Getting Started with VAXlab

Order Number: AA-KN96B-TE

August 1988

This document describes the VAXlab system and the VAXlab Software Library. It provides a conceptual overview of VAXlab, describes the utility for performing system management tasks, and presents programming language-specific information helpful when developing application programs using the VAXlab Software Library.

Revision/Update Information: This is a revised document.

Operating System and Version: VMS Version 5.0

Software Version: VAXlab Software Library Version 1.3

digital equipment corporation
maynard, massachusetts

First Printing, December 1987
Revised, August 1988

The Reader's Comments form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	MicroVAX	VAXstation
DECnet	Q-bus	VMS
DRB32	VAX	VT
LN03	VAXcluster	
LN03 Plus	VAX GKS	
LN03R	VAXlab	

digital™

This document was prepared using VAX DOCUMENT, Version 1.0.

Contents

TABLES

Preface

Intended Audience

Getting Started with VAXlab is intended for scientists and engineers who are unfamiliar with VAXlab, and who may or may not be familiar with VMS system management tasks and procedures.

This document assumes a basic understanding of computer concepts and a working knowledge of at least one high-level programming language.

Document Structure

Getting Started with VAXlab describes the VAXlab system and the VAXlab Software Library. It provides a conceptual overview of VAXlab, describes the utility for performing system management tasks, and presents language-specific information helpful when developing application programs using the VAXlab Software Library.

The document is divided into three chapters:

Chapter Number	Contents
Chapter 1	Describes the components of the VAXlab system and the capabilities of the VAXlab Software Library.
Chapter 2	Explains the Manager Utility and the VAXlab system management tasks you can perform using this utility.
Chapter 3	Describes how to develop programs using the VAXlab Software Library, and provides information specific to programming languages you need to consider when writing VAXlab application programs.

Associated Documents

In addition to this guide, the VAXlab documentation set includes the following guides:

- The *VAXlab Master Index* contains index entries from all documents in the VAXlab documentation set.
- The *VAXlab Installation Guide* details how to install the VAXlab software.
- The *Guide to the VAXlab Laboratory I/O Routines* describes how to initiate, set up, control, and terminate I/O to and from VAXlab I/O devices.
- The *Guide to the VAXlab Interactive Data Acquisition Tool* describes how to communicate with VAXlab through the Interactive Data Acquisition Tool (IDAT) to establish parameters for data acquisition and to initiate, control, obtain, analyze, and plot real-time data.
- The *Guide to the VAXlab Laboratory Graphics Package* describes how to specify plotting attributes, and how to plot real-time data or data produced by calculations in two dimensions, three dimensions, and two-dimensional contours from a three-dimensional view.
- The *Guide to the VAXlab Signal-Processing Routines* describes how to use the signal-processing routines to perform Fourier transforms, correlation functions, data filtering, and spectral windowing.

The following is a list of associated software documents that you should reference for additional information about programming concepts and techniques not covered in this guide.

- The *Laboratory Interfacing Handbook* presents detailed descriptions of laboratory I/O concepts. If you are unfamiliar with laboratory data acquistion and control techniques, such as instruments, signals, and interfaces, or if you require additional information about computers, I/O hardware, or applications, read this handbook before you begin using the VAXlab system.
- The *VAX Realtime User's Guide* describes those features of VAX systems which pertain to real-time applications in scientific and industrial settings. If you are unfamiliar with VAX systems, read this guide before you begin using the VAXlab system.

Conventions

Getting Started with VAXlab uses the following documentation conventions:

Convention	Meaning
Italics	Words or phrases appearing in *italics* indicate referencing of an associated document.
Bold	A boldface word or phrase indicates emphasis on an important concept or word, or indicates a subroutine argument appearing in text.
RETURN	Press the key labeled **Return** on the terminal keyboard.
Ellipses	Vertical ellipses indicate that portions of a display or programming example are excluded for presentation purposes.
"Double quotes"	Double quotes enclose screen prompts appearing in text.
0	Press the key labeled **0** on the auxiliary keypad on the terminal keyboard.

