN system after the initial system is built.

After a program image is loaded, the CREATE_JOB procedure is used to execute the program image.

Procedure	Purpose
LOAD_PROGRAM	Loads the specified image file into a running system.
UNLOAD_PROGRAM	Unloads the specified program from the system.

9.4.13 Time-Representation Procedure

The following procedure is used to set a new system time.

Procedure	Purpose
SET_TIME	Sets a new system time.

9.4.14 Virtual-to-Physical Address Procedure

See DMA Device Handling Procedures

9.4.15 Other Services

The following procedures are also available.

Procedure	Purpose
DISABLE_SWITCH	Disables process switching for the job from which it is called.
ENABLE_SWITCH	Restores preemptive process scheduling, or switching, for the calling job.
ENTER_KERNEL_CONTEXT	Executes the specified user routine in the kernel processor mode.
INITIALIZATION_DONE	Informs the kernel that the calling job has completed an initialization sequence, and other programs can be started if specified.
SET_JOB_PRIORITY	Sets the scheduling priority of the current job.

9.5 VAX/VMS Services Available with VAXELN Ada

The following VAX/VMS-compatible services are available with VAXELN Ada in the package STARLET:

SYS$ASCTIM
SYS$BINTIM
SYS$CANEXH
SYS$DCLEXH (Restriction—exit reason code not used)
SYS$EXIT
SYS$FAO (Restriction—!%U and !%I not supported)
SYS$FAOL (see SYS$FAO)
SYS$GETMSG
SYS$GETTIM
SYS$NUMTIM
SYS$PUTMSG (Third program argument used as file specification for output, if null, CONSOLE: is used)
SYS$UNWIND

The following VAX/VMS-compatible Run-Time Library routines are available:

LIB$SIGNAL
LIB$STOP (Restriction—does not disallow continuation)
LIB$CREATE_USER_VM_ZONE
LIB$CREATE_VM_ZONE
LIB$CVT_DTB
LIB$CVT_HTB
LIB$CVT_OTB
LIB$DELETE_VM_ZONE
LIB$FREE_VM
LIB$FREE_VM_PAGE
LIB$GET_VM
LIB$GET_VM_PAGE
LIB$MATCH_COND
LIB$RESET_VM_ZONE
LIB$SIG_TO_RET
LIB$TPARSE (Restriction—TPA$_FILESPEC, TPA$_UIC, TPA$_IDENT not supported)
OTS$CVT_L_TB
OTS$CVT_L_TI
OTS$CVT_L_TL
OTS$CVT_L_TO
OTS$CVT_L_TU

OTS$CVT_L_TZ
OTS$MOVE3
OTS$MOVE3_R5
OTS$MOVE5
OTS$MOVE5_R5

Appendix A

Package VAXELN_SERVICES

The first section of this appendix presents definitions for the types, subtypes, and constants used in the package VAXELN_SERVICES. The second section lists and describes the VAXELN Ada procedure specifications used in calling VAXELN services.

A.1 Type Definitions

The following information describes the types, subtypes, and constants used in VAXELN Ada service calls. It is the first major section of the package VAXELN_SERVICES. The complete specifications of this package can be obtained using the ACS EXTRACT SOURCE command. Explanatory text precedes each type definition.

```
with SYSTEM; use SYSTEM;
with CONDITION_HANDLING; use CONDITION_HANDLING;

package VAXELN_SERVICES is

    -- Package VAXELN_SERVICES provides the types, operations,
    -- constants, and so on that are needed to call VAXELN
    -- kernel services and utility routines.  Use of this
    -- package is described in Chapter 9 of the
    -- VAXELN Ada User's Manual.
    -- VAXELN_SERVICES is divided into the following major sections:
    --
    --      1.  Types, subtypes, and constants used in
    --          service calls.
    --
    --      2.  Kernel service routines.
    --
    --      3.  Utility routines.
    --
    --      4.  Status values that are returned by VAXELN services and
    --          utility routines.  These include all KER_ and ELN_
    --          status values.

-- Section 1.   Types, subtypes, and constants used in
--              service calls.

    -- Ada parameter types and subtypes for VAXELN
    -- service calls.
    --
    -- Types defined in predefined STANDARD that are used
    -- as parameter types for VAXELN service calls
    -- include BOOLEAN, INTEGER and STRING.
    -- Several types in predefined SYSTEM are used,
    -- including ADDRESS, UNSIGNED_LONGWORD,
    -- and other unsigned types.
    -- Type COND_VALUE_TYPE from package CONDITION_HANDLING
    -- is used.
    --
    -- Additional parameter types are defined as follows:

    -- Varying string types
    --
    -- Many VAXELN utility routines and some kernel services require
    -- string arguments to be passed as varying strings, following the
    -- VAX/VMS standard datatype VT (varying text) format.  This format
    -- includes an unsigned word for the current length of the string,
    -- followed immediately by the string data itself.  This format
    -- is also used as the VAXELN Pascal VARYING_STRING type.
    --
    -- While it is possible to define a general purpose varying string
    -- package in Ada, it is more efficient and more convenient to
    -- define a set of record types, one for each type of varying string
    -- used.  The current length and string value are then directly
    -- accessible and can be manipulated by the user as desired.

    -- For each varying string type, the record consists of a LENGTH and
    -- a VALUE.  LENGTH is an unsigned word integer that is constrained
    -- to be within the maximum length of the string type.  VALUE is
    -- the "body" of the string; a fixed length string of the proper
    -- length.  The constraints are necessary because the VAXELN services
    -- and utility routines assume that arguments are the correct type.
-- VS_6
--
-- A varying string with a maximum length of 6 characters.
--
type VS_6 is
    record
        LENGTH : NATURAL range 0..6 := 0;
        VALUE : STRING (1..6);
    end record;

for VS_6 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_12
--
-- A varying string with a maximum length of 12 characters.
--
type VS_12 is
    record
        LENGTH : NATURAL range 0..12 := 0;
        VALUE : STRING (1..12);
    end record;

for VS_12 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_16
--
-- A varying string with a maximum length of 16 characters.
--
type VS_16 is
    record
        LENGTH : NATURAL range 0..16 := 0;
        VALUE : STRING (1..16);
    end record;

for VS_16 use
    record
        LENGTH at 0 range 0..15;
    end record;
```

```
-- VS_20
--
-- A varying string with a maximum length of 20 characters.
--
type VS_20 is
    record
        LENGTH : NATURAL range 0..20 := 0;
        VALUE : STRING (1..20);
    end record;

for VS_20 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_30
--
-- A varying string with a maximum length of 30 characters.
--
type VS_30 is
    record
        LENGTH : NATURAL range 0..30 := 0;
        VALUE : STRING (1..30);
    end record;

for VS_30 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_32
--
-- A varying string with a maximum length of 32 characters.
--
type VS_32 is
    record
        LENGTH : NATURAL range 0..32 := 0;
        VALUE : STRING (1..32);
    end record;

for VS_32 use
    record
        LENGTH at 0 range 0..15;
    end record;
```

```
-- VS_40
--
-- A varying string with a maximum length of 40 characters.
--
type VS_40 is
    record
        LENGTH : NATURAL range 0..40 := 0;
        VALUE : STRING (1..40);
    end record;

for VS_40 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_64
--
-- A varying string with a maximum length of 64 characters.
--
type VS_64 is
    record
        LENGTH : NATURAL range 0..64 := 0;
        VALUE : STRING (1..64);
    end record;

for VS_64 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_100
--
-- A varying string with a maximum length of 100 characters.
--
type VS_100 is
    record
        LENGTH : NATURAL range 0..100 := 0;
        VALUE : STRING (1..100);
    end record;

for VS_100 use
    record
        LENGTH at 0 range 0..15;
    end record;
```

```
-- VS_128
--
-- A varying string with a maximum length of 128 characters.
--
type VS_128 is
    record
        LENGTH : NATURAL range 0..128 := 0;
        VALUE : STRING (1..128);
    end record;

for VS_128 use
    record
        LENGTH at 0 range 0..15;
    end record;

-- VS_255
--
-- A varying string with a maximum length of 255 characters.
--
type VS_255 is
    record
        LENGTH : NATURAL range 0..255 := 0;
        VALUE : STRING (1..255);
    end record;

for VS_255 use
    record
        LENGTH at 0 range 0..15;
    end record;


-- VAXELN object and argument types
--
-- This section contains the types of arguments that are passed to
-- VAXELN kernel services and utility procedures.
--

-- AREA_NAME_TYPE
--
-- A string of 1 to 31 characters specifying the name of an
-- AREA object.
--
subtype AREA_NAME_TYPE is STRING;
```

```
-- AREA_TYPE
--
-- 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.  Values of type AREA_TYPE identify
-- areas, and are returned by the CREATE_AREA procedure.  This
-- type should be considered private to the VAXELN kernel.
--
subtype AREA_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- AUTH_FIELDS_TYPE
--
-- A structure that is used to specify in
-- a call to the AUTH_MODIFY_USER procedure which fields
-- are to be modified.  If a component of
-- the record has the value TRUE, the
-- corresponding element of the authorization
-- record is changed.
--
type AUTH_FIELDS_TYPE is
    record
        USERNAME_FIELD  : BOOLEAN := FALSE;
        NODENAME_FIELD  : BOOLEAN := FALSE;
        PASSWORD_FIELD  : BOOLEAN := FALSE;
        UIC_FIELD       : BOOLEAN := FALSE;
        USERDATA_FIELD  : BOOLEAN := FALSE;
        FILLER_1        : UNSIGNED_27 := 0;
    end record;
for AUTH_FIELDS_TYPE use
    record
        USERNAME_FIELD      at 0 range 0 .. 0;
        NODENAME_FIELD      at 0 range 1 .. 1;
        PASSWORD_FIELD      at 0 range 2 .. 2;
        UIC_FIELD           at 0 range 3 .. 3;
        USERDATA_FIELD      at 0 range 4 .. 4;
        FILLER_1            at 0 range 5 .. 31;
    end record;
for AUTH_FIELDS_TYPE'SIZE use 32;

-- AUTH_STRING_TYPE
--
-- A varying string specifying a user name or password.
--
subtype AUTH_STRING_TYPE is VS_20;
```

```
-- ARG_LIST_TYPE
--
-- A procedure argument list consisting of one or more
-- longwords; the first longword contains an unsigned
-- integer count of the number of successive, contiguous
-- longwords, each of which is a parameter to be passed to
-- a procedure by means of a VAX CALLG instruction.
--
subtype ARG_LIST_TYPE is SYSTEM.UNSIGNED_LONGWORD_ARRAY;

-- AXV_DAC_CHANNEL_TYPE
--
-- An enumerated type that specifies which DAC channel
-- is to be used in a call to the AXV_WRITE procedure.
--
type AXV_DAC_CHANNEL_TYPE is (CHANNEL_A, CHANNEL_B);

-- AXV_DATA_TYPE
--
-- The values read from or written to an AXV device, or
-- read from an ADV device, are of this type.  The interpretation
-- of the values depends on the Data Notation (binary, offset
-- binary or two's complement) as specified by jumpers on the
-- board.
--
subtype AXV_DATA_TYPE is SYSTEM.UNSIGNED_WORD;

-- AXV_GAIN_ARRAY_TYPE
--
-- An array of AXV_GAIN_VALUES elements, one per channel,
-- used in a call to the AXV_READ procedure.
--
type AXV_GAIN_VALUES is (ONE, TWO, FOUR, EIGHT);
    for AXV_GAIN_VALUES'SIZE use 16;

type AXV_GAIN_ARRAY_TYPE is array (INTEGER range <>) of AXV_GAIN_VALUES;

-- AXV_IDENTIFIER_TYPE
--
-- The value used to identify a connection to an ADV or
-- AXV device.  This value is returned by a call to the
-- AXV_INITIALIZE procedure.
--
subtype AXV_IDENTIFIER_TYPE is SYSTEM.UNSIGNED_LONGWORD;
AXV_IDENTIFIER_ZERO : constant AXV_IDENTIFIER_TYPE := 0;

-- CIRCUIT_DATA_TYPE
--
-- A varying string that specifies or receives optional data in
-- a call to the ACCEPT_CIRCUIT procedure or the CREATE_CIRCUIT procedure.
--
subtype CIRCUIT_DATA_TYPE is VS_16;
```

```
-- COND_VALUE_TYPE (CONDITION_HANDLING.COND_VALUE_TYPE)
--
-- A VAXELN (and VAX/VMS) condition value.  (See
-- package CONDITION_HANDLING.)
--
COND_VALUE_ZERO: constant
    CONDITION_HANDLING.COND_VALUE_TYPE := 0;
COND_VALUE_1: constant
    CONDITION_HANDLING.COND_VALUE_TYPE := 1;

-- CREATE_DEVICE_NAME_TYPE
--
-- A string of 1 to 30 characters naming a device
-- in a call to the CREATE_DEVICE procedure.  The name must match one of th
-- device names established with the System Builder Utility.
--
subtype CREATE_DEVICE_NAME_TYPE is STRING;

-- DATE_TIME_TYPE
--
-- A 64-bit unsigned, binary integer denoting a date and
-- time as the number of elapsed 100-nanosecond units
-- since 00:00 o'clock, November 17, 1858
--
subtype DATE_TIME_TYPE is SYSTEM.UNSIGNED_QUADWORD;

-- DESTINATION_NAME_TYPE
--
-- A string specifying the destination for a circuit
-- connection request in a call to the CONNECT_CIRCUIT procedure.
-- The destination can be a name established by the
-- CREATE_NAME procedure or can be a DECnet
-- object number.  See the description of the
-- CONNECT_CIRCUIT procedure for more information.
--
subtype DESTINATION_NAME_TYPE is STRING;

-- DEVICE_NAME_TYPE
--
-- A varying string specifying the name of a device in
-- a call to a device driver utility initialization
-- routine.
--
subtype DEVICE_NAME_TYPE is VS_30;
```

```
-- DEVICE_NUMBER_TYPE
--
-- An integer value in the range of 0 to 15 specifying
-- a particular device in an array of devices.
--
subtype DEVICE_NUMBER_TYPE is NATURAL range 0..15;
DEVICE_NUMBER_ZERO: constant DEVICE_NUMBER_TYPE := 0;

-- DEVICE_TYPE and DEVICE_ARRAY_TYPE
--
-- 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.  Values of type DEVICE_TYPE identify
-- devices, and are returned by the CREATE_DEVICE procedure.
-- This type should be considered private to the VAXELN kernel.
-- The DEVICE_ARRAY_TYPE denotes an array of DEVICE objects, one for
-- each unit on a device controller, which is passed to
-- the CREATE_DEVICE procedure.
--
subtype DEVICE_TYPE is SYSTEM.UNSIGNED_LONGWORD;
type DEVICE_ARRAY_TYPE is array (INTEGER range <>) of DEVICE_TYPE;

-- DIR_CONTEXT_TYPE and DIR_CONTEXT_ACCESS_TYPE
--
-- The directory context as returned by the DIRECTORY_OPEN,
-- and specified on calls to DIRECTORY_CLOSE and DIRECTORY_LIST, to
-- identify a directory search context.
--
-- Note that prior to calling the DIRECTORY_OPEN procedure, you must
-- allocate a variable of type DIR_CONTEXT_TYPE.  For example:
--
--
-- DIRECTORY_CONTEXT : DIR_CONTEXT_ACCESS_TYPE := new DIR_CONTEXT_TYPE;
--
--
-- DIRECTORY_OPEN (DIR_CONTEXT := DIRECTORY_CONTEXT
--

type DIR_CONTEXT_TYPE is
    record
        DAPD          : SYSTEM.ADDRESS;
        DIR_STATUS    : SYSTEM.UNSIGNED_LONGWORD;
        SERVER        : VS_255;
        VOLUME        : VS_255;
        DIRECTORY     : VS_255;
    end record;

pragma PACK (DIR_CONTEXT_TYPE);

type DIR_CONTEXT_ACCESS_TYPE is access DIR_CONTEXT_TYPE;
```

```
-- DISK_BADBLOCK_TYPE
--
-- A record that describes one bad block on a disk volume,
-- as a range of either logical or physical block numbers.
-- DISK_BADBLOCK_TYPE is a variant record with the
-- discriminant being Boolean field PBN_FORMAT.  If the PBN_FORMAT
-- field is FALSE, the record describes a range of logical blocks.
-- If PBN_FORMAT is TRUE, the record describes a range of
-- physical blocks.  See the descriptions of types
-- DISK_BADLIST_TYPE and DISK_BADLIST_DESCR_TYPE, and of
-- the INIT_VOLUME procedure, for more information.
--
type DISK_BADBLOCK_TYPE (PBN_FORMAT : BOOLEAN := FALSE) is
    record
       FILL: SYSTEM.UNSIGNED_15 := 0;  -- Must be zero
       case PBN_FORMAT is
           when FALSE =>  -- logical block format
               START_LBN: INTEGER;
               LBN_COUNT: SYSTEM.UNSIGNED_WORD;
           when TRUE =>   -- physical block format
               SECTOR: SYSTEM.UNSIGNED_BYTE;
               TRACK: SYSTEM.UNSIGNED_BYTE;
               CYLINDER: SYSTEM.UNSIGNED_WORD;
               PBN_COUNT: SYSTEM.UNSIGNED_WORD;
       end case;
    end record;

for DISK_BADBLOCK_TYPE use
    record
        START_LBN at 0 range 0..31;
        LBN_COUNT at 4 range 0..15;
        SECTOR at 0 range 0..7;
        TRACK at 1 range 0..7;
        CYLINDER at 2 range 0..15;
        PBN_COUNT at 4 range 0..15;
        PBN_FORMAT at 6 range 0..0; -- discriminant
        FILL at 6 range 1..15;
    end record;

for DISK_BADBLOCK_TYPE'SIZE use 64;

-- DISK_BADLIST_DESCR_TYPE
--
-- A descriptor of an array that specifies a list of bad blocks
-- as input to the INIT_VOLUME procedure.  The LIST_SIZE field is to
-- be set to the size of the array of type DISK_BADLIST_TYPE that
-- is to be used; use the 'SIZE attribute to obtain this information.
-- The LIST_ADDRESS field is to be set to the address of the array;
-- use the 'ADDRESS attribute to obtain this information.
--
```

```
type DISK_BADLIST_DESCR_TYPE is
    record
       LIST_SIZE: INTEGER;
       LIST_ADDRESS: SYSTEM.ADDRESS;
    end record;

DISK_BADLIST_DESCR_NONE: constant DISK_BADLIST_DESCR_TYPE :=
    (LIST_SIZE => 0,
     LIST_ADDRESS => SYSTEM.ADDRESS_ZERO);

-- DISK_BADLIST_TYPE
--
-- An array of items of type DISK_BADBLOCK_TYPE that specifies
-- a list of bad blocks to the INIT_VOLUME procedure.
--
type DISK_BADLIST_TYPE is array (INTEGER range <>) of DISK_BADBLOCK_TYPE;


-- DISK_DATA_CHECK_TYPE
--
-- An enumerated value specifying the type of data
-- checking to be done for a disk.
--
type DISK_DATA_CHECK_TYPE is (CHECK_READ, CHECK_WRITE, NOCHECK);

-- DISK_INDEX_POSITION_TYPE
--
-- An enumerated value specifying the location of
-- a disk index.
--
type DISK_INDEX_POSITION_TYPE is (POSITION_BEGINNING,
    POSITION_MIDDLE, POSITION_END);

-- DISK_VOLUME_NAME_TYPE
--
-- A varying string specifying the name of a disk volume.
--
subtype DISK_VOLUME_NAME_TYPE is VS_12;

-- DLV_BLOCK_TYPE
--
-- Data returned from a call to the DLV_READ_BLOCK procedure is stored in an
-- array of characters of a fixed length, that length being the
-- value of the MAXIMUM_LENGTH argument to the DLV_INITIALIZE
-- procedure.  The call is to ensure that the
-- size of the array of type DLV_BLOCK_TYPE is sufficient to
-- hold the number of characters returned.
--
subtype DLV_BLOCK_TYPE is SYSTEM.UNSIGNED_BYTE_ARRAY;
```

```
-- DLV_IDENTIFIER_TYPE
--
-- This type identifies a connection to a specific DLV line.  Values of typ
-- DLV_IDENTIFIER_TYPE are returned by a call to the DLV_INITIALIZE
-- procedure.
--
subtype DLV_IDENTIFIER_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- DLV_STRING_TYPE
--
-- A string written to a DLV device by the DLV_WRITE_STRING procedure.
--
subtype DLV_STRING_TYPE is STRING;

-- DRV_IDENTIFIER_TYPE
--
-- This type identifies a connection to a specific DLV controller.  Values
-- type DRV_IDENTIFIER_TYPE are returned by a call to the
-- DRV_INITIALIZE procedure.
--
subtype DRV_IDENTIFIER_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- DRV_PORT_INDEX_TYPE
--
-- An enumerated value specifying which port of
-- a DRV device to use.
--
type DRV_PORT_INDEX_TYPE is (PORT_A, PORT_B,
    PORT_C, PORT_D);

-- DRV_PORT_SET_TYPE
--
-- A Boolean array specifying which DRV ports
-- are to be used for output.
--
type DRV_PORT_SET_TYPE is array (DRV_PORT_INDEX_TYPE) of BOOLEAN;
pragma PACK (DRV_PORT_SET_TYPE);

-- EVENT_TYPE
--
-- An EVENT object records occurrences of events in real time
-- and stores that information until explicitly cleared by
-- a program.  Values of type EVENT_TYPE identify events,
-- and are returned by the CREATE_EVENT procedure.  This type should
-- be considered private to the VAXELN kernel.
--
subtype EVENT_TYPE is SYSTEM.UNSIGNED_LONGWORD;
```

```
-- EVENT_STATE_TYPE
--
-- An enumerated type specifying the initial
-- value of an EVENT object.
--
type EVENT_STATE_TYPE is (CLEARED, SIGNALED);
for EVENT_STATE_TYPE use
    (CLEARED => 0, SIGNALED => 1);

-- EXIT_CONTEXT_TYPE
--
-- A value passed to an exit handler that is
-- also used to identify a particular exit handler.
--
subtype EXIT_CONTEXT_TYPE is SYSTEM.UNSIGNED_LONGWORD;
EXIT_CONTEXT_ZERO : constant EXIT_CONTEXT_TYPE := 0;

-- FILE_NAME_TYPE
--
-- A varying string that specifies or receives a
-- file specification.
--
subtype FILE_NAME_TYPE is VS_255;

-- FILE_PROTECTION_TYPE
--
-- A 16-bit file protection mask.  The mask contains four
-- 4-bit fields, each of which specifies the protection
-- to be applied to file access attempts by one of the
-- four categories of user.
--

type FILE_PROTECTION_FLAGS_TYPE is
    record
        NOREAD  : BOOLEAN := FALSE;   -- deny read access
        NOWRITE : BOOLEAN := FALSE;   -- deny write access
        NOEXE   : BOOLEAN := FALSE;   -- deny execution access
        NODEL   : BOOLEAN := FALSE;   -- deny delete access
    end record;

pragma PACK (FILE_PROTECTION_FLAGS_TYPE);

type FILE_PROTECTION_TYPE is
    record
        SYS : FILE_PROTECTION_FLAGS_TYPE;
        OWN : FILE_PROTECTION_FLAGS_TYPE;
        GRP : FILE_PROTECTION_FLAGS_TYPE;
        WLD : FILE_PROTECTION_FLAGS_TYPE;
    end record;
```

```
for FILE_PROTECTION_TYPE use
    record
        SYS at 0 range 0 .. 3;
        OWN at 0 range 4 .. 7;
        GRP at 1 range 0 .. 3;
        WLD at 1 range 4 .. 7;
    end record;

for FILE_PROTECTION_TYPE'SIZE use 16;

FILE_PROTECTION_FLAGS_NONE: constant FILE_PROTECTION_FLAGS_TYPE :=
    (NOREAD => FALSE,
     NOWRITE => FALSE,
     NOEXE => FALSE,
     NODEL => FALSE);
FILE_PROTECTION_TYPE_NONE: constant FILE_PROTECTION_TYPE :=
    (SYS => FILE_PROTECTION_FLAGS_NONE,
     OWN => FILE_PROTECTION_FLAGS_NONE,
     GRP => FILE_PROTECTION_FLAGS_NONE,
     WLD => FILE_PROTECTION_FLAGS_NONE);

-- IO_BUFFER_TYPE
--
-- An array of bytes used as an input-output
-- buffer by a device driver.
--
subtype IO_BUFFER_TYPE is SYSTEM.UNSIGNED_BYTE_ARRAY;

-- IPL_TYPE
--
-- A value denoting a specific Interrupt Priority
-- Level setting.  This type is used in device drivers.
--
subtype IPL_TYPE is SYSTEM.UNSIGNED_LONGWORD range 0..31;

-- JOB_ARGUMENT_TYPE
--
-- A string specifying an argument to a job created by a call
-- to the CREATE_JOB procedure.
--
subtype JOB_ARGUMENT_TYPE is STRING;

-- JOB_PRIORITY_TYPE
--
-- An integer value for the priority of a job
-- in the range of 0 to 31 with 0 being the
-- most urgent priority.
--
subtype JOB_PRIORITY_TYPE is NATURAL range 0..31;
JOB_PRIORITY_16 : constant JOB_PRIORITY_TYPE := 16;
```

```
-- KWV_CLOCK_RATE_TYPE
--
-- An enumerated value specifying the clock rate for
-- a KWV device.
--
type KWV_CLOCK_RATE_TYPE is (RATE_STOP, RATE_1MHZ,
    RATE_100KHZ, RATE_10KHZ, RATE_1KHZ, RATE_100HZ,
    RATE_ST1, RATE_LINE);

-- KWV_COUNTER_TYPE
--
-- A value range for data returned from a KWV device.
--
subtype KWV_COUNTER_TYPE is SYSTEM.UNSIGNED_WORD;
KWV_COUNTER_1 : constant KWV_COUNTER_TYPE := 1;

-- KWV_IDENTIFIER_TYPE
--
-- A value used to identify a connection to a KWV device.
--
subtype KWV_IDENTIFIER_TYPE is SYSTEM.UNSIGNED_LONGWORD;
KWV_IDENTIFIER_ZERO : constant KWV_IDENTIFIER_TYPE := 0;

-- KWV_MODE_TYPE
--
-- An enumerated value specifying the mode of operation
-- of a KWV device.
--
type KWV_MODE_TYPE is (MODE_ZERO, MODE_ONE, MODE_TWO, MODE_THREE);

-- MEMORY_PROTECTION_TYPE
--
-- An enumerated value specifying the protection to
-- be given to a set of memory pages.
--
type MEMORY_PROTECTION_TYPE is
    (READ_ONLY, READ_WRITE, NO_ACCESS);
for MEMORY_PROTECTION_TYPE use
    (READ_ONLY => 0, READ_WRITE => 1, NO_ACCESS => 2);

-- MESSAGE_TYPE
--
-- A MESSAGE object is used to send data from a job to a port,
-- which will usually be in another job.  Values of type
-- MESSAGE_TYPE identify messages, and are returned by the
-- CREATE_MESSAGE procedure.  This type should be considered
-- private to the VAXELN kernel.
--
subtype MESSAGE_TYPE is SYSTEM.UNSIGNED_LONGWORD;
```

```
-- MUTEX_TYPE
--
-- The MUTEX_TYPE type is provided as an optimization of
-- binary semaphores.  Locking and unlocking mutexes, when
-- there is no contention for the resource, do not involve
-- calls to the VAXELN kernel, resulting in a significant
-- improvement in performance compared to using a
-- SEMAPHORE object.  Mutexes are not VAXELN objects, and
-- thus cannot be used with the WAIT_ALL, WAIT_ANY and
-- SIGNAL procedures.  The operations provided for mutexes are
-- CREATE_MUTEX, LOCK_MUTEX,
-- UNLOCK_MUTEX and DELETE_MUTEX.
--
type MUTEX_TYPE is private;

-- NAME_TYPE
--
-- 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.  Values of type NAME_TYPE identify
-- names, and are returned by the CREATE_NAME procedure.  This
-- type should be considered private to the VAXELN kernel.
--
subtype NAME_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- NAME_SCOPE_TYPE
--
-- An enumerated type that specifies whether the name
-- being created (CREATE_NAME) or translated
-- (TRANSLATE_NAME) is of local or universal
-- scope.  Local names are valid only on the
-- executing node, while universal names are
-- valid throughout the application or on
-- any node.  The value BOTH is only applicable
-- to the TRANSLATE_NAME procedure.
--
type NAME_SCOPE_TYPE is (LOCAL, UNIVERSAL, BOTH);
for NAME_SCOPE_TYPE use
    (LOCAL => 0, UNIVERSAL => 1, BOTH => 2);

-- NODE_NAME_TYPE
--
-- A varying string specifying or receiving a node name.
--
subtype NODE_NAME_TYPE is VS_32;
```

```
-- PHYSICAL_ADDRESS_TYPE
--
-- The physical address of a memory location.
--
subtype PHYSICAL_ADDRESS_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- PORT_NAME
--
-- A string specifying or receiving the name of a PORT object.
--
subtype PORT_NAME_TYPE is STRING;

-- PORT_TYPE
--
-- 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.  Values of
-- type PORT_TYPE identify ports, and are returned by the
-- CREATE_PORT procedure.  This type should be considered
-- private to the VAXELN kernel.
--
type PORT_TYPE is array (1..4) of SYSTEM.UNSIGNED_LONGWORD;

-- PROCESS_TYPE
--
-- 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.  Values of type PROCESS_TYPE identify processes, and
-- are returned by the CREATE_PROCESS procedure.  This type should
-- be considered private to the VAXELN kernel.
--
subtype PROCESS_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- PROCESS_PRIORITY_TYPE
--
-- An integer value for the priority of a process
-- in the range of 0 to 15 with 0 being the
-- most urgent priority.  Note that the Ada task
-- priority is (15 - p) where p is the VAXELN
-- process priority.
--
subtype PROCESS_PRIORITY_TYPE is NATURAL range 0..15;
PROCESS_PRIORITY_8 : constant PROCESS_PRIORITY_TYPE := 8;
```

```
-- PROGRAM_ARGUMENT_TYPE
--
-- A string containing an individual job argument string.
--
subtype PROGRAM_ARGUMENT_TYPE is VS_255;
-- PROGRAM_NAME_TYPE
--
-- A string specifying the name of a program to be loaded
-- or unloaded.
--
subtype PROGRAM_NAME_TYPE is VS_40;

-- A string specifying the name of a program to be created.
--
subtype PROGRAM_NAME_STRING_TYPE is STRING;

-- SEMAPHORE_TYPE
--
-- 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 values of type SEMAPHORE_TYPE identify semaphores, and are
-- returned by the CREATE_SEMAPHORE procedure.  This type should
-- be considered private to the VAXELN kernel.  See also MUTEX_TYPE.
--
subtype SEMAPHORE_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- SERVER_NAME_TYPE
--
-- A varying string specifying the name of a DECnet node.
--
subtype SERVER_NAME_TYPE is VS_64;

-- SYSTEM_VALUE_TYPE
--
-- Any of the following VAXELN object types:
-- AREA, DEVICE, EVENT, PROCESS or SEMAPHORE.
-- PORT is not included.  This type is used in
-- WAIT_ALL and WAIT_ANY procedures that can accept any of
-- the listed types.  The WAIT_ALL and WAIT_ANY procedures
-- can also accept PORT_TYPE values; see the description
-- of these services in the VAXELN Ada User's Manual for
-- details.
--
subtype SYSTEM_VALUE_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- TAPE_VOLUME_NAME_TYPE
--
-- A varying string specifying the name of a tape volume.
--
subtype TAPE_VOLUME_NAME_TYPE is VS_6;
```

```
-- UIC_LONGWORD_TYPE
--
-- An unsigned longword denoting a User Identification
-- Code (UIC)
--
subtype UIC_LONGWORD_TYPE is SYSTEM.UNSIGNED_LONGWORD;
UIC_LONGWORD_ZERO : constant UIC_LONGWORD_TYPE := 0;
UIC_LONGWORD_1_1 : constant UIC_LONGWORD_TYPE := 16  00010001  ;

-- UNIBUS_ADDRESS_TYPE
--
-- The 18-bit UNIBUS or 22-bit QBUS address of a mapped
-- memory buffer.
--
subtype UNIBUS_ADDRESS_TYPE is SYSTEM.UNSIGNED_LONGWORD;

-- USER_DATA_TYPE
--
-- A varying string that specifies or receives data in calls
-- to Authorization Service utility routines.
--
subtype USER_DATA_TYPE is VS_128;

-- VECTOR_NUMBER_TYPE
--
-- An integer value from 1 to 128 specifying which vector
-- of a multiple-interrupt-vector device should be
-- connected to the interrupt-service routine in a call
-- to the CREATE_DEVICE procedure.
--
subtype VECTOR_NUMBER_TYPE is NATURAL range 1..128;
VECTOR_NUMBER_1: constant VECTOR_NUMBER_TYPE := 1;
-- VS_DESCR_TYPE
--
-- A descriptor of a varying string; needed when calling
-- certain services that accept varying strings of
-- arbitrary maximum length.  This descriptor is compatible
-- with the VAX/VMS class VS string descriptor.  To create
-- a descriptor of a varying string, set the MAXLEN field
-- to the maximum length of the string (use the 'SIZE
-- attribute of the string's VALUE field), and set the
-- POINTER field to the address of the varying string
-- record.
--
type VS_DESCR_TYPE is
    record
       MAXLEN: SYSTEM.UNSIGNED_WORD := 0;
       DTYPE: SYSTEM.UNSIGNED_BYTE := 37;  -- DSC$K_DTYPE_VT
       CLASS: SYSTEM.UNSIGNED_BYTE := 11;  -- DSC$K_CLASS_VS
       POINTER: SYSTEM.ADDRESS := SYSTEM.ADDRESS_ZERO;
    end record;
```