Chapter 1

Overview of the VAXlab System and the VAXlab Software Library

This chapter describes the VAXlab system components and presents an overview of the VAXlab Software Library (VSL).

1.1 Overview of the VAXlab System

The VAXlab system is a combination of hardware and software components that create the environment that the VAXlab Software Library requires, and that includes the VAXlab Software Library. The VAXlab hardware components are described in Section 1.2, VAX Hardware Components. The VAXlab software components are described in Section 1.3, VAX Software Components.

The core of the VAXlab system is the VAXlab Software Library (VSL), which consists of:

- Libraries of subroutines you can use to perform real-time I/O, mathematical and statistical analysis, signal-processing, peak-processing, and plotting operations.
- Interactive, menu-driven utilities through which you can perform system management operations and access the interactive data acquisition tool to acquire, process, store, retrieve, output, and plot data.

Using the VAXlab system, you can:

- Control the real-time I/O devices, which consist of analog-to-digital converters, digital-to-analog converters, parallel boards, clocks, disk files, and virtual memory
- Perform digital filtering of data and signal-processing operations, such as fast Fourier tranforms (FFTs)
- Perform mathematical and statistical analysis of data
- Perform peak-processing of data
- Produce multidimensional graphical representations of data
- Perform system management tasks

1.2 VAXlab Hardware Components

The VAXlab system can run on any of the following processors:

- MicroVAX II
- MicroVAX 2000-series
- MicroVAX 3000-series
- VAX 6000-series
- VAX 8000-series
- VAXstations (except for the VAXstation I)

The basic system hardware includes:

- One RD53-A or RA-series disk
- One TK50 or TK70 cartridge tape drive
- Five MB RAM
- One console graphics terminal
- One KWV11-C real-time clock board
- One Universal Data Interface Panel, UDIP-KA[1]

[1] In the basic configuration, this panel is connected to the real-time clock board. With this panel, you can access the clock without opening the system cabinet.

The optional hardware consists of:

- Analog options:
  - AAV11-D, a digital-to-analog converter, using a UDIP-DA
  - ADQ32, a high-speed, analog-to-digital converter with an on-board clock
  - ADV11-D, an analog-to-digital converter, using a UDIP-AA
  - AXV11-C, an analog I/O board, using a UDIP-AX
  - Preston GMAD-series, high-speed analog-to-digital converters
- Digital options:
  - DRB32, a 32-bit, DMA parallel I/O port for the VAXBI bus
  - DRB32W, a DR11W-compatible port for the VAXBI bus
  - DRQ3B, a high-speed, 16-bit DMA parallel I/O board
  - DRV11-J, a 16-bit parallel I/O board
  - DRV11-WA, a 16-bit DMA parallel I/O board
- IEQ11-A, the IEEE-488 bus controller
- Plotter, printer, and terminal options:
  - HP7550™ pen plotter
  - LA12, LA34, LA50, LA75, LA100, and LA210 line printers
  - LCP01 ink jet plotter
  - LN03 PLUS laser printer
  - LN03R laser printer
  - LSP40 laser printer
  - LVP16 pen plotters
  - TEKTRONIX™ 4014 and TEKTRONIX 4107
  - VAXstation II and VAXstation II/GPX workstations
  - VT125, VT240, VT241, VT330, and VT340 terminals

™ HP7550 is a registered trademark of Hewlett Packard.
™ TEKTRONIX is a registered trademark of TEKTRONIX, Inc.

- Serial line options:
  - DH11, a 16-line asynchronous serial terminal multiplexer
  - DHV11, an 8-line asynchronous serial terminal multiplexer
  - DMB32, an 8-line asynchronous serial terminal multiplexer
  - DMF32, an 8-line asynchronous serial terminal multiplexer
  - DS100, an 8-line asynchronous serial terminal multiplexer
  - DS200, an 8-line asynchronous serial terminal multiplexer
  - DZ11, an 8-line asynchronous serial terminal multiplexer
  - DZQ11, a 4-line asynchronous serial terminal multiplexer
  - DZV11, a 4-line asynchronous serial terminal multiplexer
  - Serial lines on a MicroVAX 2000
  - Any other serial line device supported by the VMS terminal driver
- Additional memory

When you want to connect your laboratory devices to the system, the connection information you need varies, depending on how you connect the devices. If you plan to use the Universal Data Interface Panel (UDIP), you can connect and disconnect the laboratory devices without having to open up the system cabinet. The *Universal Data Interface Panel Reference Card* provides diagrams of the panel connectors to the real-time modules so you can see how to connect to the panel.

If you plan to connect the laboratory devices by using the distribution panel connector kit for the appropriate module, the *VAXlab Hardware Information Kit* contains information for making connections to the modules. The connector kits are installed in the I/O distribution panel in the rear of the system cabinet. They provide a 25-pin D-connector for easy access to the real-time modules.

1.3 VAXlab Software Components

The VAXlab software runs layered on the VMS operating system. It consists of:

- VAX GKS, the graphics software
- VAXlab Software Library (VSL) application software
- For VAXstations only: the standard VAXstation software (VWS)

The next sections describe several of these software components in detail.

1.3.1 VAX GKS Graphics Software

The VAX GKS graphics software is DIGITAL's implementation of the Graphical Kernel System (GKS). VAX GKS provides an interface between an application program and conventional hardware graphics. The VAXlab Laboratory Graphics Package (LGP) is a set of subroutines that use the VAX GKS software to plot laboratory data in the form of charts, graphs, and histograms. You can also use VAX GKS routines directly. For specific information about VAX GKS, see the *VAX GKS Reference Manual,* Volumes I and II.

1.3.2 VAXlab Software Library

The VAXlab Software Library consists of the following sets of routines and utilities:

- The **Manager Utility**, a menu-driven, interactive system management tool you can use to perform system management tasks, such as adding new accounts or copying files to permanent offline storage media, by selecting options from a menu. The Manager Utility is described in Chapter 2, VAX System Management, of this document.
- The **Laboratory I/O Routines (LIO)**, a set of routine calls that can communicate with real-time input/output devices. The routines are described in the *Guide to the VAXlab Laboratory I/O Routines*.

- The **Interactive Data Acquisition Tool (IDAT)**, a simple menu-driven utility you can use to acquire, store, retrieve, process, and plot data. This utility is described in the *Guide to the VAXlab Interactive Data Acquisition Tool.*
- The **Laboratory Signal-Processing Routines (LSP)**, a set of routine calls that include Fourier transform routines derived from the fast Fourier transform (FFT) algorithm, a histogram routine for multi-channel analysis, routines for four types of digital filters, spectral-windowing routines, power spectra routines, cross-correlation and auto-correlation routines, and phase angle-amplitude routines. The routines are described in the *Guide to the VAXlab Signal-Processing Routines.*
- The **Mathematics and Statistics Routines**, a set of routine calls for performing calculations. The routines are described in Appendix A, Mathematics and Statistics Routines, of the *Guide to the VAXlab Signal-Processing Routines.*
- The **Peak-Processing Routine**, one routine you can use to perform peak analysis of data. This routine is described in Appendix B, The Peak-Processing Routine, of the *Guide to the VAXlab Signal-Processing Routines.*
- The **Laboratory Graphics Package (LGP)**, a set of routine calls that can plot both real-time data and data produced by calculations. Using the graphics routines, you can produce two- and three-dimensional plots, as well as contour plots from a three-dimensional view. The graphics routines are described in the *Guide to the VAXlab Laboratory Graphics Package.*

Chapter 2

VAXlab System Management

Managing the VAXlab system includes planning, maintaining, and monitoring system performance. This chapter provides an overview of topics to consider during the planning phase, lists the maintenance and monitoring tasks, and describes how to use the Manager Utility to accomplish these tasks.

2.1 Planning Your VAXlab System

This section presents a summary of topics to consider as you plan to integrate the VAXlab system into your laboratory. Taking time to write down your plans and goals for the following topics can help to ensure the smooth integration of the VAXlab system into your overall laboratory operations.