A.2 VAXELN Service Procedure Descriptions

This section contains the procedure specifications for calling VAXELN services and provides detailed argument descriptions for each procedure. A list of possible status values that may be returned by the procedure is also provided where applicable. This list is not intended to be exhaustive, as most procedures invoke other procedures, which may fail and return error status messages. The messages listed are those that a program would return in a normal operation and for which some recovery may be possible. Your program should always check the status returned by a procedure for a possible failure.

Note that, as a quick reference, the word "optional" appears beside the arguments that are not required for each procedure specification.

ACCEPT_CIRCUIT

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.

Procedure Declaration

```
procedure ACCEPT_CIRCUIT (
   STATUS       : out COND_VALUE_TYPE;       --optional
   SOURCE_PORT  : in  PORT_TYPE;
   CONNECT_PORT : in  PORT_TYPE;             --optional
   FULL_ERROR   : in  BOOLEAN;               --optional
   ACCEPT_DATA  : in  CIRCUIT_DATA_TYPE;     --optional
   CONNECT_DATA : out CIRCUIT_DATA_TYPE);    --optional
```

Arguments

STATUS

This argument receives the completion status of the ACCEPT_CIRCUIT procedure.

SOURCE_PORT

This argument supplies the value of the port on which to wait for a connection request. Unless the CONNECT_PORT argument is present, this port also forms the invoker's half of the circuit. If, during the call, this port receives a message that is not a connection request, the message is ignored.

CONNECT_PORT

This argument supplies a different port, which is used for the actual connection; if the CONNECT_PORT argument is absent, the SOURCE_PORT value is used for the connection.

FULL_ERROR

This argument supplies a value to enable or disable the implicit wait caused when the partner port is full. The default is FALSE, meaning that the sender waits if the partner is full. If TRUE is supplied, an error status or the corresponding exception occurs with the SEND procedure when you attempt to send a message and the partner's port is full.

ACCEPT_DATA
This argument supplies a varying string value that is passed to the process requesting the circuit connection (that is, the requesting process receives this value in the ACCEPT_DATA parameter of its CONNECT_CIRCUIT call).

CONNECT_DATA
This argument receives varying string data passed by the requesting process in the CONNECT_DATA parameter of its CONNECT_CIRCUIT call.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_STATE	A port specified to the ACCEPT_CIRCUIT procedure contains unreceived messages or has an incomplete CONNECT_CIRCUIT or ACCEPT_CIRCUIT procedure pending.
KER_CONNECT_PENDING	A CONNECT_CIRCUIT procedure is pending, and the port cannot be used for another purpose until the connection has completed.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_SUCH_PORT	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 ACCEPT_CIRCUIT procedure.

ALLOCATE_MAP

The ALLOCATE_MAP procedure assigns a contiguous block of UNIBUS map registers for use by a device driver program to map VAX memory to UNIBUS memory addresses. This procedure can be called only from programs running in kernel mode.

Procedure Declaration

```
procedure ALLOCATE_MAP (
   STATUS      : out COND_VALUE_TYPE;    --optional
   REGISTER    : out ADDRESS;
   NUMBER      : out INTEGER;
   COUNT       : in  INTEGER;
   DEVICE      : in  DEVICE_TYPE;
   SPT_ADDRESS : out ADDRESS);           --optional
```

Arguments

STATUS
This argument receives the completion status of the ALLOCATE_MAP procedure.

REGISTER
This argument receives the address of the first register allocated.

NUMBER
This argument receives the starting map register number (0 to 495).

COUNT
This argument supplies the number of registers to allocate.

DEVICE
This argument supplies the DEVICE value that identifies the device for which the registers are to be used.

SPT_ADDRESS
This argument receives the base address of the system page table (SPT).

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MODE	The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.
KER_BAD_TYPE	The DEVICE argument is not of type DEVICE_TYPE.
KER_BAD_VALUE	The DEVICE argument is invalid or refers to a deleted device.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_MAP_REGISTER	No free UNIBUS map registers are currently available; there are 496 map registers per UNIBUS.

ALLOCATE_MEMORY

The ALLOCATE_MEMORY procedure assigns physical VAX memory to the job that calls the procedure. The memory allocation can be specified to start at a given virtual address or at a given physical address.

Procedure Declaration

```
procedure ALLOCATE_MEMORY (
   STATUS     : out COND_VALUE_TYPE;     --optional
   MEMPOINTER : out ADDRESS;
   SIZE       : in  INTEGER;
   VIRTUAL    : in  ADDRESS;             --optional
   PHYSICAL   : in  ADDRESS);            --optional
```

Arguments

STATUS
This argument receives the completion status of the ALLOCATE_MEMORY procedure.

MEMPOINTER
This argument receives a pointer to the first location of the allocated memory. The received value is always a virtual address.

SIZE
This argument supplies the number of bytes of memory to allocate. The value you supply is rounded up to the next multiple of 512.

VIRTUAL
This argument supplies the starting virtual address of the allocated memory. If necessary, the value is truncated to address a 512-byte page boundary. If this argument is omitted, the memory is allocated using any available contiguous address space in region P0. If the argument is present, allocation is attempted at the specified location in P0 or P1.

PHYSICAL

This argument supplies the starting physical address of the allocated memory. If necessary, the value is truncated to address a 512-byte page boundary. If it is omitted, the allocated memory comes from the system's pool of free memory.

Notes:

Allocating physical memory that is part of the processor's main memory may have unpredictable results. This procedure is provided primarily so a process can map some other addressable object (for example, a bit-map graphics screen) into its virtual address space.

Specific physical memory can be allocated only from programs running in kernel mode. The kernel does not restrict or control the specific physical address when this parameter is used. Therefore, a program can accidentally "double-map" pages of memory that are already in use. This feature is intended primarily for very specialized applications; for example, multiported memories or video memories.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MODE	The PHYSICAL argument was specified by a program that was not running in kernel mode; kernel mode is required to allocate specific physical memory.
KER_BAD_VALUE	The VIRTUAL argument is not in the job's address space.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_MEMORY	No free pages of physical memory are currently available.
KER_NO_VIRTUAL	No free contiguous virtual address space is currently available for the process; the size of process virtual address space can be set using the System Builder Utility.

ALLOCATE_PATH

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

Procedure Declaration

```
procedure ALLOCATE_PATH (
   STATUS   : out COND_VALUE_TYPE;     --optional
   REGISTER : out ADDRESS;
   NUMBER   : out INTEGER;
   DEVICE   : in  DEVICE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the ALLOCATE_PATH procedure.

REGISTER
This argument receives the address of the allocated datapath register.

NUMBER
This argument receives the allocated datapath register number (1 to 3).

DEVICE
This argument supplies the DEVICE value that identifies the device for which the datapath is allocated.

Notes:

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 the VAXELN system 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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MODE	The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.
KER_BAD_TYPE	The DEVICE argument is not of type DEVICE_TYPE.
KER_BAD_VALUE	The DEVICE argument is invalid or refers to a deleted device.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_PATH_REGISTER	No free UNIBUS adapter datapath register is currently available; there are three buffered datapaths per VAX-11/750 UNIBUS adapter.

AUTH_ADD_USER

The AUTH_ADD_USER procedure adds a new user record to the authorization database. This procedure requires that the caller be authorized with a system group UIC (that is, a UIC less than or equal to 16#0008FFFF# or [10,177777]).

Procedure Declaration

```
procedure AUTH_ADD_USER (
   STATUS  : out COND_VALUE_TYPE;          --optional
   AUTH_CIRCUIT : in  PORT_TYPE;
   USER_NAME    : in  AUTH_STRING_TYPE;
   NODE_NAME    : in  NODE_NAME_TYPE;
   PASSWORD     : in  AUTH_STRING_TYPE;
   UIC          : in  UIC_LONGWORD_TYPE;
   USER_DATA    : in  USER_DATA_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the AUTH_ADD_USER procedure.

AUTH_CIRCUIT
This argument specifies the port that is connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

USER_NAME
This argument supplies a varying string containing the user name for the new user; it cannot be blank. The reserved name $ANY can be specified for the USER_NAME argument, meaning that any user from the specified node who does not match one of the explicit user names is authorized with the specified UIC. See Table 4-8 for more information on the Authorization Service.

NODE_NAME
This argument supplies a varying string containing the node name for the node on which the new user is authorized; it can be blank. If NODE_NAME is specified, the database record represents a proxy authorization and the password is unused. If NODE_NAME is not specified, the record represents a destination authorization. The reserved name $ANY can be specified for the NODE_NAME argument, meaning 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.

PASSWORD
This argument supplies a varying string containing the password for the new user. If a destination authorization record is added, the password is stored with the record. Passwords are always stored in a scrambled form so they cannot be read once stored. The password argument can be blank.

UIC
This argument supplies the UIC assigned to the new user.

USER_DATA
This argument is an arbitrary string of user-specified data. It is stored with the user record for use by applications.

Notes:

If $ANY is specified for NODE_NAME and USER_NAME, users who do not match an explicit USER_NAME/NODE_NAME combination are authorized with the specified UICs—in other words, the default.

This procedure requires that the caller be authorized with a system group UIC (that is, a UIC of less than or equal to 16#0008FFFF# or [10,177777]).

AUTH_ADD_USER

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_AUTH_NO_PRIVILEGE	The caller does not have privilege to perform the requested action.
ELN_AUTH_DUPLICATE_USER	The AUTH_ADD_USER procedure was called with a name that is a duplicate of an existing name.
ELN_AUTH_INVALID_UIC	The caller does not have the authorized UIC.
KER_BAD_TYPE	The AUTH_CIRCUIT argument does not specify a port.
KER_NO_SUCH_PORT	The AUTH_CIRCUIT argument specifies a port that cannot be found.

AUTH_MODIFY_USER

The AUTH_MODIFY_USER procedure modifies an existing user record in the authorization database. This procedure requires that the caller be authorized with a system group UIC (that is, a UIC less than or equal to 16#0008FFFF# or [10,177777]).

Procedure Declaration

```
procedure AUTH_MODIFY_USER (
   STATUS        : out COND_VALUE_TYPE;      --optional
   AUTH_CIRCUIT  : in  PORT_TYPE;
   USER_NAME     : in  AUTH_STRING_TYPE;
   NODE_NAME     : in  NODE_NAME_TYPE;
   NEW_FIELDS    : in  AUTH_FIELDS_TYPE;
   NEW_USER_NAME : in  AUTH_STRING_TYPE;
   NEW_NODE_NAME : in  NODE_NAME_TYPE;
   NEW_PASSWORD  : in  AUTH_STRING_TYPE;
   NEW_UIC       : in  UIC_LONGWORD_TYPE;
   NEW_USER_DATA : in  USER_DATA_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the AUTH_MODIFY_USER procedure.

AUTH_CIRCUIT
This argument specifies the port that is connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

USER_NAME
This argument supplies a varying string containing the user name for the record to be modified; it cannot be blank.

NODE_NAME
This argument supplies a varying string containing the node name, which must be the user's authorized node.

NEW_FIELDS
This argument, which can be blank, supplies a structure which is used to specify which fields are to be modified.

NEW_USER_NAME
This argument supplies a varying string containing a new user name for the user; it cannot be blank. The reserved name $ANY can be specified for the NEW_USER_NAME argument, meaning that any user from the specified node who does not match one of the explicit user names is authorized with the specified UIC. Note that if the user name is modified, the password must be reset as well.

NEW_NODE_NAME
This argument supplies a varying string containing a new node name for the user's authorized node; it can be blank. If NEW_NODE_NAME is specified, the database record represents a proxy authorization and the password is unused. If NEW_NODE_NAME is not specified, the record represents a destination authorization. The reserved name $ANY can be specified for the NEW_NODE_NAME argument, meaning 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.

NEW_PASSWORD
This argument supplies a varying string containing a new password for the user; it can be blank. If a destination authorization record is added, the password is stored with the record. Passwords are always stored in a scrambled form so they cannot be read once stored. Note that if the user name is modified, the password must be reset as well.

NEW_UIC
This argument supplies the new UIC assigned to the user.

NEW_USER_DATA
This argument is an arbitrary string of user-specified data. It is stored with the user record for use by applications.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_AUTH_NO_PRIVILEGE	The caller does not have privilege to perform the requested action.
ELN_AUTH_NO_SUCH_USER	The user name does not match any existing user name.
ELN_AUTH_INVALID_UIC	The caller does not have the authorized UIC.
KER_BAD_TYPE	The AUTH_CIRCUIT argument does not specify a port.
KER_NO_SUCH_PORT	The AUTH_CIRCUIT argument specifies a port that cannot be found.

AUTH_REMOVE_USER

The AUTH_REMOVE_USER procedure removes an existing user record from the authorization database. This procedure requires that the caller be authorized with a system group UIC (that is, a UIC less than or equal to 16#0008FFFF# or [10,177777]).

Procedure Declaration

```
procedure AUTH_REMOVE_USER (
   STATUS       : out COND_VALUE_TYPE;     --optional
   AUTH_CIRCUIT : in  PORT_TYPE;
   USER_NAME    : in  AUTH_STRING_TYPE;
   NODE_NAME    : in  NODE_NAME_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the AUTH_REMOVE_USER procedure.

AUTH_CIRCUIT
This argument specifies the port that is connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

USER_NAME
This argument supplies a varying string containing the user name of the user to be removed; it cannot be blank.

NODE_NAME
This argument supplies a varying string containing the node name, which must be the node that the user is no longer authorized to use.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_AUTH_NO_PRIVILEGE	The caller does not have privilege to perform the requested action.
ELN_AUTH_NO_SUCH_USER	The user name does not match any existing user name.
KER_BAD_TYPE	The AUTH_CIRCUIT argument does not specify a port.
KER_NO_SUCH_PORT	The AUTH_CIRCUIT argument specifies a port that cannot be found.

AUTH_SHOW_USER

The AUTH_SHOW_USER procedure returns authorization database information for the specified users.

Procedure Declaration

```
procedure AUTH_SHOW_USER (
   STATUS               : out COND_VALUE_TYPE;     --optional
   AUTH_CIRCUIT         : in  PORT_TYPE;
   USER_NAME            : in  AUTH_STRING_TYPE;
   NODE_NAME            : in  NODE_NAME_TYPE;
   SHOW_ROUTINE_ADDRESS : in  ADDRESS);
```

Arguments

STATUS
This argument receives the completion status of the AUTH_SHOW_USER procedure.

AUTH_CIRCUIT
This argument specifies the port that is connected in a circuit to the Authorization Service's AUTH$MAINTENANCE port.

USER_NAME
This argument supplies a varying string containing the user name of the user records to be accessed; it cannot be blank.

NODE_NAME
This argument supplies a varying string containing the node name which must be the user's authorized node.

SHOW_ROUTINE_ADDRESS
This argument supplies the address of a user routine to be invoked by the AUTH_SHOW_USER procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_AUTH_NO_PRIVILEGE	The caller does not have privilege to perform the requested action.
ELN_AUTH_NO_SUCH_USER	The user name does not match any existing user name.
KER_BAD_TYPE	The AUTH_CIRCUIT argument does not specify a port.
KER_NO_SUCH_PORT	The AUTH_CIRCUIT argument specifies a port that cannot be found.

AXV_INITIALIZE

The AXV_INITIALIZE procedure readies an ADV or AXV device for input-output and creates all needed data structures. This procedure must be called at least once for each AXV or ADV device used. (The only reason to call this procedure more than once for a single device is to change the values of the Boolean flag parameters or the MAXIMUM_VALUES parameter.)

Procedure Declaration

```
procedure AXV_INITIALIZE (
   DEVICE_NAME           : in  DEVICE_NAME_TYPE;
   IDENTIFIER            : out AXV_IDENTIFIER_TYPE;
   MAXIMUM_VALUES        : in  INTEGER;
   CLOCK_START_ENABLE    : in  BOOLEAN;              --optional
   EXTERNAL_START_ENABLE : in  BOOLEAN;              --optional
   RE_INITIALIZE         : in  BOOLEAN;              --optional
   USE_POLLING           : in  BOOLEAN;              --optional
   STATUS                : out COND_VALUE_TYPE);     --optional
```

Arguments

DEVICE_NAME
This argument supplies a varying string giving the name of the device to be initialized. This name must match the name established with the System Builder Utility.

IDENTIFIER
This argument receives a value to be used to identify this device in subsequent calls to the procedures AXV_INITIALIZE, AXV_READ, and AXV_WRITE.

MAXIMUM_VALUES
This argument supplies the maximum number of data values that can be read from this device in a single call to the AXV_READ procedure.

CLOCK_START_ENABLE
This argument is a Boolean expression. TRUE enables a conversion to be initiated by the KWV clock option. The default value is FALSE.

EXTERNAL_START_ENABLE
This argument is a Boolean expression. TRUE enables a conversion to be initiated by an external signal. The default value is FALSE.

RE_INITIALIZE
This argument is a Boolean expression. TRUE means that the device has been initialized previously, in which case the DEVICE_NAME argument is ignored and the IDENTIFIER argument is used to identify the device. No new data structures or objects are created unless the MAXIMUM_VALUES argument is greater than the previous value for this device. The default value is FALSE.

USE_POLLING
This argument is a Boolean expression. TRUE causes the device to be driven by polling rather than interrupts; FALSE means interrupts will be used to gather data. Polling is always done at device IPL (four for this device). The default value is FALSE. Polling is only recommended if clock or external starting is being used to initiate conversions.

STATUS
This argument receives the completion status of the AXV_INITIALIZE procedure. The only possible value is 1, which indicates that the procedure completed successfully.

Status Value

KER_SUCCESS	The procedure completed successfully.

AXV_READ

The AXV_READ procedure causes analog data to be sampled from specified channels. This data is then converted to binary form by the device, and stored in a data array. The procedure performs one read for each desired channel, and continues until all data has been collected.

Procedure Declaration

```
procedure AXV_READ (
   IDENTIFIER        : in  AXV_IDENTIFIER_TYPE;
   START_CHANNEL     : in  INTEGER;
   END_CHANNEL       : in  INTEGER;
   READS_PER_CHANNEL : in  INTEGER;
   DATA_ARRAY_PTR    : out ADDRESS;
   KWV_IDENTIFIER    : in  KWV_IDENTIFIER_TYPE;     --optional
   GAIN_ARRAY        : in  AXV_GAIN_ARRAY_TYPE;     --optional
   STATUS            : out COND_VALUE_TYPE);        --optional
```

Arguments

IDENTIFIER
This argument supplies a value that identifies the device. This value is the one returned in the identifier parameter after a call to the AXV_INITIALIZE procedure.

START_CHANNEL
This argument supplies the first analog channel number to be read.

END_CHANNEL
This argument supplies the last analog channel number to be read.

READS_PER_CHANNEL
This argument supplies the number of items to be gathered from each channel.

DATA_ARRAY_PTR

This argument receives the address of an array containing converted data from the device. (The meaning of the converted data depends on the positions of several hardware jumpers. The first array element corresponds to the first channel read. All or part of the array may be overwritten by subsequent calls to the AXV_READ procedure for this device.

KWV_IDENTIFIER

This argument supplies the identifier of a KWV real-time clock device; this value is the one returned in the identifier parameter after a call to the KWV_INITIALIZE argument. If the KWV_IDENTIFIER argument is present, it is assumed that the KWV device's clock overflow is connected to the AXV/ADV's clock start line. Just before the data is sampled, the clock is started, and it is stopped when all data has been gathered. The KWV device must have been initialized to operate in mode 1 (if more than one value is to be read) or mode 0 (if only one value is to be read) and set up with the desired tick count (which controls how often an overflow is generated) by a call to the KWV_WRITE procedure. The call to the KWV_WRITE procedure must also have specified the ST2_GO_ENABLE argument as TRUE so that the call to the AXV_READ procedure will do the actual starting of the clock. If the KWV_IDENTIFIER procedure is not present, the call to the AXV_READ procedure does nothing to start a real-time clock.

GAIN_ARRAY

This argument supplies the gain to be used in the data conversion for each channel being read. The first array element corresponds to the first channel to be read. The allowable values for this argument are one, two, four, and eight, which are specified by the enumerated type AXV_GAIN_VALUES. If this argument is not present, the gain value that was used for the last conversion from this AXV device will be used. If no gains were ever used on this device, its initial hardware value of 1 will be used.

STATUS

This argument receives the completion status of the AXV_READ procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_DEVICE_ERROR	This value indicates that either a sampling rate is too high and the data is subject to error, or that a conversion was finished before the previous conversion data was read. Both of these conditions can occur only if conversions are being initiated by the clock or an external signal.