Carefully consider the following topics as you plan your VAXlab system:

- The specific purpose and goals of the system.
- The devices to which VAXlab will be connected.
- Networking requirements, and the security of connections made through DECnet or dialup service.
- The expected growth of the system in terms of users, devices, and number of nodes on the network.
- The access protections for system data.
- The procedures for protecting system data by periodic backups, offline storage, and emergency procedures.

- The physical security of the processors.
- In multiuser environments, the number of users; their tasks and system requirements; their level of understanding; the configuration of their accounts; the information they need, such as names for printers and devices; and where they are physically located.

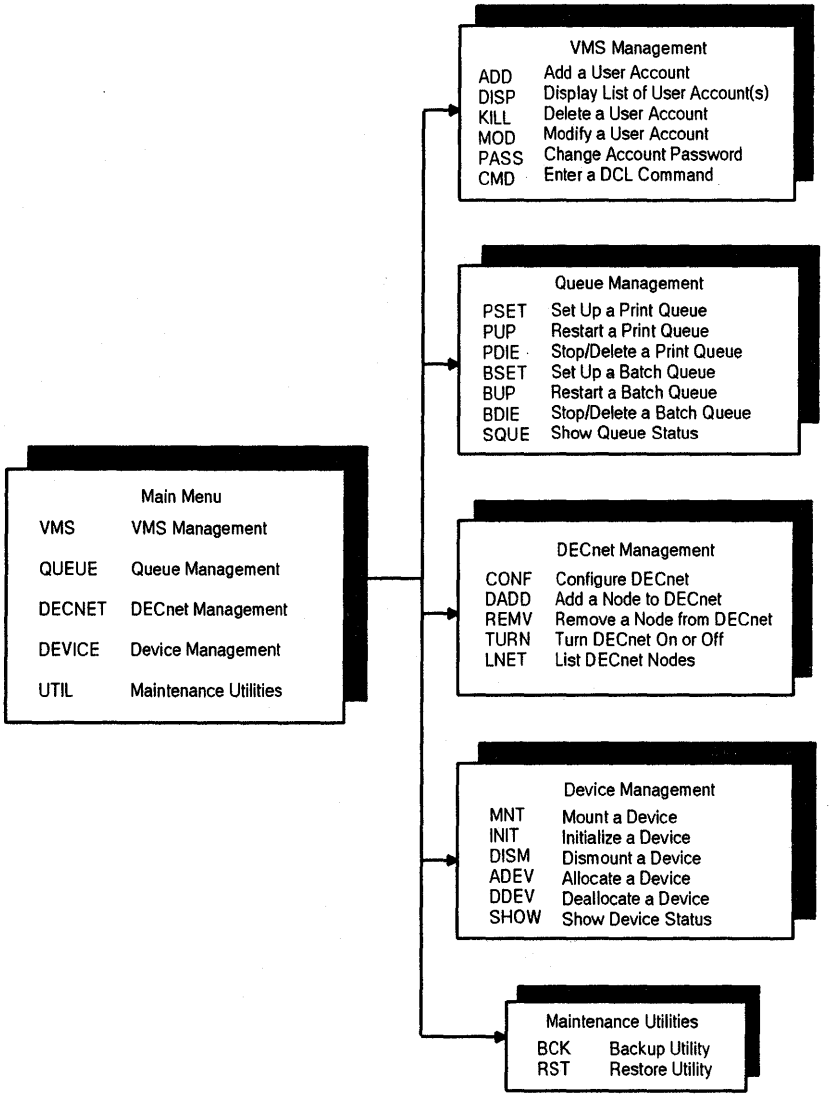
Once you have planned your VAXlab system, you are ready to begin using it.

2.2 Overview of the Manager Utility

The Manager Utility provides an interactive, menu-driven interface to many of the DCL commands normally used to perform system management tasks. The utility is designed to assist users who are not familiar with system management tasks in identifying and performing these tasks easily.

The Manager Utility consists of **menu screens** and **data entry screens.** Manager Utility menu screens consist of options, or choices, from which you select one option. The MANAGER Main Menu displays a list of the five main categories or types of system management you can perform using this utility. The Manager Utility also contains a menu for each of the main categories of system management. These menus list the individual system management tasks associated with each main category. Figure 2–1 lists the MANAGER menus and the individual system management tasks associated with each menu.

Figure 2–1: MANAGER Menus



You perform the individual system management tasks through data entry screens. Data entry screens consist of **prompts** and **data entry fields**. A prompt is a label that describes the type of information you enter in the data entry field after the prompt. Some data entry fields contain default values supplied by MANAGER that are used if you do not supply values in place of the default values. The default values satisfy most situations.

Be careful when you change default values, because in some cases you can adversely affect the performance of the system. For that reason, although any user account can access MANAGER, only SYSTEM or privileged accounts can perform certain tasks. **Be sure to log in to a SYSTEM or privileged account before you run the Manager Utility.** In addition, limit SYSTEM or privileged accounts to the user or users responsible for system management.

2.3 Running MANAGER

To run the Manager Utility, enter the following command after the DCL ($) prompt:

```
$ MANAGER RETURN
```

The Manager Utility displays the MANAGER Main Menu and is ready to begin accepting information. Enter the menu option you want after the "Select option:" prompt and press RETURN. You can enter any menu option, even if the option is not listed on the menu currently displayed on the terminal screen.

2.4 Using the MANAGER Function Keys

This section describes the established conventions for entering and editing data on MANAGER screens, accepting default values supplied by MANAGER, getting help information about MANAGER, and signaling system management tasks to begin.

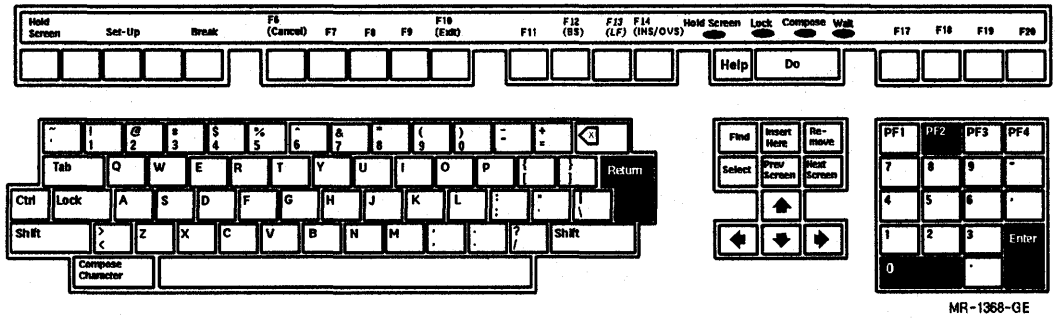
You can use the following four function keys with the Manager Utility:

Table 2–1: MANAGER Function Keys

Key	Location on the terminal keyboard
ENTER	Auxiliary keypad
0	Auxiliary keypad
PF2	Auxiliary keypad
RETURN	Main keypad

Figure 2–2 shows the layout of the LK201 keyboard with the MANAGER function keys highlighted so that you can locate them easily on your terminal keyboard.

Figure 2–2: MANAGER Function Key Layout



MR-1368-GE

When you press the RETURN key, the cursor moves from one field to the next and signals MANAGER to accept either the default value or the data you entered in the data entry field. When you enter data in a data entry field, press RETURN before you press any of the other function keys. When you press RETURN after the last data entry field on a screen, the task begins to execute.

If you are unsure about how to respond to a prompt, you can press the PF2 key to get help information. Make sure that the cursor is located in the data entry field after the prompt about which you require help information. Then, press the PF2 key, and MANAGER displays a full-screen help message about the type of information you need to enter and the appropriate format in which to enter the information.

To exit or quit a data entry screen without executing the task, you can press the 0 key. If you make an error as you enter data in a field or if you want to edit data you have already entered, quit the screen using the 0 key, and then select that screen again.

You signal the management task to begin by pressing the ENTER key when the values on a data entry screen are exactly the way you want them.

2.5 System Management Tasks

Table 2–2 lists the system management tasks associated with the VAXlab system, including the corresponding menu options you need to select to perform each task.

See the *Guide to VAX/VMS System Management and Daily Operations* for further information about the management tasks described in this chapter.