AXV_WRITE

The AXV_WRITE procedure causes a value to be written to an analog-to-digital conversion output register on an AXV11C device. These registers are not present on an ADV11C device; therefore, this procedure cannot be called from an ADV11C device. Calling this procedure causes an analog output voltage to be generated on the specified channel.

Procedure Declaration

```
procedure AXV_WRITE (
   IDENTIFIER  : in  AXV_IDENTIFIER_TYPE;
   DAC_CHANNEL : in  AXV_DAC_CHANNEL_TYPE;
   DATA        : in  AXV_DATA_TYPE;
   STATUS      : out COND_VALUE_TYPE);        --optional
```

Arguments

IDENTIFIER
This argument identifies the device to be written to; this value is the one returned in the identifier parameter after a call to the AXV_INITIALIZE procedure.

DAC_CHANNEL
This argument supplies the output channel to be written to.

DATA
This argument supplies the actual data to be written. This value determines the output voltage by means of hardware jumper settings, as described in the *Microcomputer Products Handbook.*

STATUS
This argument receives the completion status of the AXV_WRITE procedure. The only possible value is 1, which indicates that the procedure completed successfully.

Status Value

KER_SUCCESS	The procedure completed successfully.

CANCEL_EXIT_HANDLER

The CANCEL_EXIT_HANDLER procedure allows you to cancel an exit handler (identified by the exit handler and an associated context value), that was enabled by the DECLARE_EXIT_HANDLER procedure.

Procedure Declaration

```
procedure CANCEL_EXIT_HANDLER (
   EXIT_HANDLER : in  ADDRESS;
   EXIT_CONTEXT : in  EXIT_CONTEXT_TYPE);     --optional
```

Arguments

EXIT_HANDLER
This argument supplies a procedure address that identifies the exit handler routine to be canceled.

EXIT_CONTEXT
This argument must exactly match the EXIT_CONTEXT_TYPE variable used in the DECLARE_EXIT_HANDLER call in order for the proper handler to be canceled.

Status Value

KER_SUCCESS	The procedure completed successfully.

CLEAR_EVENT

The CLEAR_EVENT procedure sets the state of an EVENT value to cleared.

Procedure Declaration

```
procedure CLEAR_EVENT (
   STATUS : out COND_VALUE_TYPE;     --optional
   EVENT  : in  EVENT_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the CLEAR_EVENT procedure.

EVENT
This argument supplies the EVENT value to be cleared.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The EVENT argument is not of type EVENT.
KER_BAD_VALUE	The EVENT argument is invalid or refers to a deleted event.

CONNECT_CIRCUIT

The CONNECT_CIRCUIT procedure connects a port to a specified destination port. If the process receiving the connection request accepts the request, the two ports are bound together in a circuit. The destination port can be specified either by name or by PORT value.

Procedure Declaration

```
procedure CONNECT_CIRCUIT (
   STATUS           : out COND_VALUE_TYPE;         --optional
   SOURCE_PORT      : in  PORT_TYPE;
   DESTINATION_PORT : in  PORT_TYPE;               --optional
   DESTINATION_NAME : in  DESTINATION_NAME_TYPE;   --optional
   FULL_ERROR       : in  BOOLEAN;                 --optional
   CONNECT_DATA     : in  CIRCUIT_DATA_TYPE;       --optional
   ACCEPT_DATA      : out CIRCUIT_DATA_TYPE);      --optional
```

Arguments

STATUS
This argument receives the completion status of the CONNECT_CIRCUIT procedure.

SOURCE_PORT
This argument supplies a value that will form the caller's half of the circuit.

DESTINATION_PORT
This argument supplies a value giving the destination for the connection request message. (Such a value can be obtained from the REPLY_PORT argument of the RECEIVE procedure.) The argument can be omitted only if a destination name is supplied by the DESTINATION_NAME argument.

DESTINATION_NAME
This argument supplies the destination for the connection request message as a character-string name, usually a name established by the CREATE_NAME procedure. If the destination is specified this way, by means of

a NAME object, the DESTINATION_NAME argument is automatically translated to a destination port. If the DESTINATION_PORT argument is also specified, it overrides this argument. (Either this argument or the DESTINATION_PORT argument must be present.) The DESTINATION_NAME argument can also have the following forms:

```
NODE_NAME::LOCAL_PORT_NAME
```

```
NODE_NUMBER::LOCAL_PORT_NAME
```

for connection to a port in a VAXELN system (where the LOCAL_PORT_NAME is a local NAME established on the VAXELN node), or:

```
NODE_NUMBER::OBJECT
```

for connection to a DECnet–VAX (VAX/VMS) system (where object is the name of the object on the VAX/VMS system that will handle the connection).

FULL_ERROR
This argument enables or disables the implicit wait performed (with the SEND procedure) when the partner port is full. The default is FALSE, meaning that the sender waits until the partner port is not full. If TRUE is specified, the SEND procedure returns an error status or raises the corresponding exception if the partner port is full.

CONNECT_DATA
This argument supplies varying string data to the process receiving the connection request.

ACCEPT_DATA
This argument receives any varying string data supplied by the accepting process in its ACCEPT_CIRCUIT call.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_STATE	A port specified to the CONNECT_CIRCUIT procedure contains unreceived messages or has an incomplete CONNECT_CIRCUIT or ACCEPT_CIRCUIT procedure pending.
KER_CONNECT_TIMEOUT	The connection request was not accepted by the destination within the connection timeout limit; the connection timeout can be set by the System Builder Utility.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_DESTINATION	Neither a DESTINATION_PORT value nor port name was specified in the procedure call.
KER_NO_SUCH_NAME	The procedure call specified a NAME value for which there is no translation.
KER_NO_SUCH_PORT	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 CONNECT_CIRCUIT procedure.

COPY_FILE

The COPY_FILE procedure copies a file on a File Service disk volume.

Procedure Declaration

```
procedure COPY_FILE (
   SOURCE_FILE                 : in  FILE_NAME_TYPE;
   DESTINATION_FILE            : in  FILE_NAME_TYPE;
   STATUS                      : out COND_VALUE_TYPE;    --optional
   SOURCE_FILE_ERROR           : out BOOLEAN;            --optional
   BLOCK_MODE                  : out BOOLEAN;            --optional
   COUNT                       : out INTEGER;            --optional
   RESULTANT_SOURCE_FILE       : out FILE_NAME_TYPE;     --optional
   RESULTANT_DESTINATION_FILE  : out FILE_NAME_TYPE);    --optional
```

Arguments

SOURCE_FILE
This argument is a varying string of up to 255 characters giving the file specification of the file to be copied.

DESTINATION_FILE
This argument is a varying string of up to 255 characters giving the file specification of the copied file.

STATUS
This argument receives the completion status of the COPY_FILE procedure.

SOURCE_FILE_ERROR
This argument receives an indication that an error exists in one of the files. TRUE indicates that the error exists in the source file and FALSE indicates that the error exists in the destination file. This value should only be used if the STATUS argument returns an error status.

BLOCK_MODE
This argument receives an indication of the mode used to copy the file. TRUE indicates block mode and FALSE indicates record mode.

COUNT
This argument receives the number of blocks or records copied, as determined by the BLOCK_MODE argument.

RESULTANT_SOURCE_FILE
This argument is a varying string of up to 255 characters giving the resultant file name of the source file.

RESULTANT_DESTINATION_FILE
This argument is a varying string of up to 255 characters giving the resultant file name of the destination file.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_xxx	Any error status returned by the file service.

CREATE_AREA

The CREATE_AREA procedure creates a new area object or maps an existing area.

Procedure Declaration

```
procedure CREATE_AREA (
   STATUS  : out COND_VALUE_TYPE;      --optional
   AREA    : out AREA_TYPE;
   DATA    : out ADDRESS;
   SIZE    : in  UNSIGNED_LONGWORD;
   NAME    : in  AREA_NAME_TYPE;
   VIRTUAL : in  ADDRESS);             --optional
```

Arguments

STATUS
This argument receives the completion status of the CREATE_AREA procedure.

AREA
This argument receives the new AREA value.

DATA
This argument receives the address of the data portion of the area.

SIZE
This argument supplies the size in bytes of the area of memory.

NAME
This argument supplies the name for the area (as a 1- to 31-character string).

VIRTUAL
This argument supplies the base virtual address where the area is to be placed; it must be in P0 space.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The NAME argument has a bad length, the base virtual address or ending address is not in P0 space, or the virtual address is specified and does not match the area's specified virtual address.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_MEMORY	There were not enough memory pages to complete the operation.
KER_NO_OBJECT	No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available. The size of the system pool can be set by the System Builder Utility.
KER_NO_VIRTUAL	The necessary virtual address range is not available in the calling job's virtual address space.

CREATE_DEVICE

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

Procedure Declaration

```
procedure CREATE_DEVICE (
   STATUS            : out COND_VALUE_TYPE;          --optional
   DEVICE_NAME       : in  CREATE_DEVICE_NAME_TYPE;
   VECTOR_NUMBER     : in  VECTOR_NUMBER_TYPE;       --optional
   SERVICE_ROUTINE   : in  ADDRESS;                  --optional
   REGION_SIZE       : in  UNSIGNED_LONGWORD;        --optional
   REGION            : out ADDRESS;                  --optional
   REGISTERS         : out ADDRESS;                  --optional
   ADAPTER_REGISTERS : out ADDRESS;                  --optional
   VECTOR            : out ADDRESS;                  --optional
   PRIORITY          : out UNSIGNED_LONGWORD;        --optional
   DEVICE_ARRAY      : in  DEVICE_ARRAY_TYPE;
   DEVICE_COUNT      : in  INTEGER;                  --optional
   POWERFAIL_ROUTINE : in  ADDRESS);                 --optional
```

Arguments

STATUS
This argument receives the completion status of the CREATE_DEVICE procedure.

DEVICE_NAME
This argument supplies a 1- to 30-character string naming the device. The name must match one of the device names established with the System Builder Utility.

VECTOR_NUMBER
This argument supplies a value of VECTOR_NUMBER_TYPE from 1 to 128, specifying which vector of a multiple-interrupt vector device should

be connected to the interrupt service routine. If this argument is omitted, the default is 1 (first vector).

SERVICE_ROUTINE
This argument supplies the address of an interrupt service routine. To drive a device by polling instead of interrupts, omit this argument. If the interrupt address is supplied, the routine is called by the kernel on the occurrence of a device interrupt.

REGION_SIZE
This argument supplies the size of the communication region to be created, if any.

REGION
This argument receives a pointer to the communication region of the interrupt service routine. The region is zeroed by the CREATE_DEVICE procedure. The pointer is passed by the kernel to the interrupt service routine on the occurrence of a device interrupt. If the argument is omitted, no region is created, and the interrupt service routine (if any) receives the ADDRESS_ZERO value instead of the region's address. Note that every call with this argument creates a new communication region; if you use the same pointer variable from one call to another, the procedure will overwrite its previous value with the address of the new communication region.

REGISTERS
This argument receives a pointer to the first device control register. (The address of the first control register is part of the device description established with the System Builder Utility.) The pointer is passed to the interrupt service routine on the occurrence of a device interrupt. The argument can be omitted if no SERVICE_ROUTINE argument is supplied. Within the interrupt service routine, the corresponding parameter is declared to specify the type of the register pointer.

ADAPTER_REGISTERS
This argument receives a pointer to the first adapter control register.

VECTOR
This argument receives a pointer to the interrupt vector in the system control block. The interrupt vector address is part of the device description established with the System Builder Utility.

PRIORITY
This argument receives the interrupt priority level (IPL) of the device. The IPL is part of the device description established with the System Builder Utility.

DEVICE_ARRAY
This argument supplies an array of DEVICE objects, one for each unit.

DEVICE_COUNT
This argument receives the number of objects of type DEVICE_TYPE in the DEVICE_ARRAY argument.

POWERFAIL_ROUTINE
This argument supplies the address of an interrupt service routine that is called, before any process or interrupt service routine is restarted, when the processor enters a power recovery sequence.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MODE	The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.
KER_BAD_VALUE	The DEVICE argument is an array with more than 16 elements.
KER_DEVICE_CONNECTED	The device named in the CREATE_DEVICE call is already connected to a DEVICE_TYPE value.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_OBJECT	No free job object table entries are currently available. There are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.

KER_NO_SUCH_DEVICE	The device name specified in a CREATE_DEVICE call cannot be found in the list of devices created by the System Builder Utility.
KER_NO_SYSTEM_PAGE	No free system page table entries are currently available to map the I-O region.

CREATE_DIRECTORY

The CREATE_DIRECTORY procedure creates a directory on the specified File Service disk volume. Note that the directory must be created on a VAXELN disk volume; the procedure cannot create a directory on a non-VAXELN volume.

Procedure Declaration

```
procedure CREATE_DIRECTORY (
   DIRECTORY_NAME            : in  FILE_NAME_TYPE;
   STATUS                    : out COND_VALUE_TYPE;      --optional
   OWNER                     : out UIC_LONGWORD_TYPE;    --optional
   RESULTANT_DIRECTORY_NAME  : out FILE_NAME_TYPE);      --optional
```

Arguments

DIRECTORY_NAME
This argument is a varying string giving the file specification for the directory to be created. For example, "DISK$TEST:[DATA]" creates the directory DATA.DIR in the master file directory of the volume. Note that the procedure creates only the last directory in the specification; any intermediate directories (as in "DISK$TEST:[INTERMEDIATE.LAST]") must already exist.

STATUS
This argument receives the completion status of the CREATE_DIRECTORY procedure.

OWNER
This argument specifies the User Identification Code (UIC) of the owner of the file.

RESULTANT_DIRECTORY_NAME
This argument is a varying string of up to 255 characters giving the resultant file name of the created directory file.

CREATE_DIRECTORY

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_xxx	Any error status returned by the file service.

CREATE_EVENT

The CREATE_EVENT procedure creates and initializes an EVENT value.

Procedure Declaration

```
procedure CREATE_EVENT (
   STATUS        : out COND_VALUE_TYPE;     --optional
   EVENT         : out EVENT_TYPE;
   INITIAL_STATE : in  EVENT_STATE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of CREATE_EVENT.

EVENT
This argument receives the new EVENT value.

INITIAL_STATE
This argument supplies a value of the predeclared enumerated type EVENT_STATE_TYPE. It gives the initial state of the EVENT value.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.

CREATE_JOB

The CREATE_JOB procedure creates a new job, which executes a specified program image.

Note that the CREATE_JOB procedure runs a program image already built into the system (with the System Builder Utility or the LOAD_PROGRAM procedure); it cannot add a new image to a system.

Procedure Declaration

```
procedure CREATE_JOB (
   STATUS                   : out COND_VALUE_TYPE;      --optional
   JOB_PORT                 : out PORT_TYPE;
   PROGRAM_NAME_STRING_TYPE : in  PROGRAM_NAME_TYPE;
   EXIT_PORT                : in  PORT_TYPE;            --optional
   ARG1                     : in  JOB_ARGUMENT_TYPE;    --optional
   ARG2                     : in  JOB_ARGUMENT_TYPE;    --optional
   ARG3                     : in  JOB_ARGUMENT_TYPE;    --optional
   ARG4                     : in  JOB_ARGUMENT_TYPE;    --optional
   ARG5                     : in  JOB_ARGUMENT_TYPE;    --optional
   ARG6                     : in  JOB_ARGUMENT_TYPE);   --optional
```

Arguments

STATUS
This argument receives the completion status of the CREATE_JOB procedure.

JOB_PORT
This argument receives the new JOB PORT value. The value can be used by the caller of the CREATE_JOB procedure to send messages to the new job. The same value is returned within the new job by the JOB_PORT procedure.

PROGRAM_NAME_STRING_TYPE
This argument supplies a string that names the program the job is to run. The name is one of the program names that appears on the System Builder Utility's map listing, or one created by a call to the LOAD_PROGRAM procedure.

EXIT_PORT

This argument supplies a PORT value for termination notification. If this argument is present, a termination message is sent to the port when the new job terminates. (Note that the port must already be created.) The message data of the termination message is the value making up the completion status of the created job's master process. 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). If the argument is omitted, no message is sent.

ARG1,ARG2...ARG6

These arguments supply strings that are to be used as arguments to the program. Arguments can also be supplied to the program with the System Builder Utility, as part of a program description. (Any arguments supplied here override arguments supplied with the System Builder Utility.)

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_LENGTH	A string argument was too long.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_MEMORY	No free pages of physical memory are currently available.
KER_NO_PAGE_TABLE	No free process pages table is currently available; the number of process pages tables can be set by the System Builder Utility.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.
KER_NO_PORT	No free system port table entries are currently available; the size of the system port table can be set by the System Builder Utility.
KER_NO_SUCH_PROGRAM	No program with the specified name can be found in the program list created by the System Builder Utility.

CREATE_MESSAGE

The CREATE_MESSAGE procedure creates a MESSAGE object and its associated text variable.

Procedure Declaration

```
procedure CREATE_MESSAGE (
   STATUS       : out COND_VALUE_TYPE;        --optional
   MESSAGE      : out MESSAGE_TYPE;
   DATA_ADDRESS : out ADDRESS;
   MESSAGE_SIZE : in  UNSIGNED_LONGWORD);
```

Arguments

STATUS
This argument receives the completion status of the CREATE_MESSAGE procedure.

MESSAGE
This argument receives the new MESSAGE value.

DATA_ADDRESS
This argument receives the address of the message's data part. The returned ADDRESS value is valid in the current job; it becomes invalid if the message is sent or deleted.

MESSAGE_SIZE
This argument specifies the size in bytes of the message's data part.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_MEMORY	There was not enough physical memory to create the message data area.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.

CREATE_MUTEX

The CREATE_MUTEX procedure initializes a MUTEX variable for use in guarding the access to a shared variable or other shared resource. The initial state is "unlocked."

Procedure Declaration

```
procedure CREATE_MUTEX(
   STATUS :  out COND_VALUE_TYPE;     --optional
   MUTEX  :  out MUTEX_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the COND_VALUE_TYPE procedure.

MUTEX
This argument receives the new MUTEX value.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The specified initial count is greater than the maximum count.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.

CREATE_NAME

The CREATE_NAME procedure creates a NAME object that refers to a specified port. 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, the NAME object is not created, and an error status is returned.

Procedure Declaration

```
procedure CREATE_NAME (
   STATUS    : out COND_VALUE_TYPE;      --optional
   NAME      : out NAME_TYPE;
   PORT_NAME : in  PORT_NAME_TYPE;
   PORT      : in  PORT_TYPE;
   SCOPE     : in  NAME_SCOPE_TYPE);     --optional
```

Arguments

STATUS
This argument receives the completion status of the CREATE_NAME procedure.

NAME
This argument receives the new NAME value.

PORT_NAME
This argument supplies a string of 1 to 31 characters that is associated with this specified port.

PORT
This argument specifies the port that is associated with the name being created.

SCOPE
This argument supplies the enumerated value LOCAL or UNIVERSAL. It specifies that the new name is either local (valid only in this system or node) or universal (valid throughout the application or on any node).

LOCAL is the default. If the system does not contain the Network Service, all names are placed in the local name table.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_LENGTH	A string argument was too long.
KER_DUPLICATE	The CREATE_NAME procedure was called with a name that is a duplicate of an existing name.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.

CREATE_PORT

The CREATE_PORT procedure creates a message port and optionally specifies its maximum message capacity.

Procedure Declaration

```
procedure CREATE_PORT (
   STATUS        : out COND_VALUE_TYPE;     --optional
   PORT          : out PORT_TYPE;
   MESSAGE_LIMIT : in  INTEGER);            --optional
```

Arguments

STATUS
This argument receives the completion status of the CREATE_PORT procedure.

PORT
This argument receives the new PORT value.

MESSAGE_LIMIT
This argument supplies the maximum number of messages that can be queued to the port at one time. If the limit is exceeded, further messages are lost unless the port is connected in a circuit. The default value is 4.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.
KER_NO_PORT	No free system port table entries are currently available; the size of the system port table can be set by the System Builder Utility.

CREATE_PROCESS

The CREATE_PROCESS procedure creates a new process executing a specified procedure.

Procedure Declaration

```
procedure CREATE_PROCESS (
   STATUS        : out COND_VALUE_TYPE;        --optional
   PROCESS       : out PROCESS_TYPE;
   ENTRY_ROUTINE : in  ADDRESS;
   EXIT_STATUS   : out EXIT_STATUS_TYPE);      --optional
```

Arguments

STATUS
This argument receives the completion status of the CREATE_PROCESS procedure.

PROCESS
This argument receives the new PROCESS value.

ENTRY_ROUTINE
This argument supplies the address of the procedure to be executed in the new process.

EXIT_STATUS
This argument receives the final status of the created process. Such a value can be returned with the EXIT procedure; by convention, odd-numbered values indicate success and even-numbered values indicate errors (not necessarily fatal). If the argument is omitted, neither status is returned.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_COUNT	The procedure call specified an incorrect number of arguments.
KER_BAD_VALUE	The exit status variable is in P1 space.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_MEMORY	No free pages of physical memory are currently available.
KER_NO_PAGE_TABLE	No free process page table is currently available; the number of process pages tables can be set by the System Builder Utility.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.
KER_NO_STATUS	The process was deleted; therefore, no exit STATUS value is available to return (This value is returned only as an exit status, in the CREATE_PROCESS EXIT argument).

CREATE_SEMAPHORE

The CREATE_SEMAPHORE procedure creates and initializes a semaphore.

Procedure Declaration

```
procedure CREATE_SEMAPHORE (
   STATUS        : out COND_VALUE_TYPE;     --optional
   SEMAPHORE     : out SEMAPHORE_TYPE;
   INITIAL_COUNT : in  INTEGER;
   MAXIMUM_COUNT : in  INTEGER);
```

Arguments

STATUS
This argument receives the completion status of the CREATE_SEMAPHORE procedure.

SEMAPHORE
This argument receives the new SEMAPHORE value.

INITIAL_COUNT
This argument supplies the initial semaphore count. The initial count must not exceed the maximum count.

MAXIMUM_COUNT
This argument supplies the maximum semaphore count. Signaling the semaphore beyond this count is an error.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The specified initial count is greater than the maximum count.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.
KER_NO_POOL	No free system pool is currently available; the size of the system pool can be set by the System Builder Utility.

CURRENT_PROCESS

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

Procedure Declaration

```
procedure CURRENT_PROCESS (
   STATUS  : out COND_VALUE_TYPE;     --optional
   PROCESS : out PROCESS_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the CURRENT_PROCESS procedure. KER_SUCCESS is the only possible status.

PROCESS
This argument receives the PROCESS value.

Status Value

KER_SUCCESS	The procedure completed successfully.

DECLARE_EXIT_HANDLER

The DECLARE_EXIT_HANDLER procedure allows you to declare an exit handler for a program. The named exit handler procedure is called upon the termination of a job with the EXIT procedure.

Procedure Declaration

```
procedure DECLARE_EXIT_HANDLER (
   EXIT_HANDLER : in  ADDRESS;
   EXIT_CONTEXT : in  EXIT_CONTEXT_TYPE);     --optional
```

Arguments

EXIT_HANDLER
This argument supplies the address of an exit handler routine to be called upon the termination of a job with the EXIT procedure.

EXIT_CONTEXT
This argument will be passed to the specified exit handler routine when it is invoked.

Status Value

KER_SUCCESS	The procedure completed successfully.

DELETE_AREA

The DELETE_AREA procedure deletes an AREA object, unmapping it from the calling process's address space.

Procedure Declaration

```
procedure DELETE_AREA (
   STATUS : out COND_VALUE_TYPE;     --optional
   AREA   : in AREA_NAME_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_AREA procedure.

AREA
This argument specifies the area to delete.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The AREA argument is invalid or refers to a deleted object.
KER_NOACCESS	An argument specified is not accessible to the calling program.

DELETE_DEVICE

The DELETE_DEVICE procedure deletes the specified device.

Procedure Declaration

```
procedure DELETE_DEVICE (
   STATUS : out COND_VALUE_TYPE;     --optional
   DEVICE : in  DEVICE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_DEVICE procedure.

DEVICE
This argument supplies the DEVICE to be deleted.

Notes:

When a DEVICE object is deleted, the memory used for its communication region is deleted, and any pointers to that memory 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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_STATE	A device specified to be deleted has an interrupt pending.
KER_BAD_VALUE	The DEVICE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_EVENT

The DELETE_EVENT procedure deletes an EVENT object from the system. Any waiting processes are removed from their wait states immediately; the status of WAIT_ANY or WAIT_ALL is KER_BAD_VALUE.

Procedure Declaration

```
procedure DELETE_EVENT (
   STATUS : out COND_VALUE_TYPE;     --optional
   EVENT  : in  EVENT_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_EVENT procedure.

EVENT
This argument supplies the device to be deleted.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The DEVICE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_FILE

The DELETE_FILE procedure deletes a file from a mounted disk volume.

Procedure Declaration

```
procedure DELETE_FILE (
   FILE_NAME           : in  FILE_NAME_TYPE;
   STATUS              : out COND_VALUE_TYPE;     --optional
   RESULTANT_FILE_NAME : out FILE_NAME_TYPE);     --optional
```

Arguments

FILE_NAME
This argument is a string of up to 255 characters giving the file specification of the file to be deleted. The file specification must have an explicit version number or a semicolon or period to indicate the most recent version. For example, "TEST.DAT;23" designates version 23 to be deleted; "TEST.DAT;" and "TEST.DAT." both designate the most recent version of the file.

STATUS
This argument receives the completion status of the DELETE_FILE procedure.

RESULTANT_FILE_NAME
This argument is a string of up to 255 characters giving the resultant file name of the deleted file.

Status Value

ELN_xxx	Any error status returned by the file service.

DELETE_MESSAGE

The DELETE_MESSAGE procedure deletes the specified message object. Once deleted, the message is unavailable for sending or receiving, and any pointers to the message's text variable become invalid.

Procedure Declaration

```
procedure DELETE_MESSAGE (
   STATUS  : out COND_VALUE_TYPE;     --optional
   MESSAGE : in  MESSAGE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_MESSAGE procedure.

MESSAGE
This argument identifies the message to be deleted.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_STATE	A device specified to be deleted has an interrupt pending.
KER_BAD_VALUE	The MESSAGE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_MUTEX

The DELETE_MUTEX procedure deletes the specified mutex.

Procedure Declaration

```
procedure DELETE_MUTEX (
   STATUS : out COND_VALUE_TYPE;     --optional
   MUTEX : in  MUTEX_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_MUTEX procedure.

MUTEX
This argument identifies the MUTEX value to be deleted.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The VALUE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_NAME

The DELETE_NAME procedure deletes the specified name.

Procedure Declaration

```
procedure DELETE_NAME (
   STATUS : out COND_VALUE_TYPE;     --optional
   NAME   : in  NAME_TYPE);
```

Arguments

STATUS

This argument receives the completion status of the DELETE_NAME procedure.

NAME

This argument identifies the NAME value to be deleted.

Notes:

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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The VALUE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_PORT

The DELETE_PORT procedure deletes the specified port.

Procedure Declaration

```
procedure DELETE_PORT (
   STATUS : out COND_VALUE_TYPE;     --optional
   PORT   : in  PORT_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_PORT procedure.

PORT
This argument identifies the port to be deleted.

Notes:

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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The VALUE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_PROCESS

The DELETE_PROCESS procedure deletes the specified process.