Table 2–2: System Management Tasks

Management Task	Menu Option
VMS MANAGEMENT:	
Add a user account	ADD
Display list of user account(s)	DISP
Delete a user account	KILL
Modify a user account	MOD
Change account password[1]	PASS
Enter a DCL command[2]	CMD
QUEUE MANAGEMENT:	
Set up a print queue	PSET
Restart a print queue	PUP
Stop/delete a print queue	PDIE
Set up a batch queue	BSET
Restart a batch queue	BUP
Stop/delete a batch queue	BDIE
Show queue status	SQUE
DECnet MANAGEMENT:	
Configure DECnet	CONF
Add a node to DECnet	DADD
Remove a node from DECnet	REMV
Turn DECnet on or off	TURN
List DECnet nodes	LNET

[1]The PASS option does not display a screen. Instead, it displays the prompts for the old password and the new password. After you enter the passwords, MANAGER redisplays the menu from which you called the PASS option.

[2]Use the CMD option to access the DCL prompt ($) from within the Manager Utility. After you execute a DCL command, MANAGER redisplays the menu from which you called the CMD option.

Table 2–2 (Cont.): System Management Tasks

Management Task	Menu Option
DEVICE MANAGEMENT:	
Mount a device	MNT
Initialize a device	INIT
Dismount a device	DISM
Allocate a device	ADEV
Deallocate a device	DDEV
Show device status	SHOW
MAINTENANCE UTILITIES:	
Backup Utility	BCK
Restore Utility	RST

2.6 VMS Management Tasks

The following sections describe how to perform each of the VMS management tasks.

2.6.1 Adding a User Account

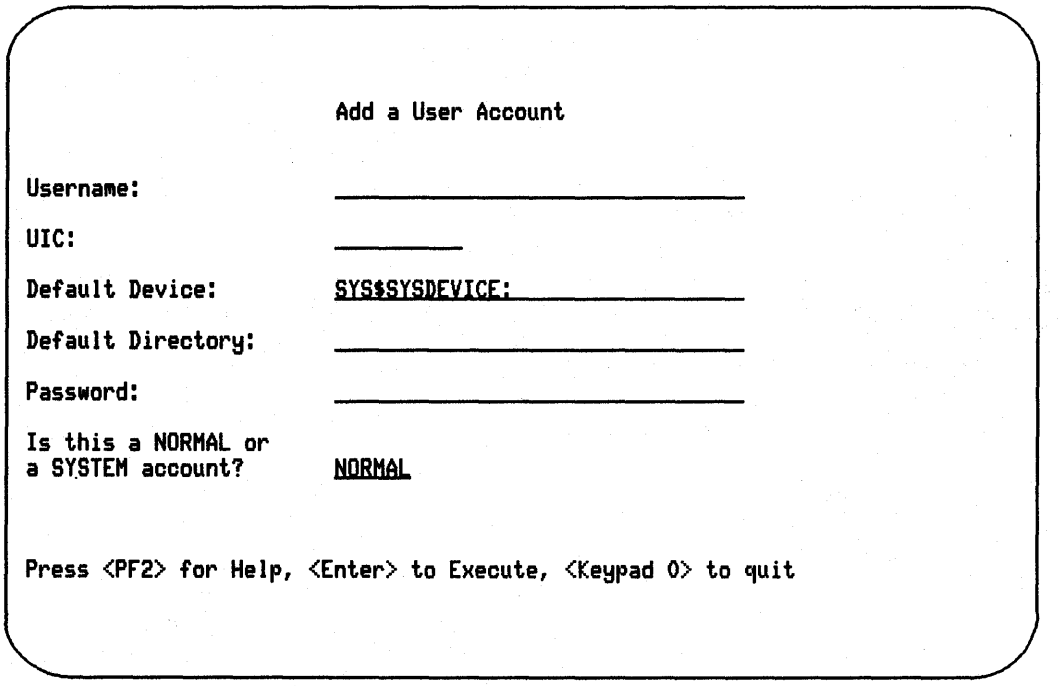
To add a user account, select the ADD option from the VMS Management menu. MANAGER displays the Add a User Account screen shown in Figure 2–3.