Procedure Declaration

```
procedure DELETE_PROCESS (
   STATUS  : out COND_VALUE_TYPE;     --optional
   PROCESS : in  PROCESS_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_PROCESS procedure.

PROCESS
This argument identifies the process to be deleted.

Notes:

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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The VALUE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DELETE_SEMAPHORE

The DELETE_SEMAPHORE procedure deletes the specified semaphore. Any waiting processes are removed from their wait states immediately; the status of WAIT_ANY or WAIT_ALL is KER_BAD_VALUE.

Procedure Declaration

```
procedure DELETE_SEMAPHORE (
   STATUS    : out COND_VALUE_TYPE;     --optional
   SEMAPHORE : in  SEMAPHORE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DELETE_SEMAPHORE procedure.

SEMAPHORE
This argument identifies the semaphore to be deleted.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The VALUE argument is invalid or refers to an object that was deleted.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

DIRECTORY_CLOSE

The DIRECTORY_CLOSE procedure closes an existing directory on a mounted disk volume.

Procedure Declaration

```
procedure DIRECTORY_CLOSE (
   DIR_CONTEXT : in  DIR_CONTEXT_ACCESS_TYPE;
   STATUS      : out COND_VALUE_TYPE);          --optional
```

Arguments

DIR_CONTEXT
This argument supplies the address of the context value that was stored by a previous call to the DIRECTORY_OPEN procedure.

STATUS
This argument receives the completion status of the DIRECTORY_CLOSE procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_NMF	There are no more files.
ELN_xxx	Any error status returned by the file service.

DIRECTORY_LIST

The DIRECTORY_LIST procedure obtains the next resultant file name from a mounted disk directory.

Procedure Declaration

```
procedure DIRECTORY_LIST (
   DIR_CONTEXT     : in  DIR_CONTEXT_ACCESS_TYPE;
   DIRECTORY_NAME  : out FILE_NAME_TYPE;
   FILE_NAME       : out FILE_NAME_TYPE;
   STATUS          : out COND_VALUE_TYPE;            --optional
   FILE_ATTRIBUTES : in  ADDRESS);                   --optional
```

Arguments

DIR_CONTEXT
This argument supplies the address of the context value that was stored by a previous call to the DIRECTORY_OPEN procedure.

DIRECTORY_NAME
This argument receives the resultant directory specification (when a wildcard directory specification is used in the DIRECTORY_OPEN procedure). That is, if more than one directory is affected by the DIRECTORY_LIST procedure, the directory name will change.

FILE_NAME
This argument receives the file name not the volume or directory name.

STATUS
This argument receives the completion status of the DIRECTORY_LIST procedure. An exception is raised if the procedure does not succeed and this argument is omitted.

FILE_ATTRIBUTES
This argument supplies the address of a record of type FILE_ATTRIBUTES_TYPE that is filled in by the DIRECTORY_LIST procedure with the attributes of the file found.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_NMF	There are no more files.
ELN_xxx	Any error status returned by the file service.

DIRECTORY_OPEN

The DIRECTORY_OPEN procedure opens an existing directory on a mounted disk volume in preparation for a DIRECTORY_LIST operation.

Procedure Declaration

```
procedure DIRECTORY_OPEN (
   DIR_CONTEXT     : in  DIR_CONTEXT_ACCESS_TYPE;
   SEARCH_NAME     : in  FILE_NAME_TYPE;
   VOLUME_NAME     : out FILE_NAME_TYPE;
   DIRECTORY_NAME  : out FILE_NAME_TYPE;
   STATUS          : out COND_VALUE_TYPE;           --optional
   SERVER_NAME     : out SERVER_NAME_TYPE;            --optional
   FILE_ATTRIBUTES : out ADDRESS);                  --optional
```

Arguments

DIR_CONTEXT

This argument specifies the address of a variable that receives a context value to be used in calls to the DIRECTORY_LIST and DIRECTORY_CLOSE procedures.

SEARCH_NAME

This argument supplies a varying string of up to 255 characters giving a specification of an existing directory. This string is used to search for the directory string as follows:

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

The file name, type, and version can use the "wildcard" characters, percent (%) and asterisk (*), as in VAX/VMS file specifications. The percent (%) character matches any character in the corresponding position; the asterisk (*) 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 volumes, in directory specifications.

If the directory is on a VAX/VMS volume, the asterisk (*) and ellipsis (...) can be used in the directory specification. The ellipsis following a directory name matches all directories below including the named directory. For example, the string:

```
[testdata...]*.*;*
```

VOLUME_NAME
This argument receives the volume name if the procedure is successful.

DIRECTORY_NAME
This argument receives the directory name if the procedure is successful.

STATUS
This argument receives the completion status of the DIRECTORY_OPEN procedure. An exception is raised if the procedure does not succeed and this argument is omitted.

SERVER_NAME
This argument is a varying string of up to 64 characters that receives the resultant node specification or server process port name.

FILE_ATTRIBUTES
The address of a record of type FILE_ATTRIBUTES_TYPE that is filled in by the DIRECTORY_LIST procedure with the attributes of the file found.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_xxx	Any error status returned by the file service.

DISABLE_ASYNCH_EXCEPTION

The DISABLE_ASYNCH_EXCEPTION procedure prevents the delivery of asynchronous exceptions (such as KER_QUIT_SIGNAL and KER_POWER_SIGNAL) to the calling process.

Procedure Declaration

```
procedure DISABLE_ASYNCH_EXCEPTION (
   STATUS : out COND_VALUE_TYPE);     --optional
```

Arguments

STATUS
This argument receives the completion status of the DISABLE_ASYNCH_EXCEPTION procedure. The only possible status is KER_SUCCESS.

Status Value

KER_SUCCESS	The procedure completed successfully.

DISABLE_INTERRUPT

The DISABLE_INTERRUPT procedure prevents interrupts from a device, by raising the interrupt priority level (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. The DISABLE_INTERRUPT procedure can be called only from programs running in kernel mode.

Procedure Declaration

```
procedure DISABLE_INTERRUPT (
   PRIORITY : in IPL_TYPE);
```

Arguments

PRIORITY
This argument supplies an integer in the range of 1 to 31, giving the new interrupt priority level.

Notes:

The current interrupt priority level is part of the processor-wide state of a VAX processor. Disabling interrupts of a given 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 the ENABLE_INTERRUPT procedure.

If 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, but, if it occurs with interrupts disabled, continuation from the exception causes unpredictable effects.

Status Values

None

DISABLE_SWITCH

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 the ENABLE_SWITCH procedure.

Procedure Declaration

```
procedure DISABLE_SWITCH (
   STATUS : out COND_VALUE_TYPE);    --optional
```

Arguments

STATUS

This argument receives the completion status of the DISABLE_SWITCH procedure.

Notes:

Process switching is reenabled automatically if the process calls the EXIT_PROCESS procedure or deletes itself. See the description of the ENABLE_SWITCH procedure for more information.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_COUNT_OVERFLOW	The DISABLE_SWITCH procedure was called more times than the ENABLE_SWITCH procedure.

DISCONNECT_CIRCUIT

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 the DISCONNECT_CIRCUIT procedure.

Procedure Declaration

```
procedure DISCONNECT_CIRCUIT (
   STATUS  : out COND_VALUE_TYPE;     --optional
   PORT    : in  PORT_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the DISCONNECT_CIRCUIT procedure.

PORT
This argument supplies a PORT value representing the caller's half of the circuit.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_STATE	A port specified to the DISCONNECT_CIRCUIT procedure was not connected.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_SUCH_PORT	No port with the specified value can be found in the system or network, or else the port is not owned by the current job, as required by the DISCONNECT_CIRCUIT procedure.

DISMOUNT_TAPE_VOLUME

The DISMOUNT_TAPE_VOLUME procedure dismounts a File Service tape on the specified tape drive. The procedure must be called on the same node that has the File Service tape.

Procedure Declaration

```
procedure DISMOUNT_TAPE_VOLUME (
   DEVICE_NAME : in  DEVICE_NAME_TYPE;
   UNLOAD      : in  BOOLEAN;            --optional
   STATUS      : out COND_VALUE_TYPE);   --optional
```

Arguments

DEVICE_NAME
This argument supplies a varying string of up to 31 characters naming the tape drive; for example, 'MUA0' for drive 0 on tape controller MUA.

UNLOAD
This argument specifies whether the tape is unloaded. The default is FALSE, implying that the tape is rewound but not unloaded when the volume is dismounted.

STATUS
This argument receives the completion status of the DISMOUNT_TAPE_VOLUME procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_FILE_xxx	Any error status returned by the file service.

DISMOUNT_VOLUME

The DISMOUNT_VOLUME procedure dismounts a File Service volume on the specified device. The procedure must be called on the same node that has the File Service volume. A dismounted disk can be used for logical I-O.

Procedure Declaration

```
procedure DISMOUNT_VOLUME (
   DEVICE_NAME : in  DEVICE_NAME_TYPE;
   STATUS      : out COND_VALUE_TYPE);     --optional
```

Arguments

DEVICE_NAME
This argument supplies a varying string of up to 31 characters naming the device; for example, 'DQA1' for drive 1 on disk controller DQA.

STATUS
This argument receives the completion status of the DISMOUNT_VOLUME procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_FILE_xxx	Any error status returned by the file service.

DLV_INITIALIZE

The DLV_INITIALIZE procedure readies an Asynchronous Serial Line Controller for input-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.

Procedure Declaration

```
procedure DLV_INITIALIZE (
   DEVICE_NAME      : in  DEVICE_NAME_TYPE;
   IDENTIFIER       : out DLV_IDENTIFIER_TYPE;
   MAXIMUM_LENGTH   : in  INTEGER;                  --optional
   STRING_MODE      : in  BOOLEAN;                  --optional
   USE_POLLING      : in  BOOLEAN);                 --optional
```

Arguments

DEVICE_NAME
This argument supplies a varying string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder Utility.

IDENTIFIER
This argument receives an identifier to be used to identify this device in subsequent calls to DLV_READ_STRING, DLV_READ_BLOCK, and DLV_WRITE_STRING procedures.

MAXIMUM_LENGTH
This argument supplies the maximum string or block length, in bytes, that will be read or written. The default value is 256.

STRING_MODE
This argument is a Boolean expression. TRUE causes the serial line to be used in string mode; FALSE causes it to be used in block mode. The default value is TRUE. String mode means the input is obtained by calling the DLV_READ_STRING procedure and will always be terminated by

a carriage return character. Block mode means the input is obtained by calling the DLV_READ_BLOCK procedure and will be fixed-length blocks of data, with no carriage return checking performed.

USE_POLLING

This argument is a Boolean expression. TRUE means the read procedures will poll the device register; FALSE means the read procedures will use interrupts. Polling is always done at the device's interrupt priority level, which is four for the DLV. The default value is FALSE.

Status Values

None

DLV_READ_BLOCK

The DLV_READ_BLOCK procedure causes characters to be read from a DLV device 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 the DLV_INITIALIZE procedure.

Procedure Declaration

```
procedure DLV_READ_BLOCK (
   IDENTIFIER  : in  DLV_IDENTIFIER_TYPE;
   BLOCK_DESCR : in  DLV_BLOCK_DESCR_TYPE;
   TIMEOUT     : in  DATE_TIME_TYPE);          --optional
```

Arguments

IDENTIFIER
This argument identifies the serial line device to be read; this value is the one returned in the identifier parameter after a call to the DLV_INITIALIZE procedure.

BLOCK_DESCR
This argument supplies a descriptor of the array into which the data is to be read. See the description of the type DLV_BLOCK_DESCR_TYPE in Chapter 8 for more information.

TIMEOUT
This LARGE_INTEGER argument specifies a time interval that is the maximum time allowed for the block of characters to be read. If the timeout occurs, the block is returned incomplete. The default value is zero, which implies no timeout.

Status Values

None

DLV_READ_STRING

The DLV_READ_STRING procedure 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 the DLV_INITIALIZE procedure.

Procedure Declaration

```
procedure DLV_READ_STRING (
   IDENTIFIER      : in  DLV_IDENTIFIER_TYPE;
   STRING_DESCR    : in  VS_DESCR_TYPE);
```

Arguments

IDENTIFIER
This argument identifies the serial line device to be read; this value is the one returned in the identifier parameter after a call to the DLV_INITIALIZE procedure.

STRING_DESCR
This argument specifies a descriptor of a varying string that is to receive the string read from the serial line.

Status Values

None

DLV_WRITE_STRING

The DLV_WRITE_STRING procedure causes the 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.

Procedure Declaration

```
procedure DLV_WRITE_STRING (
   IDENTIFIER : in  DLV_IDENTIFIER_TYPE;
   STRING     : in  DLV_STRING_TYPE);
```

Arguments

IDENTIFIER
This argument identifies the serial line device to be read; this value is the one returned in the identifier parameter after a call to the DLV_INITIALIZE procedure.

STRING
This argument specifies the character string to be written to the serial line.

Status Values

None

DRV_INITIALIZE

The DRV_INITIALIZE procedure readies a Parallel Line Interface Controller for input-ouput and creates all needed data structures. This procedure must be called once for each DRV controller used.

Procedure Declaration

```
procedure DRV_INITIALIZE (
   DEVICE_NAME  : in  DEVICE_NAME_TYPE;
   IDENTIFIER   : out DRV_IDENTIFIER_TYPE;
   BUFFER_PTR   : out ADDRESS;
   BUFFER_SIZE  : in  INTEGER;
   OUTPUT_PORTS : in  DRV_PORT_SET_TYPE;
   USE_POLLING  : in  BOOLEAN);
```

Arguments

DEVICE_NAME

This argument supplies a varying string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder Utility.

IDENTIFIER

This argument receives an identifier to be used to identify this device in subsequent calls to the DRV_READ and DRV_WRITE procedures.

BUFFER_PTR

This argument receives the address of the input-output buffer. The buffer is a two-dimensional array of data words; the first array index specifies the port number and the second array index specifies a data word number. The input-output buffer is allocated by the DRV_INITIALIZE procedure; it will receive all data read from the device and should be filled with all data to be written to a port.

BUFFER_SIZE

This argument specifies the size (the number of 16-bit words) of the input-output buffer for each port allocated in the buffer array. This argument is also an upper bound on the buffer array's second index.

OUTPUT_PORTS

This argument specifies the set of port numbers of type DRV_PORT_SET_TYPE that are to be used for output instead of input. If a port is specified as an output port, the port's "DIR" bit is set in the port register; otherwise, it is cleared.

USE_POLLING

This argument is a Boolean expression. TRUE means the read procedures will poll the device register; FALSE means the procedures will use interrupts. Polling is always done at the device's interrupt priority level, which is four for the DRV11-J.

Status Values

None

DRV_READ

The DRV_READ procedure causes data words to be read from the specified parallel port. The resulting data is stored in the buffer pointed to by the BUFFER_PTR parameter returned by the DRV_INITIALIZE procedure.

Procedure Declaration

```
procedure DRV_READ (
   IDENTIFIER : in  DRV_IDENTIFIER_TYPE;
   PORT       : in  DRV_PORT_INDEX_TYPE;
   WORD_COUNT : in  INTEGER);
```

Arguments

IDENTIFIER
This argument supplies a value that identifies the device to be read; this value is the one returned in the identifier parameter after a call to the DRV_INITIALIZE procedure.

PORT
This argument supplies a value specifying which port to read.

WORD_COUNT
This argument supplies a value specifying the number of 16-bit words to be read.

Status Values

None

DRV_WRITE

The DRV_WRITE procedure 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_PTR parameter returned by the DRV_INITIALIZE procedure.

Procedure Declaration

```
procedure DRV_WRITE (
   IDENTIFIER : in  DRV_IDENTIFIER_TYPE;
   PORT       : in  DRV_PORT_INDEX_TYPE;
   WORD_COUNT : in  INTEGER);
```

Arguments

IDENTIFIER
This argument supplies a value that identifies the serial line device to be written to; this value is the one returned in the identifier parameter after a call to DRV_INITIALIZE.

PORT
This argument supplies a value specifying which port will be written to.

WORD_COUNT
This argument supplies an INTEGER value specifying the number of 16-bit words to be written.

Status Values

None

ENABLE_ASYNCH_EXCEPTION

The ENABLE_ASYNCH_EXCEPTION procedure allows the delivery of asynchronous exceptions (such as KER_QUIT_SIGNAL and KER_POWER_SIGNAL) to the calling process. Asynchronous exceptions are enabled by default and may be reenabled only after being explicitly disabled.

Procedure Declaration

```
procedure ENABLE_ASYNCH_EXCEPTION (
   STATUS : out COND_VALUE_TYPE);     --optional
```

Arguments

STATUS
This argument receives the completion status of the ENABLE_ASYNCH_EXCEPTION procedure. The only possible status is KER_SUCCESS.

Status Value

KER_SUCCESS	The procedure completed successfully.

ENABLE_INTERRUPT

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

Procedure Declaration

```
procedure ENABLE_INTERRUPT;
```

Arguments

There are no arguments.

Status Values

None

ENABLE_SWITCH

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.

Procedure Declaration

```
procedure ENABLE_SWITCH (
   STATUS : out COND_VALUE_TYPE);    --optional
```

Arguments

STATUS

This argument receives the completion status of the ENABLE_SWITCH procedure.

Notes:

The ENABLE_SWITCH and DISABLE_SWITCH procedures have a feature that allows them to be called with reasonable effects from nested routines. The implementation uses a counter that is incremented whenever the DISABLE_SWITCH procedure is called and decremented whenever the ENABLE_SWITCH procedure is called. Switching is enabled only when the number of calls to the ENABLE_SWITCH procedure is equal to the number of calls to the DISABLE_SWITCH procedure for a given process. For example:

```
procedure A;
begin
   DISABLE_SWITCH;
   .
   .
   .
   ENABLE_SWITCH;
end;
```

```
procedure B;
begin
    DISABLE_SWITCH;
    .
    .
    .
    A; -- Call procedure A
    ENABLE_SWITCH;
end;
```

Here, Procedure B disables process switching and then calls Procedure A. Procedure A also disables process switching during its execution and then calls the ENABLE_SWITCH procedure. This call does not reenable process switching; however, since that would cause an error upon returning to Procedure B. Process switching is reenabled only when Procedure B calls the ENABLE_SWITCH procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_COUNT_UNDERFLOW	The ENABLE_SWITCH procedure was called more times than the DISABLE_SWITCH procedure.

ENTER_KERNEL_CONTEXT

The ENTER_KERNEL_CONTEXT procedure executes the specified user routine in kernel mode.

Procedure Declaration

```
procedure ENTER_KERNEL_CONTEXT (
   STATUS        : out COND_VALUE_TYPE;
   ROUTINE       : in  ADDRESS;
   ARGUMENT_LIST : in  ARGUMENT_LIST_TYPE);
```

Arguments

STATUS

This argument receives the function value of the call to the ROUTINE specified by the ROUTINE argument.

ROUTINE

This argument supplies the address of the routine to be executed in kernel mode. The routine should be a function that returns a value of COND_VALUE_TYPE as the result.

ARGUMENT_LIST

This argument supplies a pointer to the array of longwords that will be passed as arguments to the user's function, which is passed as the preceding ROUTINE argument. This argument block should be in the standard VAX/VMS format; a longword containing both the argument count and the argument longwords themselves.

Note:

The result returned from the call to ROUTINE is returned in the STATUS argument.

EXIT_PROCESS

The EXIT_PROCESS procedure causes an immediate exit from the calling process. If the calling process is the master process, all the objects it owns (including subprocesses) are deleted. The status of open files is unpredictable.

Procedure Declaration

```
procedure EXIT_PROCESS (
   STATUS      : out COND_VALUE_TYPE;      --optional
   EXIT_STATUS : in  EXIT_STATUS_TYPE);    --optional
```

Arguments

STATUS
This argument receives the completion status of the EXIT_PROCESS procedure. The only possible status is KER_SUCCESS.

EXIT_STATUS
This argument supplies the exit status of the current process to its creator. If it is omitted, the creating process receives a STATUS value indicating that KER_NO_STATUS was returned.

Notes:

If process switching was disabled by the calling process, it is reenabled automatically when the EXIT_PROCESS procedure is called.

Status Value

KER_SUCCESS	The procedure completed successfully.

FREE_MAP

The FREE_MAP procedure frees a set of previously allocated UNIBUS map registers. It can be called only from a program running in kernel mode. Any pointers to the freed registers become invalid.

Procedure Declaration

```
procedure FREE_MAP (
   STATUS  : out COND_VALUE_TYPE;     --optional
   COUNT   : in  INTEGER;
   NUMBER  : in  INTEGER;
   DEVICE  : in  DEVICE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the FREE_MAP procedure.

COUNT
This argument supplies the number (count) of contiguous map registers to be freed.

NUMBER
This argument supplies the map register number of the first map register, such as the one returned by the ALLOCATE_MAP procedure.

DEVICE
This argument identifies the device for which the registers are freed.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MODE	The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.
KER_BAD_VALUE	The DEVICE argument is invalid or identifies a deleted device.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

FREE_MEMORY

The FREE_MEMORY procedure frees a region of memory previously allocated by the ALLOCATE_MEMORY procedure. Any pointers to the freed memory become invalid.

Procedure Declaration

```
procedure FREE_MEMORY (
   STATUS          : out COND_VALUE_TYPE;      --optional
   SIZE            : in  UNSIGNED_LONGWORD;
   VIRTUAL_ADDRESS : in  ADDRESS);
```

Arguments

STATUS
This argument receives the completion status of the FREE_MEMORY procedure.

SIZE
This argument supplies the number of bytes of memory to be freed. This value is rounded up to the next 512-byte page.

VIRTUAL_ADDRESS
This argument supplies the starting virtual address of the memory, as returned by the ALLOCATE_MEMORY procedure. This value is truncated to a 512-byte page address.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The VIRTUAL_ADDRESS argument is not in the calling job's address space.

FREE_PATH

The FREE_PATH procedure frees a previously allocated UNIBUS adapter buffered datapath. This procedure can only be called from programs running in kernel mode. The VAX-11/750 is the only processor supported by the VAXELN system that has UNIBUS buffered datapaths.

Procedure Declaration

```
procedure FREE_PATH (
   STATUS  : out COND_VALUE_TYPE;      --optional
   NUMBER  : in  INTEGER;
   DEVICE  : in  DEVICE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the FREE_PATH procedure.

NUMBER
This argument is an INTEGER value that supplies the datapath register number, such as the one returned by the ALLOCATE_PATH procedure.

DEVICE
This argument supplies the DEVICE value that identifies the device for which the datapath is freed.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MODE	The procedure was called from a program that was not running in kernel mode; kernel mode is required for this procedure.
KER_BAD_VALUE	The DEVICE argument is invalid or identifies a deleted device.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

GET_TIME

The GET_TIME procedure returns the current system time.

Procedure Declaration

```
procedure GET_TIME (
   STATUS : out COND_VALUE_TYPE;     --optional
   TIME   : out DATE_TIME_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the GET_TIME procedure.

TIME
This argument receives a value representing the time of day.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_TIME_NOT_SET	The time of day has not been set. This is an alternate success status.
KER_BAD_COUNT	The procedure call specified an incorrect number of arguments.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

GET_USER

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.

Procedure Declaration

```
procedure GET_USER (
   STATUS    : out COND_VALUE_TYPE;       --optional
   CIRCUIT   : in  PORT_TYPE              --optional
   USER_NAME : out AUTH_STRING_TYPE;
   UIC       : out UIC_LONGWORD_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the GET_USER procedure.

CIRCUIT
This argument supplies a PORT value specifying the partner process's port in the 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 the CONNECT_CIRCUIT procedure; that is, the GET_USER procedure can only provide information about the object of a connection, not the subject.

USER_NAME
This argument receives a string of up to 20 characters that is the user name of either the calling process or the partner process.

UIC
This argument is an INTEGER value that supplies 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"), the GET_USER procedure returns zero for the UIC parameter.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_LENGTH	The USER_NAME argument is too long.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

INITIALIZATION_DONE

The INITIALIZATION_DONE procedure informs the kernel that the calling program has completed an initialization sequence, and other programs can be started.

Procedure Declaration

```
procedure INITIALIZATION_DONE (
   STATUS : out COND_VALUE_TYPE);    --optional
```

Arguments

STATUS
This argument receives the completion status of the INITIALIZATION_DONE procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_INITIALIZATION	The calling program was specified with a "no Init required" characteristic.

INIT_TAPE_VOLUME

The INIT_TAPE_VOLUME procedure initializes a File Service tape for use as a file-structured volume. Tapes must be initialized before they are used. The procedure is similar to the VAX/VMS command INITIALIZE, as used for tape volumes. (For additional information, consult the VAX/VMS documentation.)

You can initialize any volume on any node running a VAXELN system, but only if the volume is not mounted or open for logical input-output.

Procedure Declaration

```
procedure INIT_TAPE_VOLUME (
   DEVICE_NAME          : in  DEVICE_NAME_TYPE;
   VOLUME_NAME          : in  TAPE_VOLUME_NAME_TYPE;
   DENSITY              : in  DENSITY_TYPE;
   STATUS               : out COND_VALUE_TYPE);           --optional
```

Arguments

DEVICE_NAME
This argument supplies a varying string of up to 31 characters, giving the device specification of the tape drive; for example, 'DQA1' for drive 1 of controller DQA. The node must be specified explicitly for a drive on another node.

VOLUME_NAME
This argument supplies a varying string of up to 12 characters, giving the volume label for the disk.

DENSITY
This argument is an INTEGER value that supplies the density (in bytes per inch) that the tape will be initialized to. If the specified density is not supported, the tape will be initialized to the supported density closest to it. The default density is the highest density supported by the specified tape drive.

STATUS
This argument is an INTEGER variable that receives the completion status of the INIT_TAPE_VOLUME procedure.

Notes:

VAXELN processes that create files on the tape are not restricted by the privileges implied by such parameters as VOLUME_PROTECTION. These parameters are provided for compatibility with VAX/VMS systems.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_TAPE_DEVINUSE	The device is in use by another process.
ELN_TAPE_DEVMOUNT	The device is already mounted.
ELN_TAPE_DEVERROR	A device error occurred.
ELN_TUTL_INVCHRVOL	There is an invalid character in the volume label.
ELN_FILE_xxx	Any other error returned by disk service.
KER_NO_SUCH_DEVICE	The device does not exist.

INIT_VOLUME

The INIT_VOLUME procedure initializes a File Service disk for use as a file-structured volume. Disks must be initialized before they are used. The procedure is similar to the VAX/VMS command INITIALIZE, as used for disk volumes. (For additional information, consult the VAX/VMS documentation.)

You can initialize any volume on any node running a VAXELN system, but only if the volume is not mounted or open for logical input-output.

Procedure Declaration

```
procedure INIT_VOLUME (
   DEVICE_NAME           : in  DEVICE_NAME_TYPE;
   VOLUME_NAME           : in  DISK_VOLUME_NAME_TYPE;
   DEFAULT_EXTENSION     : in  UNSIGNED_WORD;              --optional
   USERNAME              : in  AUTH_STRING_TYPE;           --optional
   OWNER                 : in  UIC_LONGWORD_TYPE;          --optional
   VOLUME_PROTECTION     : in  FILE_PROTECTION_TYPE;       --optional
   FILE_PROTECTION       : in  FILE_PROTECTION_TYPE;       --optional
   RECORD_PROTECTION     : in  FILE_PROTECTION_TYPE;       --optional
   ACCESSED_DIRECTORIES  : in  UNSIGNED_BYTE;              --optional
   MAXIMUM_FILES         : in  INTEGER;                    --optional
   USER_DIRECTORIES      : in  UNSIGNED_WORD;              --optional
   FILE_HEADERS          : in  INTEGER;                    --optional
   WINDOWS               : in  UNSIGNED_BYTE;              --optional
   CLUSTER_SIZE          : in  UNSIGNED_WORD;              --optional
   INDEX_POSITION        : in  DISK_INDEX_POSITION_TYPE;   --optional
   DATA_CHECK            : in  DISK_DATA_CHECK_TYPE;       --optional
   SHARE                 : in  BOOLEAN;                    --optional
   GROUP                 : in  BOOLEAN;                    --optional
   SYSTEM                : in  BOOLEAN;                    --optional
   VERIFIED              : in  BOOLEAN;                    --optional
   BADLIST               : in  DISK_BADLIST_DESCR_TYPE;
   STATUS                : out COND_VALUE_TYPE);           --optional
```

Arguments

DEVICE_NAME

This argument supplies a varying string of up to 31 characters, giving the device specification of the disk drive; for example, 'DQA1' for drive

1 of controller DQA. The node must be specified explicitly for a drive on another node.

VOLUME_NAME
This argument supplies a varying string of up to 12 characters, giving the volume label for the disk.

DEFAULT_EXTENSION
This argument supplies a value in the range of 0 to 65,535 (type SYSTEM.UNSIGNED_WORD) giving 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 an update. The default is 5 blocks.

USER_NAME
This argument supplies a string of up to 20 characters, giving a user name to be recorded on the volume. If it is omitted, the default is VAXELN.

OWNER
This argument supplies an integer identifying the volume owner UIC. The default is 16#00010001#.

VOLUME_PROTECTION
This argument supplies a value (see the example at the end of this section) that specifies the VAX/VMS protection for the volume. If it is not specified, users in all categories (system, owner, group, and world) have RWED (read, write, execute, and delete) access. When you specify protection for an entire disk volume, "execute" privilege implies create privilege. Note that the GROUP, SHARE, and SYSTEM arguments can also be used to specify volume protection.

FILE_PROTECTION
This argument supplies a value that specifies the default protection code for all files on the volume. If it is omitted, the system and owner have RWED access, the group has RE access, and the world has no access.

RECORD_PROTECTION
This argument supplies a value that supplies the write-protection code for records. If it is omitted, the system and owner have RWED access, the group has read access, and the world has no access.

ACCESSED_DIRECTORIES
This argument supplies a value in the range of 0 to 255 designating the number of directories that can be stored by the File Service by default. The default is 3.

MAXIMUM_FILES
This argument is an integer that supplies the maximum number of files that can exist on a disk. The default is calculated by the procedure based on the size of the disk.

USER_DIRECTORIES
This argument supplies a value in the range of 16 to 16000 specifying the number of entries that are preallocated for user directories. The default is 16.

FILE_HEADERS
This argument is an integer that supplies the number of file headers allocated initially for the index file (the file for the volume's file structure). The MAXIMUM value is the same as the MAXIMUM_FILES value. The default is 16.

WINDOWS
This argument is a value in the range of 7 to 80 that supplies the number of mapping pointers to be allocated for file windows. When a file is opened, the mapping pointers are used to describe the logical segments of the file for access. The default is 7.

CLUSTER_SIZE
This argument supplies a value in the range of 1 to 1/100 the size of the volume giving the cluster size. The default is 1 for volumes with less than or equal to 50000 blocks; otherwise the default is 3. The cluster size is the minimum allocation unit for the volume.

INDEX_POSITION
This argument supplies a value that specifies the position of the index file. Possible values are POSITION_BEGINNING, POSITION_MIDDLE, and POSITION_END. The default is POSITION_MIDDLE.

DATA_CHECK
This argument enables or disables data checking on read or write operations. Possible values are READ (check following all read operations), WRITE (check following all write operations), and NOCHECK. The default is NOCHECK.

SHARE

This argument designates whether the volume is shareable. The default is TRUE, implying that users in all categories have read, write, execute, and delete privileges. If the argument is FALSE, the default protection is no access for group and world, RWED access for system and owner.

GROUP

This argument designates that the disk volume is a group volume. If it is TRUE, the owner UIC defaults to the group number as specified in the OWNER argument, and the member number defaults to 0. The default is FALSE. If group is TRUE and share is FALSE, the volume protection is RWED for the group, owner, and system. However, if group and share are both TRUE, the volume protection is RWED for all user categories.

SYSTEM

This argument designates that the volume is a "system volume." In this case, the default protection is RWED access for all users of the system. Only VAX/VMS users with system UICs can create directories on system volumes. The default is TRUE.

VERIFIED

This argument supplies a Boolean value that designates whether the volume has information about where bad blocks are located. The default is TRUE. FALSE means that the procedure should ignore information already on the disk about bad blocks.

BAD_LIST

This 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. For physical block numbers, PBN_FORMAT must be TRUE; for logical block numbers, it must be FALSE. The argument is required, although a null list can be specified. To specify a null list of bad blocks, allocate a zero-extent array.

STATUS

This argument receives the completion status of the INIT_VOLUME procedure.

Notes:

VAXELN processes that create directories and files on the disk are not restricted by the privileges implied by such parameters as VOLUME_PROTECTION. These parameters are provided for compatibility with VAX/VMS systems.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_DISK_DEVMOUNT	The device is already mounted.
ELN_DISK_NOTFILEDEV	The device is not file structured.
ELN_DISK_xxx	Any other error returned by disk service.
KER_NO_SUCH_DEVICE	The device does not exist.

JOB_PORT

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

Procedure Declaration

```
procedure JOB_PORT (
   STATUS : out COND_VALUE_TYPE;     --optional
   PORT   : out PORT_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the JOB_PORT procedure.

PORT
This argument receives a PORT value identifying the caller's job port.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

KWV_INITIALIZE

The KWV_INITIALIZE procedure readies a real-time clock for input and creates all needed data structures. This procedure must be called at least once for each KWV device used. (The only reason to call this procedure more than once for a single device is to change the value of a parameter or to stop a device that is running in mode 0 or mode 1.)

Procedure Declaration

```
procedure KWV_INITIALIZE (
   DEVICE_NAME    : in  DEVICE_NAME_TYPE;
   IDENTIFIER     : out KWV_IDENTIFIER_TYPE;
   MODE           : in  KWV_MODE_TYPE;
   CLOCK_RATE     : in  KWV_CLOCK_RATE_TYPE;
   MAXIMUM_VALUES : in  INTEGER;              --optional
   RE_INITIALIZE  : in  BOOLEAN;              --optional
   USE_POLLING    : in  BOOLEAN;              --optional
   STATUS         : out COND_VALUE_TYPE);     --optional
```

Arguments

DEVICE_NAME
This argument supplies a varying string of up to 30 characters giving the name of the device to be initialized. This name must match the name established with the System Builder Utility.

IDENTIFIER
This argument receives a longword identifier to be used to identify this device in subsequent calls to the KWV_INITIALIZE, KWV_READ, and KWV_WRITE procedures.

MODE
This argument is an expression of type KWV_MODE that determines the mode in which the device is to be operated.

- KWV_MODE_ZERO is a single interval mode. In this mode, action is initiated by a call to the KWV_WRITE procedure. The clock is started by either a Schmitt Trigger #2 signal or the call itself. The clock stops after counting the number of ticks specified in the call to the

KWV_WRITE procedure. At this time (clock overflow), it interrupts the processor, if that is enabled, and asserts the clock-overflow signal line. Note that interrupting the processor on overflow is not supported by this driver.

- KWV_MODE_ONE is a repeated interval mode. This mode is identical to single interval mode, except that when clock overflow is reached, the clock is repeatedly restarted to run for the same interval. Therefore, the device produces repeated signals on the clock-overflow line and repeated processor interrupts, if that is enabled. Note that interrupting the processor on overflow is not supported by this driver.
- KWV_MODE_TWO is an external event or program timing mode. In this mode, used for timing an external event, action is initiated by a call to the KWV_READ procedure. The clock's counter is set to zero and is started by either the KWV_READ call or a Schmitt Trigger #2 signal. The clock continues to run, and its COUNTER value is read each time there is an external signal to Schmitt Trigger #2, until the desired number of values has been read.

  This mode may also be used to time a section of code; that is, the clock may be started and stopped from program control. In this case, it is started by a call to the KWV_WRITE procedure. A subsequent call to the KWV_READ procedure stops the clock and reads a single value from its counter, which represents the elapsed time since the write. If a device is used in this way, switches 3 and 4 on dip switch sw3 should be in the "off" position; otherwise, any external signals to Schmitt Trigger #2 may result in incorrect operation.
- KWV_MODE_THREE is an external event timing from zero base mode. This mode is identical to KWV_MODE_TWO, except that the counter is reset to zero each time its contents are read.

CLOCK_RATE

This argument is an expression of type KWV_CLOCK_RATE that supplies the clock frequency to be used. This can be a set crystal-controlled frequency, the AC line frequency, or Schmitt Trigger #1 input.

MAXIMUM_VALUES

This argument supplies the maximum number of data values that can be read from the specified device in a single call to the KWV_READ procedure. This argument is only significant for KWV_MODE_TWO and KWV_MODE_TWO. If this argument is not specified, a default of one is assumed.

RE_INITIALIZE

This argument is a Boolean expression. TRUE means that the device has been initialized previously, in which case the IDENTIFIER argument is ignored and is used to identify the device. No new data structures or objects are created unless the MAXIMUM_VALUES argument is greater than the previous value for this device. The default value is FALSE.

USE_POLLING

This argument is a Boolean expression. TRUE causes the device to be driven by polling rather than interrupts; FALSE means interrupts will be used to gather data. Polling is always done at device IPL (four for this device). The default value is FALSE. This argument is only significant for a device operating in KWV_MODE_TWO or KWV_MODE_THREE when the KWV_READ procedure is called to gather data (that is, not when the KWV_READ procedure is called following a call to the KWV_WRITE procedure).

STATUS

This argument receives the completion status of the KWV_INITIALIZE procedure. The only possible value is 1, which indicates that the procedure completed successfully.

Status Value

KER_SUCCESS	The procedure completed successfully.

KWV_READ

The KWV_READ procedure reads TIME values from a specified device and stores them in a data array. The procedure may only be called for a device that has been initialized to operate in KWV_MODE_TWO or KWV_MODE_THREE.

There are two possible cases in which the KWV_READ procedure would be called:

- The device is not already running. In this case, the call to the KWV_READ procedure either starts the device or sets the device so that a signal from Schmitt Trigger #2 will start it. Subsequently, the specified number of data is gathered (each representing the occurrence of a Schmitt Trigger #2 signal), and the clock is stopped.
- The device is already running, having been started by a call to the KWV_WRITE procedure. In this case, the call to the KWV_READ procedure stops the clock, then reads and returns the clock's COUNTER value.

Procedure Declaration

```
procedure KWV_READ (
   IDENTIFIER     : in  KWV_IDENTIFIER_TYPE;
   VALUE_COUNT    : in  INTEGER;
   DATA_ARRAY_PTR : out ADDRESS;
   ST2_GO_ENABLE  : in  BOOLEAN;              --optional
   STATUS         : out COND_VALUE_TYPE);     --optional
```

Arguments

IDENTIFIER
This argument supplies a value that identifies the device to be read; this value is the one returned in the IDENTIFIER argument after a call to the KWV_INITIALIZE procedure.

VALUE_COUNT
This argument supplies the number of values to be read.

DATA_ARRAY_PTR
This argument receives the address of an array containing data from the device. Output data is a signed 16-bit integer giving a count of ticks.

ST2_GO_ENABLE
This argument is a Boolean expression. TRUE causes the clock to begin counting upon receipt of a Schmitt Trigger #2 signal. FALSE causes the call to the KWV_READ procedure itself to start the counter. The default value is FALSE. This argument is ignored if the clock is already running.

STATUS
This argument receives the completion status of the KWV_READ procedure.

Status Value

ELN_KWV_DATA_OVERRUN	This value indicates that a Schmitt Trigger #2 event occurred before the driver had finished processing the previous one.

KWV_WRITE

The KWV_WRITE procedure performs differently depending on which mode the device is operating in:

- For devices initialized to operate in mode 0 or mode 1, the KWV_WRITE procedure causes the device to generate the clock-overflow signal when the specified number of ticks has occurred. Additionally, if the device was initialized to operate in mode 1, clock-overflow signals will be generated repeatedly after each interval containing the specified number of ticks. The clock can be stopped by calling the KWV_INITIALIZE procedure to reinitialize it.
- For devices initialized to operate in KWV_MODE_TWO or KWV_MODE_THREE, the KWV_WRITE procedure causes the device to begin counting from zero, or wait for an ST2 signal to do so. It is then expected that sometime a call to the KWV_READ procedure will be made, which reads the current elapsed time.

Procedure Declaration

```
procedure KWV_WRITE (
   IDENTIFIER    : in  KWV_IDENTIFIER_TYPE;
   ST2_GO_ENABLE : in  BOOLEAN;                --optional
   TICK_COUNT    : in  KWV_COUNTER_TYPE;       --optional
   STATUS        : out COND_VALUE_TYPE);       --optional
```

Arguments

IDENTIFIER

This argument supplies a value that identifies the device to be written to; this value is the one returned in the IDENTIFIER argument after a call to the KWV_INITIALIZE procedure.

ST2_GO_ENABLE

This argument is a Boolean expression. TRUE causes the clock to begin counting upon receipt of a Schmitt Trigger #2 signal. FALSE causes the call to the KWV_WRITE procedure itself to start the counter. The default value is FALSE.

TICK_COUNT
This argument is an interval in clock ticks after which a clock-overflow signal is asserted. This argument has no significance if the device was initialized to operate in mode 0 or 1.

STATUS
This argument receives the completion status of the KWV_WRITE procedure. The only possible value is 1, which indicates that the procedure completed successfully.

Status Value

KER_SUCCESS	The procedure completed successfully.

LOAD_PROGRAM

The LOAD_PROGRAM procedure loads a specified image file into a currently running VAXELN system. After the image file is loaded, the CREATE_JOB procedure is used to start the program running.

Procedure Declaration

```
procedure LOAD_PROGRAM (
   FILE_NAME          : in  FILE_NAME_TYPE;
   PROGRAM_NAME       : in  PROGRAM_NAME_TYPE;
   KERNEL_MODE        : in  BOOLEAN;                 --optional
   START_WITH_DEBUG   : in  BOOLEAN;                 --optional
   POWER_RECOVERY     : in  BOOLEAN;                 --optional
   KERNEL_STACK_SIZE  : in  INTEGER;                 --optional
   USER_STACK_SIZE    : in  INTEGER;                 --optional
   MESSAGE_LIMIT      : in  INTEGER;                 --optional
   JOB_PRIORITY       : in  JOB_PRIORITY_TYPE;       --optional
   PROCESS_PRIORITY   : in  PROCESS_PRIORITY_TYPE;   --optional
   STATUS             : out COND_VALUE_TYPE);        --optional
```

Arguments

FILE_NAME
This argument supplies a varying string giving the name of the image file to be loaded into the system. The file is opened in the context of the caller, so the file name must be provided in sufficient detail to correctly identify the file. The file can reside on the system or on a remote node.

PROGRAM_NAME
This argument supplies a varying string giving the name by which the program will be known for the CREATE_JOB call. If the argument is specified as a null string, the image name supplied by the linker is used.

KERNEL_MODE
This argument is a Boolean value specifying in which mode the program is to run. TRUE means kernal mode; FALSE (the default) means user mode.

START_WITH_DEBUG
This argument specifies whether the debugging process is to get control of the program when it is started. TRUE means the process is to get control; FALSE (the default) means the process is not to get control.

POWER_RECOVERY
This argument specifies whether the job running the specified program is to be given the power recovery exception if the power fails on the system. TRUE means the job is to be given the power recovery exception; FALSE (the default) means the job is not to be given the power recovery exception.

KERNEL_STACK_SIZE
This argument is an INTEGER value that supplies the size, in pages, of the kernel mode stack for jobs running this program. User mode programs require at least one page (the default) of kernel stack.

USER_STACK_SIZE
This argument is an INTEGER value that supplies the initial size, in pages, of the user mode stack for jobs running this program. Programs require at least one page (the default) of user stack. This parameter is ignored for kernel mode programs.

MESSAGE_LIMT
This argument is an INTEGER value that specifies the maximum number of messages the job port can contain; the default is 0.

JOB_PRIORITY
This argument is an integer from 0 to 31 that specifies the starting job priority for this program; the default is 16.

PROCESS_PRIORITY
This argument is an integer from 0 to 15 that specifies the starting process priority for this program; the default is 8.

STATUS
This argument receives the completion status of the LOAD_PROGRAM procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_xxx	Any error status returned by the file service.
KER_BAD_VALUE	One of the parameters is invalid or the program has already been loaded.
KER_BAD_IMAGE_FORMAT	The image being loaded contains an unsupported image format.
KER_NO_SUCH_IMAGE	A required shareable image was not found.
KER_NO_SYSTEM_PAGE	There is not enough system page table entries available.

LOAD_UNIBUS_MAP

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

Procedure Declaration

```
procedure LOAD_UNIBUS_MAP (
   MAP_REGISTER : in  ADDRESS;
   BUFFER       : in  IO_BUFFER_TYPE;
   BUFFER_SIZE  : in  INTEGER;
   SPT_ADDRESS  : in  ADDRESS;          --optional
   DATA_PATH    : in  INTEGER);         --optional
```

Arguments

MAP_REGISTER
This argument is a pointer to the first UNIBUS map register allocated by the ALLOCATE_MAP procedure.

BUFFER
This argument represents the I-O buffer.

BUFFER_SIZE
This argument supplies the buffer size.

SPT_ADDRESS
This argument is a pointer to the system page table (SPT). If this argument is not supplied, a device communication region (or any system space buffer) cannot be mapped.

DATA_PATH
This argument supplies a UNIBUS datapath to be used for the transfer. If the argument is not supplied, DATA_PATH 0, the direct datapath, is used.

Notes:

The LOAD_UNIBUS_MAP procedure assumes that sufficient map registers have been allocated by the calling program using the ALLOCATE_MAP procedure (the UNIBUS_MAP procedure allocates them for the caller). The LOAD_UNIBUS_MAP procedure 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."

Status Values

None

LOCK_MUTEX

The LOCK_MUTEX procedure locks a mutex (mutual exclusion semaphore).

Procedure Declaration

```
procedure LOCK_MUTEX (
   STATUS : out COND_VALUE_TYPE;     --optional
   MUTEX  : in out MUTEX_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the LOCK_MUTEX procedure.

MUTEX
This argument is the mutex to be locked. A mutex must be initialized by the CREATE_MUTEX call before it can be locked.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The MUTEX argument specifies an object that is not a mutex.
KER_BAD_VALUE	The MUTEX argument specifies a mutex that has been deleted.

MEMORY_SIZE

The MEMORY_SIZE kernel procedure scans the kernel memory database and returns, in units of 512-byte pages, the initial main memory, the current free memory, and the size of the largest, physically contiguous, block of free memory. While the MEMORY_SIZE procedure performs the memory scan, all other kernel operations are stopped.

Procedure Declaration

```
procedure MEMORY_SIZE (
   STATUS       : out COND_VALUE_TYPE;     --optional
   TOTAL_SIZE   : out INTEGER;
   FREE_SIZE    : out INTEGER;
   LARGEST_SIZE : out INTEGER);
```

Arguments

STATUS
This argument receives the completion status of the MEMORY_SIZE procedure.

TOTAL_SIZE
This argument receives the size, in units of 512-byte pages, of the initial main memory.

FREE_SIZE
This argument receives the size, in units of 512-byte pages, of the current free memory.

LARGEST_SIZE
This argument receives the size, in units of 512-byte pages, of the largest, physically contiguous, block of free memory.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

MOUNT_TAPE_VOLUME

The MOUNT_TAPE_VOLUME procedure mounts a File Service tape on the specified tape drive for use as a file-structured volume that conforms to ANSI standard X3.27-1978. The procedure requires that the device and its driver (and the tape File Service) be present in the same system from which it is called. The procedure does not return until the tape is completely mounted.

Procedure Declaration

```
procedure MOUNT_TAPE_VOLUME (
   DEVICE_NAME : in  DEVICE_NAME_TYPE;
   VOLUME_NAME : in  TAPE_VOLUME_NAME_TYPE;      --optional
   BLOCK_SIZE  : in  INTEGER;                    --optional
   STATUS      : out COND_VALUE_TYPE);           --optional
```

Arguments

DEVICE_NAME
This argument is a varying string of up to 31 characters, giving 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.

VOLUME_NAME
This argument supplies the volume label for the tape as a varying string of 1 to 6 characters.

BLOCK_SIZE
This argument supplies a value that determines the number of bytes in each block of a newly created file. The default is 2048.

STATUS
This argument receives the completion status of the MOUNT_TAPE_VOLUME procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_TAPE_DIFLBLMNT	A volume with a different label was mounted (informational message).
ELN_TAPE_VOLNAMMSK	This volume's name is masked by another volume (informational message).
ELN_TAPE_DEVINUSE	The device is in use by another process.
ELN_TAPE_DEVMOUNT	The device is already mounted.
ELN_TAPE_DEVERROR	The device error occurred.
ELN_TUTL_INVCHRVOL	There is an invalid character in the volume label.
ELN_TUTL_BLKSIZ	The invalid block size is specified.
ELN_FILE_DEVNOTMNT	There is no volume mounted on the device.
ELN_FILE_xxx	Any other error returned by the file service.
KER_NO_SUCH_DEVICE	The device does not exist.

MOUNT_VOLUME

The MOUNT_VOLUME procedure mounts a File Service disk for use as a file-structured volume. The procedure requires that the device and its driver (linked to the File Service) be present in the same system from which it is called. The procedure does not return until the disk is completely mounted.

Procedure Declaration

```
procedure MOUNT_VOLUME (
   DEVICE_NAME : in  DEVICE_NAME_TYPE;
   VOLUME_NAME : in  DISK_VOLUME_NAME_TYPE;     --optional
   STATUS      : out COND_VALUE_TYPE);          --optional
```

Arguments

DEVICE_NAME
This argument is a varying string of up to 31 characters naming the disk drive on which the volume is to be mounted; for example, 'DQA1' for drive 1 on controller DQA.

VOLUME_NAME
This argument is a varying string of up to 12 characters, supplying the volume label. If it is omitted, the procedure simply mounts whatever volume is loaded in the indicated drive.

STATUS
This argument receives the completion status of the MOUNT_VOLUME procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_DISK_DEVMOUNT	The device is already mounted.
ELN_DISK_INVCHRVOL	There is an invalid character in the volume label.
ELN_DISK_xxx	Any other error returned by the disk service.
ELN_FILE_DEVINUSE	The device is in use by another process.
ELN_FILE_DEVNOTMNT	There is no volume mounted on the device.
ELN_FILE_INCVOLLABEL	The volume label is not correct, however the volume is mounted anyway.
ELN_FILE_VOLALRMNT	The volume is already mounted.
ELN_FILE_VOLIMPDSM	The volume was improperly dismounted; please rebuild on the VAX/VMS system.
ELN_FILE_MLTVOLLABEL	A volume with this name has already been mounted.
ELN_FILE_xxx	Any other error returned by the file service.
KER_NO_SUCH_DEVICE	The device does not exist.

PHYSICAL_ADDRESS

The PHYSICAL_ADDRESS function returns the physical address of a data item. A program that calls the PHYSICAL_ADDRESS function must be linked with the kernel symbol table (ELN$:KERNEL.STB).

Procedure Declaration

```
function PHYSICAL_ADDRESS (
   POINTER : in ADDRESS) return PHYSICAL_ADDRESS_TYPE;
```

Arguments

POINTER

This argument supplies the virtual address of a data item. The returned value is the physical address of the data item.

PROGRAM_ARGUMENT

The PROGRAM_ARGUMENT function returns the character string passed as a program argument to the current job.

Procedure Declaration

```
function PROGRAM_ARGUMENT (
   POSITION : in INTEGER) return PROGRAM_ARGUMENT_TYPE;
```

Arguments

POSITION

This argument is an integer expression that gives the position in the argument list (in CREATE_JOB or the System Builder's program description). The first position is 1.

The returned value is the character string passed as the argument in, for example, a CREATE_JOB call.

If there is no argument or if POSITION exceeds the number of program arguments, the returned value is the null string.

PROGRAM_ARGUMENT_COUNT

The PROGRAM_ARGUMENT_COUNT function returns an integer indicating the number of arguments passed to the program.

Procedure Declaration

```
function PROGRAM_ARGUMENT_COUNT return INTEGER;
```

The returned value is an INTEGER value giving the number of arguments passed.

PROTECT_FILE

The PROTECT_FILE procedure changes the file ownership UIC or protection code or both for a specified disk file. This procedure is invalid for tape volumes.

Procedure Declaration

```
procedure PROTECT_FILE (
   FILE_NAME            : in  FILE_NAME_TYPE;
   OWNER                : in  UIC_LONGWORD_TYPE;       --optional
   PROTECTION           : in  FILE_PROTECTION_TYPE;    --optional
   STATUS               : out COND_VALUE_TYPE;         --optional
   RESULTANT_FILE_NAME  : out FILE_NAME_TYPE);         --optional
```

Arguments

FILE_NAME
This argument is a varying string giving the file specification. Wildcard characters are not permitted.

OWNER
This argument supplies the ownership UIC of the file. If this argument is not specified or is specified as zero, the file ownership is not changed.

PROTECTION
This argument supplies a protection code for the file. The protection code is a 16-bit word that is composed of four 4-bit fields. Each field represents a category of users: system, owner, group, and world. Each of the four fields consists of four 1-bit indicators that specify the access denied each category. If this argument is not specified, the protection code is not changed.

STATUS
This argument receives the completion status of the PROTECT_FILE procedure. An exception is raised if the procedure does not succeed and this argument is omitted.

RESULTANT_FILE_NAME
This argument is a varying string that receives the resultant file name of the file.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_xxx	Any error status returned by the file service.

RAISE_PROCESS_EXCEPTION

The RAISE_PROCESS_EXCEPTION procedure raises the asynchronous exception KER_PROCESS_ATTENTION in the specified process.

Procedure Declaration

```
procedure RAISE_PROCESS_EXCEPTION (
   STATUS  : out COND_VALUE_TYPE;    --optional
   PROCESS : in  PROCESS_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the RAISE_PROCESS_EXCEPTION procedure.

PROCESS
This argument specifies the process in which the exception is to be raised.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The PROCESS argument is not of type PROCESS.
KER_BAD_VALUE	The PROCESS argument is invalid or refers to a deleted process.

RECEIVE

The RECEIVE procedure removes a message from the designated message port and maps the text of the message into the receiver job's virtual address space.

Procedure Declaration

```
procedure RECEIVE (
   STATUS           : out COND_VALUE_TYPE;       --optional
   MESSAGE          : out MESSAGE_TYPE;
   DATA_ADDRESS     : out ADDRESS;
   MESSAGE_SIZE     : out UNSIGNED_LONGWORD;
   SOURCE_PORT      : in  PORT_TYPE;
   DESTINATION_PORT : out PORT_TYPE;             --optional
   REPLY_PORT       : out PORT_TYPE);            --optional
```

Arguments

STATUS
This argument receives the completion status of the RECEIVE procedure.

MESSAGE
This argument receives the MESSAGE value identifying the next message, if there is one in the port.

DATA_ADDRESS
This argument receives an ADDRESS value that is the location of the text of the received message. The value is valid only in the current job and becomes invalid if the message is sent or deleted.

MESSAGE_SIZE
This argument is an integer that receives the size in bytes of the text of the message.

SOURCE_PORT
This argument supplies the value of the port from which to retrieve the message.

SIZE

This argument is an integer that receives the size in bytes of the text of the message.

DESTINATION_PORT

This argument receives the value of the destination port. Normally, this is the same value supplied by the sender for the receiver's port. It is available, and returns a different value only for the internal interface between the kernel and the Network Service.

REPLY_PORT

This argument receives the value of the reply port. Note that this value is not set properly by the RECEIVE procedure if the port is connected in a circuit.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_EXPEDITED	The procedure completed successfully, and the received message is an expedited message.
KER_CONNECT_PENDING	A CONNECT_CIRCUIT procedure is pending, and the port cannot be used for another purpose until the connection has completed.
KER_DISCONNECT	The circuit was disconnected by the partner process.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_MESSAGE	No unreceived messages are currently in the port.
KER_NO_OBJECT	No free job object table entries are currently available; there are a maximum of 1024 object table entries per job.

KER_NO_SUCH_PORT	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 RECEIVE procedure.
KER_NO_VIRTUAL	No free virtual address space (to map the message text) is currently available for the job; the size of the job's virtual address space can be set using the System Builder Utility.

RENAME_FILE

The RENAME_FILE procedure renames a disk file. To rename several related files, use the DIRECTORY_LIST procedure to find them and the RENAME_FILE procedure to rename each one.

Procedure Declaration

```
procedure RENAME_FILE (
   OLD_FILE_NAME             : in  FILE_NAME_TYPE;
   NEW_FILE_NAME             : in  FILE_NAME_TYPE;
   STATUS                    : out COND_VALUE_TYPE;    --optional
   RESULTANT_OLD_FILE_NAME   : out FILE_NAME_TYPE;     --optional
   RESULTANT_NEW_FILE_NAME   : out FILE_NAME_TYPE);    --optional
```

Arguments

OLD_FILE_NAME

This argument is a varying string giving the current file specification. No wildcard characters are permitted.

NEW_FILE_NAME

This argument is a varying string giving the new file specification. 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 either supply the same volume name or no volume name. Any parts of the current specification that are not supplied in this argument are obtained from the OLD_FILE_NAME procedure.

STATUS

This argument receives the completion status of the RENAME_FILE procedure.

RESULTANT_OLD_FILE_NAME

This argument is a varying string that receives the resultant file name of the old file.

RESULTANT_NEW_FILE_NAME
This argument is a varying string that receives the resultant file name of the new file.

Status Values

KER_SUCCESS	The procedure completed successfully.
ELN_xxx	Any error status returned by the file service.

RESUME

The RESUME procedure resumes 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, as if the WAIT_ANY or WAIT_ALL procedure were called again. Any asynchronous exceptions that occurred during the suspension are raised before the procedure is resumed, including the exception KER_QUIT_SIGNAL that results from signaling the process itself.

Procedure Declaration

```
procedure RESUME (
   STATUS  : out COND_VALUE_TYPE;     --optional
   PROCESS : in  PROCESS_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the RESUME procedure.

PROCESS
This argument supplies a value of type PROCESS that identifies the process to be resumed.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_STATE	A process specified to resume is not suspended.
KER_BAD_TYPE	The first argument is not of type PROCESS.
KER_BAD_VALUE	The PROCESS argument is invalid or identifies a process that no longer exists.

SEND

The SEND procedure removes a message's text variable from the sender's address space and then places the MESSAGE object that describes the text in the destination's message port.

Procedure Declaration

```
procedure SEND (
   STATUS           : out COND_VALUE_TYPE;
   MESSAGE          : in  MESSAGE_TYPE;
   MESSAGE_SIZE     : in  UNSIGNED_LONGWORD;      --optional
   DESTINATION_PORT : in  PORT_TYPE;
   REPLY_PORT       : in  PORT_TYPE;              --optional
   EXPEDITE         : in  BOOLEAN);               --optional
```

Arguments

STATUS
This argument receives the completion status of the SEND procedure.

MESSAGE
This argument supplies the MESSAGE value identifying the message to send. After the operation, any pointers to the message's text variable are no longer valid.

MESSAGE_SIZE
This argument is an integer that supplies the length in bytes of the message text to be sent; if it is omitted, the size of the originally created text variable is the default. If size is specified, its value must be equal to or less than the text variable's size.

DESTINATION_PORT
This argument supplies the PORT value identifying the destination port; if the message is being sent through a circuit, this port is the sender's half, and the message arrives at the receiver's half.

REPLY_PORT
This argument identifies the reply port. If it is not specified, the kernel supplies the value of the sender's job port.

EXPEDITE
This argument supplies a Boolean value stating whether to expedite the message. The default is FALSE. An expedited message bypasses the normal flow-control mechanism and can be received even if the receiving port already has its maximum number of messages. The message is received by the port before any normal data messages. The size of an expedited message must not exceed 16 bytes.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_MESSAGE_SIZE	The message text variable is too large to be sent to the destination port; this can occur for the following reasons: • The message is being sent to a remote port not connected in a circuit. The maximum message text variable size that can be sent as a datagram to a remote port is the System Builder's Network Segment Size minus 32. The default segment size is 576, so the maximum size remote datagram in the default case is 544 bytes. • The message is being expedited. The maximum text variable size that can be sent as an expedited message is 16 bytes.
KER_BAD_TYPE	The argument is not of type MESSAGE.
KER_BAD_VALUE	The MESSAGE or SIZE argument is invalid or the MESSAGE argument refers to a deleted message.
KER_CONNECT_PENDING	A CONNECT_CIRCUIT procedure is pending, and the port cannot be used for another purpose until the connection has completed.

KER_COUNT_OVERFLOW	The destination port is full (with circuits, raised if the FULL_ERROR parameter was TRUE, in the ACCEPT_CIRCUIT or CONNECT_CIRCUIT procedures).
KER_DISCONNECT	The circuit was disconnected by the partner process.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.
KER_NO_SUCH_PORT	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 SEND procedure with circuits.

SET_JOB_PRIORITY

The SET_JOB_PRIORITY procedure sets the scheduling priority of the current job. The initial priority for a job can be set by the System Builder Utility as part of a program description; the default is 16.

Procedure Declaration

```
procedure SET_JOB_PRIORITY (
   STATUS   : out COND_VALUE_TYPE;        --optional
   PRIORITY : in  JOB_PRIORITY_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SET_JOB_PRIORITY procedure.

PRIORITY
This argument is an integer in the range of 0 to 31 that supplies the new priority. Priority 0 is most urgent.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The PRIORITY argument is out of range.

SET_PROCESS_PRIORITY

The SET_PROCESS_PRIORITY procedure sets the scheduling priority of the specified process. The initial priority for the processes in a job can be set by the System Builder Utility as part of a program description; the default is 8. Note that Ada task priorities are computed as 15-P, where P is the VAXELN process priority.

Procedure Declaration

```
procedure SET_PROCESS_PRIORITY (
   STATUS   : out COND_VALUE_TYPE;          --optional
   PROCESS  : in  PROCESS_TYPE;
   PRIORITY : in  PROCESS_PRIORITY_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SET_PROCESS_PRIORITY procedure.

PROCESS
This argument supplies the PROCESS value identifying the process whose priority is to be changed.

PRIORITY
This argument supplies the new priority as an integer in the range of 0 to 15. Priority 0 is the most urgent. See Section 7.3 for more information on the ordering of priorities.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The first argument is not of type PROCESS.
KER_BAD_VALUE	Either the PROCESS argument is invalid or refers to a deleted process, or the PRIORITY argument is out of range.

SET_PROTECTION

The SET_PROTECTION procedure protects a specified region of virtual memory.

Procedure Declaration

```
procedure SET_PROTECTION (
   STATUS : out COND_VALUE_TYPE;          --optional
   SIZE   : in  UNSIGNED_LONGWORD;
   BASE   : in  ADDRESS;
   CODE   : in  MEMORY_PROTECTION_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SET_PROTECTION procedure.

SIZE
This argument supplies the number of bytes of memory to be protected.

BASE
This argument supplies the virtual base address of the memory to be protected.

CODE
This argument specifies the desired access: READ_ONLY, READ_WRITE, or NO_ACCESS.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The BASE argument is not in the calling job's address space.
KER_NO_MEMORY	No free pages of physical memory are currently available.

SET_TIME

The SET_TIME procedure sets a new system time.

Procedure Declaration

```
procedure SET_TIME (
   STATUS : out COND_VALUE_TYPE;    --optional
   TIME   : in  DATE_TIME_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SET_TIME procedure.

TIME
This argument specifies an absolute system time.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The TIME value is invalid.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

SET_USER

The SET_USER procedure sets the user identity of the current process.

Procedure Declaration

```
procedure SET_USER (
   STATUS    : out COND_VALUE_TYPE;     --optional
   USER_NAME : in  AUTH_STRING_TYPE;
   UIC       : in  UIC_LONGWORD_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SET_USER procedure.

USER_NAME
This argument supplies a string of up to 20 characters giving the user name to be associated with the process.

UIC
This argument is an INTEGER value that supplies the UIC to be associated with the process.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

SIGNAL_AREA

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

Procedure Declaration

```
procedure SIGNAL_AREA
  STATUS : out COND_VALUE_TYPE;     --optional
  AREA   : in AREA_NAME_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SIGNAL_AREA procedure. The default is unit 0.

AREA
This argument selects the area to signal.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The VALUE argument identifies an object that cannot be signaled.
KER_BAD_VALUE	The VALUE argument is invalid or refers to a deleted object.

SIGNAL_DEVICE

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.

Procedure Declaration

```
procedure SIGNAL_DEVICE (
   STATUS        : out COND_VALUE_TYPE;          --optional
   DEVICE_NUMBER : in  DEVICE_NUMBER_TYPE);      --optional
```

Arguments

STATUS
This argument receives the completion status of the SIGNAL_DEVICE procedure.

DEVICE_NUMBER
This argument supplies an integer in the range of 0 to 15, identifying the element in a DEVICE array to be signaled.

Notes:

No exceptions are raised by the procedure, even if status is not requested and an error occurs.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The INTEGER argument is out of range.

SIGNAL_EVENT

The SIGNAL_EVENT procedure sets the state of an event to SIGNALED and continues all waiting processes whose wait conditions can be satisfied.

Procedure Declaration

```
procedure SIGNAL_EVENT (
   STATUS : out COND_VALUE_TYPE;      --optional
   EVENT  : in  EVENT_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SIGNAL_EVENT procedure.

EVENT
This argument selects the event to signal.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The VALUE argument identifies an object that cannot be signaled.
KER_BAD_VALUE	The VALUE argument is invalid or refers to a deleted object.

SIGNAL_PROCESS

The SIGNAL_PROCESS procedure signals a process to quit.

The process must establish a handler for the exception KER_QUIT_SIGNAL. If it does not handle the exception, the process is forced to exit.

Procedure Declaration

```
procedure SIGNAL_PROCESS (
   STATUS  : out COND_VALUE_TYPE;     --optional
   PROCESS : in  PROCESS_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SIGNAL_PROCESS procedure.

PROCESS
This argument identifies the process you are signalling to quit.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The VALUE argument identifies an object that cannot be signaled.
KER_BAD_VALUE	The VALUE argument is invalid or refers to a deleted object.
KER_COUNT_OVERFLOW	The SIGNAL procedure was called for a semaphore already at its maximum count.

SIGNAL_SEMAPHORE

The SIGNAL_SEMAPHORE procedure increments and then tests the semaphore count.

Procedure Declaration

```
procedure SIGNAL_SEMAPHORE (
   STATUS    : out COND_VALUE_TYPE;     --optional
   SEMAPHORE : in  SEMAPHORE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SIGNAL_SEMAPHORE procedure.

SEMAPHORE
This argument identifies the semaphore to be signalled.

Notes:

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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The VALUE argument identifies an object that cannot be signaled.
KER_BAD_VALUE	The VALUE argument is invalid or refers to a deleted object.
KER_COUNT_OVERFLOW	The SIGNAL procedure was called for a semaphore already at its maximum count.

SUSPEND

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

Procedure Declaration

```
procedure SUSPEND (
   STATUS  : out COND_VALUE_TYPE;    --optional
   PROCESS : in  PROCESS_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the SUSPEND procedure.

PROCESS
This argument supplies a value identifying the process to be suspended.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The first argument is not of type PROCESS_TYPE.
KER_BAD_VALUE	The PROCESS argument is invalid or refers to a deleted process.

TRANSLATE_NAME

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.

Procedure Declaration

```
procedure TRANSLATE_NAME (
   STATUS    : out COND_VALUE_TYPE;      --optional
   PORT      : out PORT_TYPE;
   PORT_NAME : in  PORT_NAME_TYPE;
   SCOPE     : in  NAME_SCOPE_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the TRANSLATE_NAME procedure.

PORT
This argument receives the value of the associated message port.

PORT_NAME
This argument supplies the name of the port. Name strings are not case sensitive; uppercase and lowercase versions of the same name mean the same thing.

SCOPE
This argument specifies which name table (local or universal) is to be searched. Possible values are values of the predeclared enumerated type NAME_SCOPE_TYPE:

- LOCAL specifies that only the local name table is searched.
- UNIVERSAL specifies that only the universal name table is searched.
- BOTH specifies that the local name table is searched first, followed by the universal table. The search ends as soon as a match is found.

Status Values	KER_SUCCESS	The procedure completed successfully.
	KER_NO_ACCESS	An argument specified is not accessible to the calling program.
	KER_NO_SUCH_NAME	The translation for the specified name string cannot be found.

UNIBUS_MAP

The UNIBUS_MAP procedure is used in device driver programs to map memory buffers for direct memory access by UNIBUS devices. The specified buffer is mapped into the UNIBUS address space, and the address of the first register is returned.

Procedure Declaration

```
procedure UNIBUS_MAP (
   DEVICE         : in  DEVICE_TYPE;
   BUFFER         : in  IO_BUFFER_TYPE;
   BUFFER_SIZE    : in  INTEGER;
   UNIBUS_ADDRESS : out UNIBUS_ADDRESS_TYPE;
   DATA_PATH      : in  INTEGER);              --optional
```

Arguments

DEVICE
This argument identifies the device that will use the mapped memory.

BUFFER
This argument represents an I-O buffer.

BUFFER_SIZE
This argument is an integer supplying the buffer size.

UNIBUS_ADDRESS
This argument receives the 18-bit UNIBUS address of the mapped buffer.

DATA_PATH
This argument supplies an integer that specifies the UNIBUS adapter datapath to use. The default is 0, specifying the unbuffered datapath.

Notes:

The procedure allocates the correct number of map registers by calling the ALLOCATE_MAP procedure. 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 the UNIBUS_MAP procedure before use, a nonzero datapath number is assumed to be unused by any other device.

Status Values

None

UNIBUS_UNMAP

The UNIBUS_UNMAP procedure is used in device driver programs to unmap memory buffers previously mapped for direct memory access by a UNIBUS device.

Procedure Declaration

```
procedure UNIBUS_UNMAP (
   DEVICE         : in  DEVICE_TYPE;
   BUFFER         : in  IO_BUFFER_TYPE;
   BUFFER_SIZE    : in  INTEGER;
   UNIBUS_ADDRESS : in  UNIBUS_ADDRESS_TYPE);
```

Arguments

DEVICE
This argument identifies the UNIBUS device that was using the mapped memory.

BUFFER
This argument represents an I-O buffer.

BUFFER_SIZE
This argument is an integer supplying the buffer size.

UNIBUS_ADDRESS
This argument supplies the 18-bit UNIBUS address of the mapped buffer.

Notes:

The procedure deallocates the correct number of map registers by calling the FREE_MAP procedure.

Status Values

None

UNLOAD_PROGRAM

The UNLOAD_PROGRAM procedure unloads a specified program from a currently running VAXELN system.

Procedure Declaration

```
procedure UNLOAD_PROGRAM (
   PROGRAM_NAME : in  PROGRAM_NAME_TYPE;
   STATUS       : out COND_VALUE_TYPE);     --optional
```

Arguments

PROGRAM_NAME
This argument supplies a varying string identifying the program to be unloaded.

STATUS
This argument receives the completion status of the UNLOAD_PROGRAM procedure.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_VALUE	The program does not exist.

UNLOCK_MUTEX

The UNLOCK_MUTEX procedure unlocks a mutex.

Procedure Declaration

```
procedure UNLOCK_MUTEX (
   STATUS : out COND_VALUE_TYPE;      --optional
   MUTEX  : in out MUTEX_TYPE);
```

Arguments

STATUS
This argument receives the completion status of the UNLOCK_MUTEX procedure.

MUTEX
The argument is a variable of type MUTEX.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_TYPE	The MUTEX argument specifies an object that is not a mutex.
KER_BAD_VALUE	The MUTEX argument specifies a mutex that has been deleted.

WAIT_ALL or WAIT_ANY

The WAIT_ALL and WAIT_ANY procedures are used to make a process wait for one or more objects, including processes, ports, semaphores, events, devices, and areas. The WAIT_ANY procedure allows the invoking process to proceed if any of the wait conditions are satisfied; the procedure WAIT_ALL requires that all the conditions be satisfied simultaneously. The WAIT_ANY procedure identifies the object that satisfied the wait.

Procedure Declaration

```
procedure WAIT_ALL (
   STATUS : out COND_VALUE_TYPE;        --optional
   RESULT : out INTEGER;                --optional
   TIME   : in  DATE_TIME_TYPE;         --optional
   VALUE1 : in  SYSTEM_VALUE_TYPE;      --optional
   VALUE2 : in  SYSTEM_VALUE_TYPE;      --optional
   VALUE3 : in  SYSTEM_VALUE_TYPE;      --optional
   VALUE4 : in  SYSTEM_VALUE_TYPE);     --optional

procedure WAIT_ANY (
   STATUS : out COND_VALUE_TYPE;        --optional
   RESULT : out INTEGER;                --optional
   TIME   : in  DATE_TIME_TYPE;         --optional
   VALUE1 : in  SYSTEM_VALUE_TYPE;      --optional
   VALUE2 : in  SYSTEM_VALUE_TYPE;      --optional
   VALUE3 : in  SYSTEM_VALUE_TYPE;      --optional
   VALUE4 : in  SYSTEM_VALUE_TYPE);     --optional
```

Arguments

STATUS

This argument receives the completion status of the WAIT_ALL or WAIT_ANY procedure.

RESULT

This argument receives the argument number of the object that satisfied the wait. The value 0 means that the wait was satisfied by a timeout, as specified by the TIME argument. Otherwise, the value placed in the RESULT argument identifies for the WAIT_ANY procedure the object that satisfied the wait, where 1 indicates the first object in the list, and so forth.

With the WAIT_ALL procedure, 0 means that the procedure timed out, otherwise, the result is an integer in the range of 1 to 4, the exact value being unpredictable. The value of the RESULT argument is undefined if the procedures terminate unsuccessfully.

TIME

This argument supplies an absolute time or time interval. At the specified absolute time, or after the specified interval, the wait is satisfied regardless of the states of the specified objects.

VALUE1,VALUE2,VALUE3,VALUE4

These arguments supply one to four values of type PROCESS, SEMAPHORE, EVENT, MUTEX, DEVICE, or AREA. If no values are listed, the wait is satisfied by the timeout if one is specified, or immediately if no timeout is specified. VALUE1 may also be a value of type PORT. To specify that VALUE2, VALUE3, or VALUE4 is a PORT type, you must specify the address of the PORT variable. If you are waiting on fewer than four objects, omit VALUE4 first, then VALUE3, VALUE2, and then VALUE1.

Notes:

The WAIT procedures return immediately if one of their objects is deleted. The deletion is indicated by KER_BAD_VALUE. Both procedures also return immediately if the necessary conditions were satisfied already (before the call). Therefore, the elapsed time is only the time required to perform a procedure call, and any specified TIMEOUT value is irrelevant.

Note that, if more than one wait condition satisfies the wait, the RESULT argument does not have any predictable value.

The WAIT_ANY procedure waits for any one of a number of conditions to occur, up to a specified time. It might be used in a device driver to wait for a device interrupt or device timeout. In a multiport server, it might wait for a message to arrive on any one of several ports.

If an asynchronous exception (such as KER_POWER_SIGNAL) is delivered to a waiting process, several actions are possible, depending on the action of the exception handler:

- If the handler returns FALSE, the exception is "resignaled", meaning that the stack is searched for another handler; here the process may not reenter the waiting state.
- If the handler returns TRUE (meaning "exception handled"), the process reenters the waiting state.

- The conditions for satisfying waits for the various objects and the effects of waiting for each type of object (both procedures have the same effect on their arguments) are as follows:
  - PROCESS objects

    A wait for a process is satisfied when it terminates. Waiting for a process causes no modification to the object, and all waiting processes continue if their wait conditions are otherwise satisfied.
  - PORT objects

    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 object, and all waiting processes continue if their wait conditions are otherwise satisfied.
  - SEMAPHORE objects

    A wait for a semaphore is satisfied when the object is signaled. Waiting for a semaphore causes the semaphore count to be decremented if the wait is satisfied by signaling the semaphore; at most one process continues as the result of signaling a semaphore.
  - EVENT objects

    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.
  - DEVICE objects

    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. Only one process continues as a result of the action of an interrupt service routine.
  - AREA objects

    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. When a referencing job's main process is deleted, a check is made and 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.

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.

Status Values

KER_SUCCESS	The procedure completed successfully.
KER_BAD_COUNT	The procedure call specified an incorrect number of arguments.
KER_BAD_TYPE	An argument in the object-list is not a type that can be waited for.
KER_BAD_VALUE	An argument in the object-list is invalid or refers to a deleted object.
KER_NO_ACCESS	An argument specified is not accessible to the calling program.

Appendix B

Debugger Command Summary

This appendix lists and describes the commands used by the VAXELN Remote Debugger and summarizes the VAX/VMS Debugger commands that can be used with the remote debugger. You can obtain further information on these commands by using the VAX/VMS Debugger HELP facility.

B.1 VAXELN Remote Debugger Commands

This section lists the syntax and gives a description of each VAXELN Remote Debugger command. The commands are listed alphabetically.

```
CREATE JOB program_name [(argument[,...])] [/LOAD=filespec /KERNEL_STACK[=n] /PRIORITY=n]
```

Creates a job on the target system, running the designated program. The program name is specified as a string expression. Separate any arguments to the program by commas in a parenthesized list. There can be up to 16 arguments, all of which are strings.

If you specify the optional /LOAD qualifier, the program image is installed on the target system prior to creating the job. The program image file is opened in the context of the target system, not the context of the host system.

The /KERNEL_STACK qualifier is used to supply the size, in pages, of the kernel mode stack for jobs running this program.

The /PRIORITY qualifier specifies the starting job priority for this program.

CTRL/C

Aborts the operation in progress and enters the debugger's system session. You can use the system session to enter commands when the command session is not waiting for debug input (if, for example, the job attached to the command session is in a running state).

The prompt RDBG*> indicates that the debugger is in the system session.

CTRL/Y

Interrupts the debugger and returns you to the DCL prompt. Unlike the VAX/VMS Debugger, the remote debugger does not allow you to type DEBUG to return to the debugger prompt.

CTRL/Z

Causes orderly termination of the debugging session. Its effect is identical to the EXIT command.

DELETE JOB [job_specifier] [/UNLOAD]

Deletes a job running on the target system. If you do not provide a job specifier parameter, the job attached to the command session is deleted. (The master process is deleted, along with all subprocesses and all data and kernel objects created by any of the processes in the job.)

If the deleted job was attached to the command session, then that command session is ended and the system session is entered.

By providing a fully qualified job specifier, that is, one that contains a process number, you can delete a specific process from the specified job. For example, the command DELETE JOB 5.2 will delete only process 2 in job 5.

Using the optional /UNLOAD qualifier removes the program image from the system after the job is deleted. The /UNLOAD qualifier only unloads program images that were loaded using the CREATE JOB/LOAD command or by the LOAD_PROGRAM service. If there are any other jobs on the system that are running the same program, the program image is not unloaded until all jobs referencing the image exit.

SET JOB /CURRENT [/IMAGE=filespec] [job_specifier]

Establishes a command session with the specified job.

If a job is not in a debug-wait state, you must use the SET JOB/HALT command before you can begin a command session with the job. The SET JOB/CURRENT command causes the symbol table information to be read from the executable image file that is associated with the remote job and prepares to accept debugging commands.

If you do not supply the job specifier parameter, the command session begins with the last job that requested debugger attention.

If the SET JOB/CURRENT command is issued during a command session, a new command session is begun; all breakpoints and tracepoints are cleared for every process in the previous job and any debug context for that job is lost. Any processes in the original job that were in a debug-wait state remain in such a state until a SET JOB/CONTINUE command is issued or the process is deleted.

The optional /IMAGE qualifier allows you to provide a file specification indicating a local copy of the image being run by the target job.

Be certain that the program specified by the /IMAGE qualifier is identical to the image being run by the target job, otherwise, you will get erroneous results. If the /IMAGE qualifier is not provided, the remote debugger uses the file specification provided by the System Builder, or the file specification provided by the program loader utility.

SET JOB /CONTINUE [job_specifier [,job_specifier...]]

Causes the specified jobs that were in a debug-wait state to continue execution. A job can be waiting for debugger attention because it either started under debugger control, raised an exception, or was placed in a debug-wait state using the SET JOB/HALT command.

If no job specifier is provided, the job attached to the current command session is continued.

SET JOB/HALT [job_specifier [,job_specifier...]]

Causes the specified jobs to enter a debug-wait state. The jobs are left in a debug-wait state until either a SET JOB/CONTINUE or a SET JOB/CURRENT or DELETE JOB command is executed.

This command has the same effect that a CTRL/Y DEBUG operation has in the VAX/VMS environment. For example, if the program associated with the current command session is in an infinite loop, you can type CTRL/C (to get into the system session), followed by a SET JOB/HALT to reestablish user control.

By using a fully qualified job specifier, you can cause a specific process in the job to enter a debug-wait state. For example, the command SET JOB/HALT 5.2 causes process number 2 in job 5 to enter a debug-wait state. By default, the master process is placed in a debug-wait state.

SET JOB /PRIORITY=nn [job_specifier [,...]]

Allows you to dynamically change job priorities of jobs running on the target system. If no job specifier is provided, the priority of the current job is set. The priority can have any value between 0 (highest) and 31 (lowest). Initially, when the job is created, the job priority is set to the value specified during the system build process, or, as specified to the LOAD_PROGRAM service.

SET TIME [time_string]

Sets the system time on the remote node. The time string must be in the standard DCL format for absolute times ([dd-mmm-yyyy[:]] [hh:mm:ss.cc]). If no time string is specified, then the system time on the host system is used.

SHOW JOB [job_specifier [,job_specifier...]] {/BRIEF, /FULL}

Displays information about one or more jobs running on the target system. If you do not enter a job specifier, information about the job associated with the command session is displayed.

The /BRIEF qualifier is the default. It causes minimal information about the job to be displayed.

The /FULL qualifier causes more detailed job information as well as information about each process running in the job.

SHOW SYSTEM {/BRIEF, /FULL}

Displays system statistics for the target system. The /BRIEF qualifier (the default) displays a list of jobs in the system and information about the status of each. The /FULL qualifier displays this information and, additionally, displays information on system resources.

SHOW TIME

Displays the target system's current date and time.

B.2 Debugger Command Summary

The following sections list the debugger commands that you can use with the remote debugger in functional groupings, along with brief descriptions. Use these groupings to orient yourself among the commands.

B.2.1 Starting and Terminating a Debugging Session

EXIT, CTRL/Z, QUIT	Returns control to DCL
ATTACH	Passes control of your terminal from the current process to another process (like $ ATTACH)
SPAWN	Creates a subprocess. Lets you issue DCL commands without interrupting your debugging context (like $ SPAWN)

B.2.2 Controlling and Monitoring Program Execution

GO	Starts or resumes program execution
STEP	Executes the program up to the next line, instruction, or specified instruction
(SET,SHOW) STEP	(Establishes, displays) the current step parameters
(SET,SHOW,CANCEL) BREAK	(Sets, displays, cancels) breakpoints
(SET,SHOW,CANCEL) TRACE	(Sets, displays, cancels) tracepoints
(SET,SHOW,CANCEL) WATCH	(Sets, displays, cancels) watchpoints
(SET,CANCEL) EXCEPTION BREAK	(Sets, cancels) exception breakpoints
SHOW CALLS	Identifies the currently active routine calls
SHOW STACK	Gives additional information about the currently active routine calls
CALL	Calls a routine

B.2.3 Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location
DEPOSIT	Changes the value of a variable or the contents of a program location
EVALUATE	Evaluates a language or address expression

B.2.4 Controlling Type Selection and Symbolization

(SET,SHOW,CANCEL) RADIX	(Establishes, displays, restores) the radix for data entry and display
(SET,SHOW,CANCEL) TYPE	(Establishes, displays, restores) the type to be associated with untyped program locations
SET MODE [NO]G_FLOAT	Controls whether double precision floating-point constants are interpreted as G_FLOAT or D_FLOAT
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name + byte offset
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses
SYMBOLIZE	Converts a virtual address to a symbolic address

B.2.5 Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program
(SET,SHOW,CANCEL) MODULE	"Sets" a module by loading its symbol records into the debugger's symbol table, identifies, cancels a "set" module
(SET,SHOW,CANCEL) IMAGE	"Sets" a shareable image by loading data structures into the debugger's symbol table, identifies, cancels a "set" image
SET MODE [NO]DYNAMIC	Controls whether or not modules are "set" automatically when the debugger interrupts execution
ALLOCATE	Expands the debugger's memory pool to let you "set" more modules
(SET,SHOW,CANCEL) SCOPE	(Establishes, displays, restores) the scope for symbol lookup

B.2.6 Displaying Source Code

TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
(SET,SHOW,CANCEL) SOURCE	(Creates, displays, cancels) a source directory search list
SEARCH	Searches the source code for the specified string
(SET,SHOW) SEARCH	(Establishes, displays) the search parameters for the SEARCH command
(SET,SHOW) MAX_SOURCE_FILES	(Establishes, displays) the maximum number of source files that may be kept open at one time
(SET,SHOW) MARGINS	(Establishes, displays) the left and right margin settings for displaying source code

B.2.7 Screen Mode

SET MODE [NO]SCREEN	Enables/disables screen mode
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
DISPLAY	Modifies an existing display
(SET,SHOW,CANCEL) DISPLAY	(Creates, identifies, deletes) a display
(SET,SHOW,CANCEL) WINDOW	(Creates, identifies, deletes) a window definition
SELECT	Selects a display for a display attribute
SHOW SELECT	Identifies the displays selected for each of the display attributes
SCROLL	Scrolls a display
SAVE	Saves the current contents of a display into another display
EXTRACT	Saves a display or the current screen state into a file
EXPAND	Expands or contracts a display
MOVE	Moves a display across the screen
(SET,SHOW) TERMINAL	(Establishes, displays) the height and width of the screen
CTRL/W,DISPLAY/REFRESH	Refreshes the screen

B.2.8 Source Editing

EDIT	Invokes an editor during a debugging session
(SET,SHOW) EDITOR	(Establishes, identifies) the editor invoked by the EDIT command

B.2.9 Defining Symbols

DEFINE	Defines a symbol as an address, command, or value
DELETE (UNDEFINE)	Deletes symbol definitions
(SET,SHOW) DEFINE	(Establishes, displays) the definition parameter (address, command, or value) for the DEFINE command
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined

B.2.10 Keypad Mode

SET MODE [NO]KEYPAD	Enables/disables keypad mode
DEFINE/KEY	Creates key definitions
DELETE/KEY (UNDEFINE/KEY)	Deletes key definitions
SET KEY	Establishes the key definition state
SHOW KEY	Displays key definitions

B.2.11 Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures
(SET,SHOW) LOG	(Specifies, identifies) the debugger log file
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed

SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command
(SET,SHOW) ATSIGN	(Establishes, displays) the default file specification that the debugger uses to search for command procedures
@file-spec	Executes a command procedure

B.2.12 Control Structures

IF	Executes a list of commands conditionally
FOR	Executes a list of commands repetitively
REPEAT	Executes a list of commands repetitively
WHILE	Executes a list of commands conditionally
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop

B.2.13 Debugging Special Cases

SET OUTPUT [NO]TERMINAL	Controls whether debugger output, except for diagnostic messages, is displayed or suppressed
(SET,SHOW) LANGUAGE	(Establishes, displays) the current language
(SET,SHOW) EVENT_FACILITY	(Establishes, identifies) the current run-time facility for language-specific events
(SET,SHOW) TASK	Modifies the tasking environment, displays task information

Appendix C

VAXELN-Related Features Added to VAX Ada

The following features are designed to serve both VAX Ada and VAXELN Ada programmers. These features were added to VAX Ada after the release of Version 1.0, and so are not documented in the Version 1.0 *VAX Ada Language Reference Manual.* They will be incorporated into the Version 2.0 documentation.

C.1 Main Task Storage Allocation

VAX Ada (Version 1.1 and later) provides pragma MAIN_STORAGE to allow a fixed-size stack and stack storage areas to be specified for a main task (the task associated with a main program). The form of this pragma is as follows:

```
pragma MAIN_STORAGE(
   main_storage_option [, main_storage_option]);

main_storage_option :=
     [WORKING_STORAGE => ] static_simple_expression
   | [TOP_GUARD => ] static_simple_expression
```

The simple expression given for a main storage option must be a nonnegative static expression of some integer type; its value specifies the number of storage units (bytes) to be allocated for the working storage (stack) area or guard pages. For both WORKING_STORAGE and TOP_GUARD, the value specified is rounded up to an integral number of pages (where one page is 512 bytes). If the value is zero for WORKING_STORAGE, a default size is assumed; if the value is zero for TOP_GUARD, no guard pages are provided.

A pragma MAIN_STORAGE is only allowed in the outermost declarative part of a subprogram that is a library unit; at most, one such pragma is allowed in a subprogram. If it occurs in a subprogram other than a main program, this pragma has no effect.

Note that on the VAX/VMS system, use of this pragma causes the main stack to be allocated in P0 space (rather than in the default P1 space); on VAXELN, the main stack is always allocated in P0 space.

C.2 The Package SYSTEM

To allow a choice between VAX/VMS and VAXELN targets, VAX Ada (Version 1.2 and later) defines the type SYSTEM.NAME with two enumeration literals: VAX_VMS and VAXELN. The default value of VAX_VMS for the constant SYSTEM.SYSTEM_NAME can be changed with the pragma SYSTEM_NAME or the ACS SET PRAGMA command (see Chapter 4 for more information).

C.3 VAX Ada Additions to Package SYSTEM

For VAX Ada Version 1.1 and later, the following operations and types are declared.

C.3.1 Register Operations

VAX Ada provides the following operations for reading from and writing to device registers.

```
function  READ_REGISTER (SOURCE : UNSIGNED_BYTE)
   return UNSIGNED_BYTE;
function  READ_REGISTER (SOURCE : UNSIGNED_WORD)
   return UNSIGNED_WORD;
function  READ_REGISTER (SOURCE : UNSIGNED_LONGWORD)
   return UNSIGNED_LONGWORD;

procedure WRITE_REGISTER(SOURCE : UNSIGNED_BYTE;
                         TARGET : out UNSIGNED_BYTE);
procedure WRITE_REGISTER(SOURCE : UNSIGNED_WORD;
                         TARGET : out UNSIGNED_WORD);
procedure WRITE_REGISTER(SOURCE : UNSIGNED_LONGWORD;
                         TARGET : out UNSIGNED_LONGWORD);
```

The READ_REGISTER functions return the value of a variable reference (byte, word, or longword). The WRITE_REGISTER procedures load a specified value or group of values into a specified target variable reference (byte, word, or longword). Each READ_REGISTER and WRITE_REGISTER operation is performed by a single machine instruction and is not affected by any compiler optimizations. Use of these operations is the only safe method for reading or writing a device register, and can also be used to read or write a shared variable.

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

VAX Ada also declares the following operations to provide Ada equivalents for the VAX Move From Process Register (MFPR) and Move To Process Register (MTPR) instructions.

```
function  MFPR (REG_NUMBER : INTEGER)
   return UNSIGNED_LONGWORD;

procedure MTPR (REG_NUMBER : INTEGER,
                SOURCE     : UNSIGNED_LONGWORD);
```

Function MFPR returns the current contents of the specified VAX internal processor register; procedure MTPR moves a specified value into a specified VAX internal processor register. In both cases, the calling program must be running in kernel mode.

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

C.3.2 Interlocked Instructions

VAX Ada declares the following type and operations to provide Ada equivalents for the VAX Branch on Bit Set and Set Interlocked (BBSSI), Branch on Bit Clear and Clear Interlocked (BBCCI), and Add Aligned Word Interlocked (ADAWI) instructions. These instructions interlock memory accesses, and thus provide a means for synchronizing access to shared memory across processors.

```
procedure CLEAR_INTERLOCKED (BIT       : in out BOOLEAN;
                             OLD_VALUE : out BOOLEAN);
procedure SET_INTERLOCKED   (BIT       : in out BOOLEAN;
                             OLD_VALUE : out BOOLEAN);

type ALIGNED_SHORT_INTEGER is record
      VALUE : SHORT_INTEGER := 0;
   end record;
for  ALIGNED_SHORT_INTEGER use
   record
      at mod 2;
   end record;

procedure ADD_INTERLOCKED (ADDEND : in     SHORT_INTEGER;
                           AUGEND : in out ALIGNED_SHORT_INTEGER;
                           SIGN   : out    INTEGER);
```

The CLEAR_INTERLOCKED and SET_INTERLOCKED procedures clear or set a single bit and return the previous value of the bit, using the VAX BBCCI and BBSSI instructions.

The type ALIGNED_SHORT_INTEGER specifies a word-sized integer that is word-aligned (the type STANDARD.SHORT_INTEGER is a word-sized integer that is byte-aligned).

The ADD_INTERLOCKED procedure adds two signed-word integers (ADDEND and AUGEND), and places the result in AUGEND using the VAX ADAWI instruction. SIGN is assigned the integer result -1 if the new value of AUGEND is negative; 0 if AUGEND is zero; +1 if AUGEND is positive.

C.3.3 Queue Instructions

VAX Ada declares the following types and operations to provide Ada equivalents for the VAX queue instructions: Insert Queue into Queue at Head, Interlocked (INSQHI); Insert Entry into Queue at Tail, Interlocked (INSQTI); Remove Entry from Queue at Head, Interlocked (REMQHI); and Remove Entry from Queue at Tail, Interlocked (REMQTI).

```
type INSQ_STATUS is (OK_NOT_FIRST, FAIL_NO_LOCK, OK_FIRST);
type REMQ_STATUS is (OK_NOT_EMPTY, FAIL_NO_LOCK,
                     OK_EMPTY, FAIL_WAS_EMPTY);

procedure INSQHI (ITEM   : in  ADDRESS;
                  HEADER : in  ADDRESS;
                  STATUS : out INSQ_STATUS);

procedure REMQHI (HEADER : in  ADDRESS;
                  ITEM   : out ADDRESS;
                  STATUS : out REMQ_STATUS);

procedure INSQTI (ITEM   : in  ADDRESS;
                  HEADER : in  ADDRESS;
                  STATUS : out INSQ_STATUS);

procedure REMQTI (HEADER : in  ADDRESS;
                  ITEM   : out ADDRESS;
                  STATUS : out REMQ_STATUS);
```

The types INSQ_STATUS and REMQ_STATUS are defined to represent the status results of the procedures for manipulating self-relative queues (queues where the forward and backward links are defined as offsets from one link to the next, rather than as virtual addresses).

The INSQHI, REMQHI, INSQTI, and REMQTI procedures perform queue insertion and removal operations at the head and tail of a self-relative queue. The address values of HEADER and ITEM must be quadword-aligned. The enumeration value assigned to STATUS gives the state of the queue after the operation has been executed.

Note that the VAX INSQHI, REMQHI, INSQTI, and REMQTI instructions have "entry" parameters, but because "entry" is a reserved word in Ada, the "entry" parameter has been named ITEM in these procedures.

INDEX

A

ACCEPT_CIRCUIT procedure • 9-20, A-22
ACS • 3-1, 3-3, 3-4, 3-5, 3-7, 3-10, 3-11
  additional commands • 3-15
  compilation commands • 3-19
  file naming conventions for • 3-5
  library management commands • 3-13
  using • 2-2
ADA$ELAB routine • 3-29
ADA command • 3-19
  qualifiers for • 3-20
ADAWI instruction
  see ADD_INTERLOCKED procedure
ADD_INTERLOCKED procedure • 1-6, C-4
Add Device Description menu • 4-4, 4-18, 4-19
Add Program Description menu • 4-4, 4-15
Add Terminal Description menu • 4-4, 4-22
ALIGNED_SHORT_INTEGER type • 1-6, C-4
ALLOCATE_MAP procedure • 4-16, 9-16, A-24
ALLOCATE_MEMORY procedure • 9-19, A-26
ALLOCATE_PATH procedure • 9-16, A-28
Analog-to-Digital converter • 9-14
AREA object • 9-2
Arrow keys • 4-5, 4-6
ASCII characters
  specifying 8-bit for an executable system terminal • 4-24
  specifying 8-bit for a system console terminal • 4-25
AST • 1-4
AST_ENTRY attribute • 1-2, 1-4, 3-8, 3-16, 8-10
AST_ENTRY pragma • 1-4, 3-8, 3-16, 8-10
Asynchronous serial line controller • 9-14
Asynchronous system traps
Asynchronous system traps (cont'd.)
  see also AST_ENTRY pragma and AST_ENTRY attribute
  see AST
ATTACH command • 3-16
Attributes
  affecting application performance • 7-19
AUTH_ADD_USER procedure • 9-13, A-30
AUTH_MODIFY_USER procedure • 9-13, A-33
AUTH_REMOVE_USER procedure • 9-13, A-36
AUTH_SHOW_USER procedure • 9-13, A-38
Authorization service
  specifying for an executable system • 4-14
Authorization service procedures • 9-13
Authorization utility procedures • 9-12
Autoload driver
  specifying for an executable system • 4-20
AXV_INITIALIZE procedure • 9-14, A-40
AXV_READ procedure • 9-14, A-42
AXV_WRITE procedure • 9-14, A-45

B

BACK (EBUILD editing function) • 4-6
Baud rate
  specifying for a system terminal • 4-23
BBCCI instruction
  see CLEAR_INTERLOCKED procedure
BBSSI instruction
  see SET_INTERLOCKED procedure
Bodies • 3-2, 3-7, 3-11, 3-16
  see also Library bodies
Booting systems • 5-1
  from disks • 5-1
  from tape • 5-1

Booting systems (cont'd.)
using Ethernet • 5-3
see also Downline loading
Boot method
see also Booting systems
required for copying system images • 5-2
required for downline loading • 5-7
specifying for an executable system • 4-9
Bootstrap loader • 5-3
configuring for downline loading • 5-6
device addresses recognized by • 5-4
Breakpoint • 6-21
Building systems • 4-1
from the System Builder Main menu • 4-4 to 4-7
with the EBUILD command • 4-7
Build System choice
from System Builder Main menu • 4-4

C

CANCEL_EXIT_HANDLER procedure • 9-17, A-46
CANCEL BREAK command • 6-21
CANCEL WATCH command • 6-23
CHECK command • 3-7, 3-13
Circuits • 7-5
CLEAR_EVENT procedure • 9-4, A-47
CLEAR_INTERLOCKED procedure • 1-6, C-4
Closure • 3-6, 3-20, 3-25
Command session
interrupting and restarting • 6-14
Compilation • 3-1, 3-19
see also ADA, COMPILE, RECOMPILE commands
as source of obsolete units • 3-3
basic concepts behind • 3-1
effect of pragma INLINE on • 3-6
effect of unit dependences on • 3-3, 3-5
order-of-compilation rules for • 3-5
organization of files for efficient • 3-4
organizing mixed VAXELN and VAX Ada units for • 3-16
results of successful • 3-6, 3-11
separate • 3-2, 3-11
terminology related to • 3-1
Compilation unit files • 3-11
Compilation units • 3-2
Compilation units (cont'd.)
see also Program libraries
Ada rules for naming • 3-4
compilation closure of • 3-6
current and obsolete • 3-3
dependences affected by context clauses • 3-3
dependences affected by SYSTEM.SYSTEM_NAME • 3-3
dependences among • 3-3, 3-6
difference from source files • 3-4
effects of dependences on compiling • 3-5
execution closure of • 3-7
kinds of • 3-2
obsolete • 3-4, 3-6, 3-9
source file naming conventions for • 3-4
target-related dependences among • 3-7, 3-8
COMPILE command • 3-7, 3-19, 3-20
qualifiers for • 3-20
CONNECT_CIRCUIT procedure • 9-7, 9-20, A-48
Connect time
specifying for an executable system • 4-12
Console input-output • 6-15
Console terminal
describing from the System Builder • 4-24
device description for • 4-19
specifying for an executable system • 4-8
Context clauses
and closure of a set of compilation units • 3-7
and obsolete units • 3-3
and order of compilation • 3-5
effect on compilation unit dependences • 3-3
CONTROL_C_INTERCEPTION package • 8-10
Control characters
specifying interpretation of for a system console terminal • 4-25
specifying interpretation of for a system terminal • 4-24
Controllers
determining register and vector addresses for • 4-20
Control status register address • 4-20
Copied source files • 3-11
COPY_FILE procedure • 9-18, A-51
COPY FOREIGN command • 3-13
Copying bootable images • 5-2
COPYSYS command procedure • 5-2
COPY UNIT command • 3-7, 3-11, 3-13

CREATE_AREA procedure • 9-2, A-53
CREATE_DEVICE procedure • 4-16, 4-22, 9-3, A-55
CREATE_DIRECTORY procedure • 9-18, A-59
CREATE_EVENT procedure • 9-4, A-61
CREATE_JOB procedure • 8-2, A-62
CREATE_MESSAGE procedure • 9-5, 9-20, A-64
CREATE_MUTEX procedure • 9-5, A-66
CREATE_NAME procedure • 9-6, 9-20, A-67
CREATE_PORT procedure • 9-7, 9-20, A-69
CREATE_PROCESS procedure • 8-3, 9-8, A-71
CREATE_SEMAPHORE procedure • 9-9, A-73
CREATE JOB command • B-1
CREATE LIBRARY command • 3-12, 3-13
  changing the value of SYSTEM.SYSTEM_NAME with • 3-8
CREATE SUBLIBRARY command • 3-12, 3-13, 3-17
  changing the value of SYSTEM.SYSTEM_NAME with • 3-8
CTRL/C
  in the remote debugger • B-1
CTRL/E • 4-5
  used with System Builder menus • 4-6
CTRL/H • 4-5
  used with System Builder menus • 4-6
CTRL/R • 4-5
  used with System Builder menus • 4-6
CTRL/U • 4-5
  used with System Builder menus • 4-6
CTRL/Y
  in the remote debugger • B-1
  in VAXELN environment • 8-10
CTRL/Z
  in the remote debugger • B-1
Currency • 3-3, 3-6, 3-13, 3-20, 3-25
CURRENT_PROCESS procedure • 9-8, A-75

D

Data file • 2-3
Datalink addresses • 5-4
DDCMP
  specifying for an executable system • 4-24
DEBUG/REMOTE command • 6-11
  see also Remote debugger
  use of for downline loading • 5-9
Debugger
  see also Remote debugger
  building into a VAXELN system • 4-7
  specifying for an individual program • 4-16
Debugging
  see also Remote debugger
  downline loading during • 5-9
Debug-wait state • 6-3
DECLARE_EXIT_HANDLER procedure • 9-17, A-76
DELETE (EBUILD editing function) • 4-6
DELETE_AREA procedure • 9-2, A-77
DELETE_DEVICE procedure • 9-3, A-78
DELETE_EVENT procedure • 9-4, A-79
DELETE_FILE procedure • 9-18, A-80
DELETE_MESSAGE procedure • 9-5, 9-20, A-81
DELETE_MUTEX procedure • 9-5, A-82
DELETE_NAME procedure • 9-6, 9-20, A-83
DELETE_PORT procedure • 9-20, A-84
DELETE_PROCESS procedure • A-85
DELETE_SEMAPHORE procedure • 9-9, A-86
DELETE JOB command • 6-17, B-1
DELETE LIBRARY command • 3-14, 3-28
DELETE SUBLIBRARY command • 3-14
DELETE UNIT command • 3-14
Dependences
  see also Compilation, Compilation units, SYSTEM_NAME constant
  compilation unit • 3-3
Development environment • 1-2
Device controller name
  specifying for an executable system • 4-19
Device driver
  specifying autoload of for an executable system • 4-20
Device drivers
  execution mode for • 4-16
Device driver utility procedures • 9-13
Device interrupt priority
  specifying for an executable system • 4-20
Device names
  external • 7-7
DEVICE object • 9-3
Device register address
  determining • 4-20
  specifying for an executable system • 4-19

Device registers
predefined operations for • 1-5, C-2
reading from • C-2
writing to • C-2
Devices
datalink default addresses for • 5-4
describing from the System Builder • 4-18
equivalent SYSGEN names for QBUS • 4-21
equivalent SYSGEN names for UNIBUS • 4-21
parallel • 7-6
real-time • 7-6
serial • 7-6
supported • 7-6
Device vector address
determining • 4-20
specifying from the System Builder • 4-19
DIGITAL Data Communications Message Protocol
see DDCMP
DIRECT_IO package • 7-1
DIRECT_MIXED_IO package • 7-1
Direct memory mapping access • 9-15
DIRECTORY_CLOSE procedure • 9-18, A-87
DIRECTORY_LIST procedure • 9-18, A-88
DIRECTORY_OPEN procedure • 9-18, A-90
DIRECTORY command • 3-14
Directory name
external files • 7-8
DISABLE_ASYNCH_EXCEPTION procedure • 9-16, A-92
DISABLE_INTERRUPT procedure • 9-19, A-93
DISABLE_SWITCH procedure • 9-21, A-94
DISCONNECT_CIRCUIT procedure • 9-7, 9-20, A-95
Disk/volume names
specifying for an executable system • 4-9
Disk files • 7-2
Disks
booting systems from • 5-1
copying system images to • 5-2
Disk utility procedures • 9-17
DISMOUNT_TAPE_VOLUME procedure • 9-18, A-96
DISMOUNT_VOLUME procedure • 9-17, A-97
DLV_INITIALIZE procedure • 9-15, A-98
DLV_READ_BLOCK procedure • 9-15, A-100
DLV_READ_STRING procedure • 9-15, A-101
DLV_WRITE_STRING procedure • 9-15, A-102
DO (EBUILD editing function) • 4-5, 4-6, 4-7, 4-15, 4-18, 4-22
Downline loading • 2-4, 5-3
adding target to host database during • 5-5
configuring a host for • 5-4
configuring bootstrap loader for • 5-6
determining target hardware address for • 5-5
enabling event logging during reload • 5-9
preliminary steps for • 5-4
procedure for • 5-7
reloading using • 5-8, 5-10
using the remote debugger for • 5-9
DRV_INITIALIZE procedure • 9-15, A-103
DRV_READ procedure • 9-15, A-105
DRV_WRITE procedure • 9-15, A-106
Dynamic program space
specifying for an executable system • 4-11

E

EBUILD command • 4-1, 4-7
qualifiers for • 4-2
using to generate System Builder menus • 4-4
Edit Console Characteristics menu • 4-4, 4-25
Edit Device Descriptions menu • 4-4, 4-6, 4-18
Editing
System Builder menus • 4-5
Edit Network Node Characteristics menu • 4-4, 4-13
Edit Program Descriptions menu • 4-4, 4-6, 4-15
Edit System Characteristics menu • 4-4, 4-7
Edit Terminal Descriptions menu • 4-4, 4-6, 4-22
ENABLE_ASYNCH_EXCEPTION procedure • 9-16, A-107
ENABLE_INTERRUPT procedure • 9-19, A-108
ENABLE_SWITCH procedure • 9-21, A-109
ENTER_KERNEL_CONTEXT procedure • 9-21, A-111
ENTER FOREIGN command • 3-14
ENTER UNIT command • 3-7, 3-11, 3-14, 3-28
Environment task • 6-4
see Main task
Error messages
including text for during link • 3-26
Escape sequences
specifying recognition of for a system console terminal • 4-25

Escape sequences (cont'd.)
  specifying recognition of for a system terminal • 4-24
Ethernet
  determining hardware address when downline loading • 5-5
  determining network device for • 4-13
  using to downline load executable systems • 5-3
Event logging
  during downline loading • 5-9
EVENT object • 9-3
Exception handling procedures • 9-16
EXIT (EBUILD editing function) • 4-6
EXIT_PROCESS procedure • 9-8, A-112
EXIT command • 3-16
Exit utility procedures • 9-17
EXPORT command • 3-7, 3-27
  changing the value of SYSTEM.SYSTEM_NAME with • 3-8
  package elaboration with • 3-28
  qualifiers for • 3-28
Exporting Ada units • 3-27
  see also EXPORT command
  from more than one program library • 3-28
External file attributes • 7-9
External files
  directory name • 7-8
  type • 7-8
External node names • 7-6
EXTRACT SOURCE command • 3-14

F

F15 key • 4-6
F16 key • 4-6
Failure
  recovering from machine • 5-10
FDL (File Definition Language)
  attributes for Ada files • 7-13
  primary and secondary attributes • 7-10
  rules for using • 7-9
  tuning external files • 7-19
  use with form string in VAXELN Ada • 1-5
FDL attributes
  table of • 7-10
File • 7-2
File (cont'd.)
  see indexed file
  see nonsequential file
  see relative file
  see sequential file
  commonly used FDL attributes
    table of • 7-13
  disk • 7-2
  external attributes • 7-9
  external node names • 7-6
  FDL attributes • 7-10
  FDL attributes for tuning external • 7-19
  network • 7-4
  rules for specifying FDL attributes for external • 7-9
  standard • 7-8
  tape • 7-3
  type • 7-8
  VAXELN_CIRCUIT attribute • 7-12
File access • 7-2
File Access Listener
  specifying for an executable system • 4-13, 4-15
File Definition Language
  see FDL
Files
  default name of system image • 4-7
  naming conventions for Ada source • 3-4
  program library • 3-10
File service procedures • 9-17
File utility procedures • 9-17
Fixed-size stacks • 1-5
FORM parameter • 1-5
  see also Form string
Form string • 1-2
  see FORM parameter
  implementation of • 1-5
FREE_MAP procedure • 4-16, 9-16, A-113
FREE_MEMORY procedure • 9-19, A-115
FREE_PATH procedure • 9-16, A-116

G

Generic instantiations • 3-2
  obsolete • 3-6
  source file naming conventions for • 3-5
Generic units • 3-2

Generic units (cont'd.)
effect of compilation on • 3-6
GET_STATUS_TEXT procedure • 9-16
GET_TIME procedure • A-118
GET_USER procedure • 9-12, A-119
Guaranteed image list
specifying for an executable system • 4-9

H

HELP (EBUILD editing function) • 4-6
HELP command • 3-13, 3-16, 3-19, 3-24
Host machine
see also Downline loading
configuring for downline loading • 5-4
connecting to target during downline loading • 5-5
downline loading from • 5-7
network node database for • 5-5
preparing for downline loading • 5-4
reloading systems from • 5-10
Host systems
program development on • 1-2

I

INDEXED_IO package • 1-2, 1-5
INDEXED_MIXED_IO package • 1-2, 1-5
Indexed file • 1-5, 7-2
INIT_TAPE_VOLUME procedure • 9-18, A-122
INIT_VOLUME procedure • 9-17, A-124
INITIALIZATION_DONE procedure • 4-16, 9-21, A-121
Initialization procedure (VAXELN)
specifying for an individual program • 4-16
INLINE pragma
effect on compilation • 3-6
Input-output
differences from VAX Ada • 1-2
Input-output packages
indexed • 3-8, 3-16
relative • 3-8, 3-16
Input-output region size
specifying for an executable system • 4-11
INSQ_STATUS type • 1-6, C-5
INSQHI procedure • 1-6, C-5
INSQTI procedure • 1-6, C-5
Instantiations, generic • 3-2
obsolete • 3-6
source file naming conventions for • 3-5
Instruction emulation
specifying for an executable system • 4-8
Interlocked instructions
predefined type and operations for • 1-6, C-4
Internal processor registers
predefined operations for • 1-5
Interrupt priority level procedures • 9-18
Interrupt stack size
specifying for an executable system • 4-11
Interrupt time interval
specifying for an executable system • 4-12

J

Job • 6-3
changing priority of • 6-20
controlling execution of • 6-19
creating a • 6-16
deleting a • 6-17
monitoring a • 6-15
selecting for debugging • 6-18
JOB_PORT procedure • 9-7, 9-20, A-129
Job port message limit
specifying for an individual program • 4-17
Job priority
specifying for an executable system • 4-17
Jobs (VAXELN)
see also Programs
affected by Ada task waits • 8-10
building into a system • 4-1
determining number of page tables for in a system • 4-10
pool blocks required for • 4-10
relationship to Ada tasks • 8-2, 8-8
specifying debugger for • 4-16
specifying port message limit for • 4-17
specifying priority of • 4-17
starting automatically when system is booted • 4-16
Job specifier • 6-4

K

KER_POWER_SIGNAL exception • 4-18
Kernel debugger
  invoking • 6-23
Kernel stack
  specifying for an individual program • 4-17
KWV_INITIALIZE procedure • 9-14, A-130
KWV_READ procedure • 9-14, A-133
KWV_WRITE procedure • 9-14, A-135

L

Libraries
  see Program libraries, Sublibraries
Library bodies • 3-2, 3-13, 3-14
  Ada rules for naming • 3-4
  and execution closure • 3-7
  and unit dependences • 3-3, 3-6
  effects of compilation order on • 3-6
  obsolete • 3-3
  order-of-compilation rules for • 3-5
  source file naming conventions for • 3-5
Library index files • 3-10
Library manager
  see ACS
Library specifications • 3-2, 3-5
  Ada rules for naming • 3-4
  and obsolete units • 3-3
  dependences on • 3-3, 3-6, 3-11
  effect of pragma INLINE on • 3-6
  order-of-compilation rules for • 3-5
  organizing source files for • 3-4
  source file naming conventions for • 3-4
Library units • 3-2
  see also Library specifications, Generic instantiations, Subprograms
  Ada rules for naming • 3-4
  compilation unit dependences among • 3-3
  elaboration of • 3-26, 3-28
  exporting • 3-27
  use of MAIN_STORAGE pragma in • C-2
LINK command (ACS) • 3-7, 3-23, 3-26
  changing the value of SYSTEM.SYSTEM_NAME with • 3-8
  DCL command file created by • 3-26
LINK command (ACS) (cont'd.)
  explanation of • 3-25
  qualifiers for • 3-24
LINK command (DCL) • 3-24, 3-30
Linking • 3-1, 3-23
  see also LINK command
  Ada and non-Ada code • 3-27
  against other libraries or options files • 3-26
  against specific run-time libraries • 3-24
  basic concepts behind • 3-1, 3-3, 3-7
  effect of incomplete units on • 3-6
  effect of obsolete units on • 3-3
  in target-specific environment • 3-8
  mixed VAXELN and VAX Ada units • 3-16, 3-23
  program library files associated with • 3-10
  terminology related to • 3-1, 3-3, 3-6, 3-7
  units containing AST_ENTRY pragma or attribute • 1-4
  units containing TIME_SLICE pragma • 1-4
LOAD_PROGRAM procedure • 9-21, A-137
LOAD_UNIBUS_MAP procedure • 9-16, A-140
LOCK_MUTEX procedure • 9-6, A-142

M

MAIN_STORAGE pragma • 1-5, 8-6
  specification for • C-1
Main menu • 4-5, 4-6, 4-7, 4-15, 4-18, 4-22
  choices from System Builder • 4-4
Main program • 3-2
  relationship to Ada tasks • 8-1, 8-2
  use of MAIN_STORAGE pragma in • C-2
Main task • 8-1
  see also Main program
  see also Tasks
  controlling size of stack for • 8-6, C-1
  implementation of • 8-3, 8-5
  specifying storage for • 8-6
  stack overflow in • 8-8
  task stack for • 8-5
  termination of • 8-1
Maintenance Operation Monitor
  process created during downline loading • 5-7
  use of during system reload • 5-9

Memory
specifying maximum amount for an executable system • 4-11
MEMORY_SIZE procedure • 9-19, A-143
Memory allocation procedures • 9-19
MERGE command • 3-11, 3-14
MESSAGE object • 9-4
Message transmission procedures • 9-19
MFPR function • 1-5, 4-16, C-3
Mode
specifying execution for an individual program • 4-16
Modem
specifying for an executable system • 4-24
MOUNT_TAPE_VOLUME procedure • 9-18, A-144
MOUNT_VOLUME procedure • 9-17, A-146
MTPR procedure • 1-5, 4-16, C-3
MUTEX object • 9-5

N

NAME
see also SYSTEM_NAME
NAME object • 9-6
NAME type
enumeration literals for • C-2
NCP
using during downline loading • 5-7
using in downline loading • 5-4
using in host configuration during downline loading • 5-4
using to enable event logging • 5-9
using to prepare host for downline loading • 5-5
using to trigger a target machine • 5-8
Network Control Program
see NCP
Network device
specifying for an executable system • 4-13
Network files • 7-4
Network name server
specifying for an executable system • 4-13
Network node address
specifying during downline loading • 5-5
specifying during downling loading • 5-8
specifying for an executable system • 4-14
Network Node Characteristics menu • 5-8
Network node name
specifying during downline loading • 5-5, 5-8
specifying for an executable system • 4-13
Network segment size
specifying for an executable system • 4-15
Network service
reloading a machine having • 5-8
required for downline loading • 5-3
specifying for an executable system • 4-13
NEWBOOT (VAXELN command procedure) • 5-6, 5-8
Node triggerable • 5-8
specifying for an executable system • 4-14
Nonsequential file • 7-2

O

Object
area • 9-2
device • 9-3
event • 9-3
message • 9-4
mutex • 9-5
name • 9-6
port • 9-6
process • 9-7
semaphore • 9-8
Object file • 3-10
Obsolete units • 3-3, 3-4, 3-6, 3-9, 3-20
Output display
specifying for a system console terminal • 4-25
specifying for a system terminal • 4-23
Overflow
task stack • 8-5, 8-6, 8-8

P

Package
statically allocated variable • 6-23
Packages • 3-2, 7-1
elaboration of library • 3-28, 8-1
Page table slots
specifying for an executable system • 4-10
Parallel line interface • 9-15
Parity
specifying for a system terminal • 4-23

Pass all (cont'd.)
  specifying for a system console terminal • 4-25
  specifying for a system terminal • 4-24
PF1 key • 4-5, 4-6, 4-7, 4-15, 4-18, 4-22
PF2 key • 4-5, 4-6
PF3 key • 4-5, 4-6
PF4 key • 4-5, 4-6
PHYSICAL_ADDRESS function • A-148
Physical memory size
  specifying for an executable system • 4-12
Pool size
  specifying for an executable system • 4-10
PORT object • 9-6
Ports
  specifying for an executable system • 4-10
Power failure
  recovering from • 5-10
Powerfailure exception
  specifying for an individual program • 4-18
Printers • 7-4
PRIORITY pragma • 8-9
Procedures
  see by individual name or Appendix A
Process • 6-4
Processes (VAXELN)
  determining number of page tables for in a system • 4-10
  determining priorities for • 4-17
  master
    relationship to Ada tasks • 8-3
  relationship to Ada tasks • 8-3, 8-5, 8-9
PROCESS object • 9-7
Process priority • 8-9
  see also Tasks
  ignored for VAXELN Ada tasks • 8-9
  specifying for an individual program • 4-17
Process registers
  predefined operations for • C-3
PROGRAM_ARGUMENT_COUNT function • A-150
PROGRAM_ARGUMENT_COUNT procedure • 9-20
PROGRAM_ARGUMENT function • A-149
PROGRAM_ARGUMENT procedure • 9-20
Program argument procedures • 9-20
Program description • 6-10
Program development
  ACS commands for • 3-13, 3-15, 3-19, 3-23, 3-27
Program development (cont'd.)
  basic concepts behind Ada • 3-1
  in mixed-target environment • 3-16
  modular • 3-2
  terminology related to • 3-1
  top-down • 3-11
Program execution
  controlling • 6-20
Program libraries • 3-10
  ACS commands for managing • 3-13
  commands for setting up • 3-12
  contents of • 3-10
  currency of • 3-6
  files associated with • 3-10
  names of units in • 3-4
  sharing units among • 3-11
  updating • 3-6, 3-11
  value of SYSTEM.SYSTEM_NAME for • 3-8
Program library manager
  see ACS
Program loader procedures • 9-20
Programs
  see also Jobs (VAXELN)
  see also Main program
  building in a system • 4-4
  compiling • 3-19
  describing to System Builder • 4-15
  determining stack size for kernel mode • 4-17
  difference between VAX Ada and VAXELN Ada • 8-2
  estimating storage for tasking • 8-7
  implementation of VAXELN Ada • 8-2
  initializing before running • 4-16
  linking • 3-23
  mixing Ada with non-Ada • 8-8
  recompiling • 3-19
  recovering after a power failure • 4-18
  specifying execution mode for • 4-16
  specifying image names of to System Builder • 4-16
  specifying job priority for • 4-17
  specifying process priority for • 4-17
  task switching in • 8-8
  termination of • 8-2
Program sublibraries
  see Sublibraries
Program units • 3-2

PROTECT_FILE procedure • 9-18, A-151

Q

Q22 bus • 4-19
QBUS
  common device names for • 4-21
  determining register and vector addresses for • 4-20
  Ethernet network device for • 4-13
  specifying register address for • 4-19
Queue instructions
  predefined types and operations for • 1-6, C-5
QUIT (EBUILD editing function) • 4-6

R

RAISE_EXCEPTION procedure • 9-16
RAISE_PROCESS_EXCEPTION procedure • 9-8, 9-16, A-153
RDBG$INPUT • 6-10
RDBG$OUTPUT • 6-10
READ_REGISTER function • 1-5, C-2
READ_REGISTER functions • C-3
Real-time clock • 9-14
RECEIVE procedure • 9-5, 9-7, 9-20, A-154
Recompilation • 3-4, 3-19
  implicit • 3-9
  program library files associated with • 3-11
RECOMPILE command • 3-4, 3-7, 3-19, 3-20
  qualifiers for • 3-20
REENTER command • 3-15
RELATIVE_IO package • 1-2, 1-5
RELATIVE_MIXED_IO package • 1-2, 1-5
Relative file • 1-5, 7-2
Reloading booted systems • 5-8, 5-10
  see also Triggering, Triggers
Remote Debugger
  and Ada tasking • 6-24
  command session • 6-4
  compiling and linking considerations • 6-5
  debugging environment • 6-2
  downline loading with • 6-12
  exiting • 6-13
  interrupting and reinvoking • 6-14
  invoking • 6-11
  sample session • 6-27 to 6-30, 6-33
Remote Debugger (cont'd.)
  setting up the environment • 6-6
  support for VAX/VMS Debugger features • 6-2
  system session • 6-5
  using • 2-5
  naming of modules and pathnames with • 3-5
  receiving error message text with • 3-27
  using to detect stack overflow • 8-8
  using to downline load VAXELN systems • 5-3, 5-7, 5-9
  using to estimate task storage • 8-7
  using to measure tasking performance • 8-11
REMQ_STATUS type • 1-6, C-5
REMQHI procedure • 1-6, C-5
REMQTI procedure • 1-6, C-5
RENAME_FILE procedure • 9-18, A-157
RESUME procedure • 9-8, A-159
Running a VAXELN system • 5-1
  see also Booting systems, Downline loading
Running programs
  specifying from the System Builder • 4-16
Run-time library (VAXELN Ada)
  and Ada tasks • 8-2, 8-3, 8-5, 8-9
  linking against • 3-24

S

Secondary units • 3-2
  see also Library bodies, Subunits
SEMAPHORE object • 9-8
SEND procedure • 9-5, 9-20, A-160
Separate compilation • 3-2, 3-11
SEQUENTIAL_IO package • 7-1
SEQUENTIAL_MIXED_IO package • 7-1
Sequential file • 7-1
SET_INTERLOCKED procedure • 1-6, C-4
SET_JOB_PRIORITY procedure • 9-21, A-163
SET_PROCESS_PRIORITY procedure • 8-9, 9-8, A-164
SET_PROTECTION procedure • A-166
SET_TIME procedure • 9-21, A-168
SET_USER procedure • 9-12, A-169
SET BREAK command • 6-21
SET JOB/CONTINUE command • 6-20, B-1
SET JOB/CURRENT command • B-1
SET JOB/HALT command • 6-19, B-1
SET JOB/PRIORITY command • B-1

SET LIBRARY command • 3-13, 3-15
SET PRAGMA command • 3-8, 3-9, 3-15, C-2
SET SOURCE command • 3-20
SET TIME command • B-1
Shareable images
  building • 3-27
  linking Ada units against • 3-26
SHOW BREAK command • 6-21
SHOW JOB command • 6-15, B-1
SHOW LIBRARY command • 3-8, 3-13, 3-15
SHOW PROGRAM command • 3-7, 3-8, 3-15
SHOW SOURCE command • 3-20
SHOW SYSTEM command • 6-15, B-1
SHOW TASK command • 6-26
SHOW TIME command • B-1
SHOW WATCH command • 6-23
SIGNAL_AREA procedure • 9-2, A-170
SIGNAL_DEVICE procedure • 9-3, A-171
SIGNAL_EVENT procedure • 9-4, A-172
SIGNAL_PROCESS procedure • A-173
SIGNAL_SEMAPHORE procedure • 9-9, A-174
SPAWN command • 3-16
Specifications
  Ada • 3-2, 3-7, 3-11, 3-16
    see also Library specifications
  VAX/VMS file • 3-10
Specifications, library
  source file naming conventions for • 3-4
Stacks
  see also Overflow
  fixed size • 1-5
  kernel • 4-17
  main task • 8-3, 8-5, 8-6, C-2
  system interrupt • 4-11
  task • 8-2, 8-3, 8-5, 8-7
  user • 4-17
Stack storage • 6-25
  specifying • 1-5
Standard files • 7-8
Storage
  see also Tasks
  see Memory
  allocated for task activation • 8-5
  allocated for task object • 8-4
  allocated for tasks • 8-3
  controlling for main task • 8-6
  deallocated for task object • 8-4
Storage (cont'd.)
  default task stack size • 8-5
  estimating for Ada tasking programs • 8-7
STORAGE_ERROR exception
  raised for task stack overflow • 8-6, 8-8
STORAGE_SIZE representation clause • 8-6, 8-7
String descriptors • 9-12
Strings
  fixed-length • 9-10
  using with VAXELN services • 9-10
  varying-length • 9-10
Sublibraries • 3-10
  see also Program libraries
  ACS commands for managing • 3-13
  commands for setting up • 3-12
  contents of • 3-10
  determining the parents of • 3-13
  difference from program libraries • 3-11
  files associated with • 3-10
  scope conventions for • 3-11
  updating • 3-6, 3-11
  using to program for more than one target • 3-16
  value of SYSTEM.SYSTEM_NAME for • 3-8
Subprocess • 6-3
Subprograms • 3-2, 8-2
  use of MAIN_STORAGE pragma in • C-2
Subunits • 3-2, 3-11
  Ada rules for naming • 3-4
  and execution closure • 3-7
  as secondary units • 3-2
  compilation unit dependences among • 3-3
  effects of compilation order on • 3-6
  obsolete • 3-3
  order-of-compilation rules for • 3-5
  source file naming conventions for • 3-5
SUSPEND procedure • 9-8, A-176
Synchronization
  Ada task • 8-2
SYSGEN CONFIGURE command
  for determining device register address, device vector address • 4-20
SYSGEN Utility • 4-20
SYSTEM.NAME
  see NAME type
SYSTEM_NAME constant
  and AST_ENTRY pragma and attribute • 1-4

SYSTEM_NAME constant (cont'd.)
  and pragma TIME_SLICE • 1-4
  default value of • C-2
  dependence of input-output packages on • 1-5
  possible values of • 1-5
SYSTEM_NAME constant (in package SYSTEM) • 3-7
  default value of • 3-8
  determining value of • 3-8
  effect on compilation unit dependences • 3-3
  effect on exporting • 3-29
  effect on exporting Ada units • 3-28
  effect on linking • 3-24
  permanently setting the value of • 3-8
  temporarily setting the value of • 3-8
SYSTEM_NAME pragma • 3-8, C-2
System Builder editing functions • 4-5
System Builder menus • 4-4
  see also individual menus by name
System Characteristics menu • 8-4
System Control Block • 4-19
System image
  specifying for an executable system • 4-7
System name
  see SYSTEM_NAME
SYSTEM package
  additions to for VAX Ada V1.1 and later • C-2
  default value of SYSTEM_NAME constant in • C-2
  implicit recompilation of • 3-9
  NAME enumeration literals in • C-2
Systems (VAXELN)
  booting and running • 5-1
  boot method required for downline loading • 5-7
  building • 4-1, 4-7
  building with debugger • 4-7
  determining characteristics of • 4-7
  network service requirements for • 5-3
  obtaining map listing for • 4-3
  providing instruction emulation for • 4-8
  recovering after a machine failure • 5-10
  specifying authorization service for • 4-14
  specifying autoload of device driver for • 4-20
  specifying boot method for • 4-9
    see also Booting systems
  specifying circuit connect time for • 4-12

Systems (VAXELN) (cont'd.)
  specifying console terminal control character recognition for • 4-25
  specifying console terminal echo for • 4-25
  specifying console terminal for • 4-8
  specifying console terminal output display for • 4-25
  specifying console terminal recognition of escape sequences for • 4-25
  specifying console terminal representation of ASCII characters for • 4-25
  specifying device controller name for • 4-19
  specifying device interrupt priority for • 4-20
  specifying device register address for • 4-19
  specifying device vector address for • 4-19
  specifying disk and volume names for • 4-9
  specifying dynamic program space for • 4-11
  specifying File Access Listener for • 4-13
  specifying interpretation of control characters for • 4-24
  specifying interrupt service communication regions for • 4-11
  specifying interrupt stack size for • 4-11
  specifying interrupt time interval for • 4-12
  specifying message ports for • 4-10
  specifying message protocol for • 4-24
  specifying network device for • 4-13
  specifying network name server for • 4-13
  specifying network node address for • 4-14
  specifying network node name for • 4-13
  specifying network segment size for • 4-15
  specifying network service for • 4-13
  specifying object pool size for • 4-10
  specifying page table slots for • 4-10
  specifying physical memory for • 4-12
  specifying shareable images for • 4-9
  specifying terminal baud rate for • 4-23
  specifying terminal controller type for • 4-23
  specifying terminal echo for • 4-24
  specifying terminal output display for • 4-23
  specifying terminal parity checking for • 4-23
  specifying terminal recognition of escape sequences for • 4-24
  specifying terminal representation of ASCII characters for • 4-24
  specifying terminals for • 4-23
  specifying triggering for • 4-14

Systems (VAXELN)
  specifying triggering for (cont'd.)
    see also Triggering
  specifying use of modem for • 4-24
  specifying user identification code for • 4-14
  specifying virtual memory size for • 4-11
System session
  exiting • 6-14

T

Tape files • 7-3
Tapes
  booting systems from • 5-1
  copying system images to • 5-2
Tape utility procedures • 9-18
Target machine
  see also Downline loading
  see also SYSTEM_NAME constant
  adding to host node database during downline loading • 5-5
  booting systems onto • 5-1
  configuring bootstrap loader for • 5-6
  connecting to host during downline loading • 5-5
  downline loading • 5-3, 5-7
  installing communication hardware on • 5-4
  preparing for downline loading • 5-4
  reloading • 5-8, 5-10
  triggering • 5-8
Target systems • 1-2
Task
  in the VAXELN environment • 6-3
TASK_STORAGE pragma • 8-6, 8-7
Task control block • 8-4
  see also Tasks
  estimating size of • 8-4
Tasking
  see also Tasks
  differences from VAX Ada • 8-10
  implementation of VAXELN Ada • 8-1
  monitoring performance • 6-25
Task priority • 8-8, 8-9
Tasks
  activation of • 8-2
  as program units • 3-2
  as VAXELN processes • 8-3

Tasks (cont'd.)
  calling VAXELN kernel services and utility routines from • 8-10
  controlling size of stacks for • 8-7
  controlling size of stack top guard area for • 8-7
  default priority for • 8-9
  default stack allocation for • 8-5
  default top guard area for • 8-5
  default working area for • 8-5
  effect of VAXELN objects on • 8-10
  estimating size of task control block for • 8-4
  estimating size of working area for • 8-7
  implementation of • 8-1, 8-2
  improving responsiveness of • 8-11
  maximum number of • 8-4
  measuring and tuning performance of • 8-10
  range of priorities for • 4-17, 8-9
  relationship to main program • 8-1
  relationship to VAXELN processes • 8-2, 8-9
  separation compilation of • 3-2
  stack allocation for • 8-3
  stack overflow in • 8-8
  storage allocation for • 8-3
  switching among • 8-8
  synchronization of • 8-2
  task control blocks for • 8-4
  task stacks for • 8-5
  top guard area for • 8-5, C-1
  VAX/VMS and VAXELN equivalents for • 8-2
  working storage for • 8-5, C-1
Terminal controller type
  specifying for an executable system • 4-23
Terminal echo
  specifying for a system console terminal • 4-25
  specifying for a system terminal • 4-24
Terminal name
  specifying for an executable system • 4-23
Terminals • 7-4
  describing for an executable system • 4-22
TEXT_IO package • 7-1
TIME_SLICE pragma • 1-2, 1-4, 3-8, 3-10, 3-16, 8-9
Time-representation procedures • 9-21
Time slicing • 1-4, 8-11
  see also TIME_SLICE pragma
Top guard area
  see Tasks

Tracepoint • 6-21
TRANSLATE_NAME procedure • 9-6, 9-20, A-177
Triggering
  during downline loading • 5-9
  during system reloading • 5-8
Triggers (downline load) • 5-8
  specifying for an executable system • 4-14

U

UNIBUS
  common device names for • 4-21
  determining register and vector addresses for • 4-20
  Ethernet network device for • 4-13
  vector address for • 4-19
UNIBUS_MAP procedure • 9-16, A-179
UNIBUS_UNMAP procedure • 9-16, A-181
Units, compilation • 3-2
  Ada rules for naming • 3-4
  compilation closure of • 3-6
  dependences affected by context clauses • 3-3
  dependences affected by SYSTEM.SYSTEM_NAME • 3-3
  dependences among • 3-3, 3-6
  difference from source files • 3-4
  effects of dependences on compiling • 3-5
  execution closure of • 3-7
  obsolete • 3-4, 3-6, 3-9
  source file naming conventions for • 3-4
  target-related dependences among • 3-7, 3-8
Units, generic • 3-2
Units, library • 3-2
  Ada rules for naming • 3-4
  compilation unit dependences among • 3-3
  elaboration of • 3-26, 3-28
  exporting • 3-27
Units, obsolete • 3-3, 3-4, 3-6, 3-9, 3-20
Units, secondary • 3-2
UNLOAD_PROGRAM procedure • 9-21, A-182
UNLOCK_MUTEX procedure • 9-6, A-183
UNWIND procedure • 9-16
USE_ERROR
  raised for FDL errors in FORM parameter • 7-9
User stack
  specifying the size of for an executable system • 4-17

V

Variable
  statically allocated • 6-23
VAX Ada
  accounting for differences from VAXELN Ada • 3-7
  VAXELN Ada differences from • 1-2, 8-2, 8-9, 8-10, 8-11
VAX Ada compiler • 2-2
VAXELN_CIRCUIT attribute • 7-12
VAXELN_SERVICES package • 3-8, 3-16, 4-10, 9-1
VAXELN circuits • 7-5
VAXELN objects • 9-1
VAXELN Remote Debugger
  see Remote Debugger
VAXELN services • 9-12 to 9-21
  calling • 9-1
  calling from tasks • 8-10
VAXELN system
  monitoring • 6-15
VAXELN System Builder • 2-3
  see System Builder
VAXELN system characteristics • 6-7
VAXELN types • 9-9
VAX/VMS services • 9-22
VERIFY command • 3-15
Virtual memory size
  effect on tasking • 8-4
  specifying for an executable system • 4-11
Virtual-to-physical-address procedure • 9-21

W

WAIT_ALL procedure • 9-2, 9-3, 9-4, 9-7, 9-9, A-184
WAIT_ANY procedure • 9-2, 9-3, 9-4, 9-7, 9-9, A-184
Watchpoint • 6-21
Working storage area
  see Tasks
WRITE_REGISTER procedure • 1-5, C-2
WRITE_REGISTER procedures • C-3