Use this screen from a SYSTEM account to add a user account to the system and to create the first, top-level directory for the account.

To complete the screen, supply the new account with a user name, a User Identification Code (UIC) for the user of the account, a default device and directory for the new account's files, and a password. You can accept the default values as they are displayed on the screen, or you can substitute new values. Use the default account type, NORMAL, for most accounts. **SYSTEM accounts have system-wide privileges and should be restricted to users responsible for system management.**

You can get a list of the UICs currently assigned by pressing PF2 while the cursor is at the UIC prompt. This list is the same as that displayed by the DISP screen. When you supply a UIC, supply the brackets as well, for example, [LAB1,04].

Figure 2-3: Add a User Account Screen



2.6.2 Displaying a List of User Accounts

To display a list of user accounts, select the DISP option from the VMS Management menu. MANAGER displays the Display List of User Accounts screen shown in Figure 2–4.

Use this screen from a SYSTEM account to display information about a user account.

Figure 2–4: Display List of User Accounts Screen

```
                    Display List of User Account(s)

User Name, or "*"
to list all users    _____________________________

FULL display?        N




Press <PF2> for Help, <Enter> to Execute, <Keypad 0> to quit
```

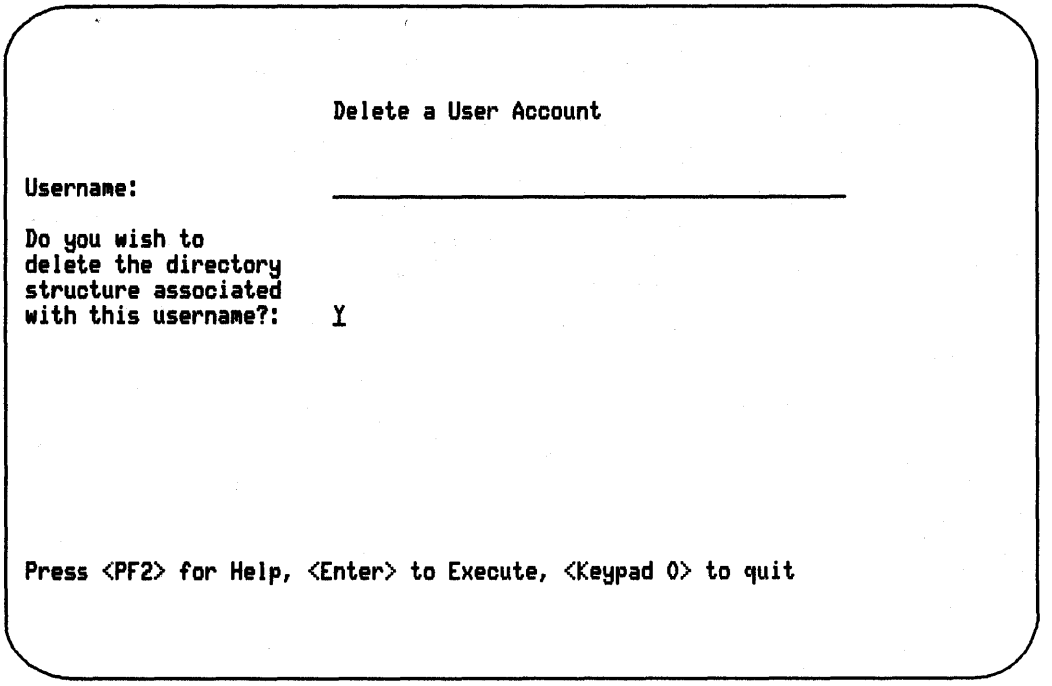
2.6.3 Deleting a User Account

To delete a user account, select the KILL option from the VMS Management menu. MANAGER displays the Delete a User Account screen shown in Figure 2–5.

Use this screen from a SYSTEM account to delete a user name from the system. You can also remove the user's directory structure.

If you answer Y (yes) to the "Do you wish to delete the directory structure associated with this username?" prompt, you delete the user name and all the user's files and directories from the system. If you answer N (no), you leave the files on the system, but disassociate the user name from them.

Figure 2–5: Delete a User Account Screen



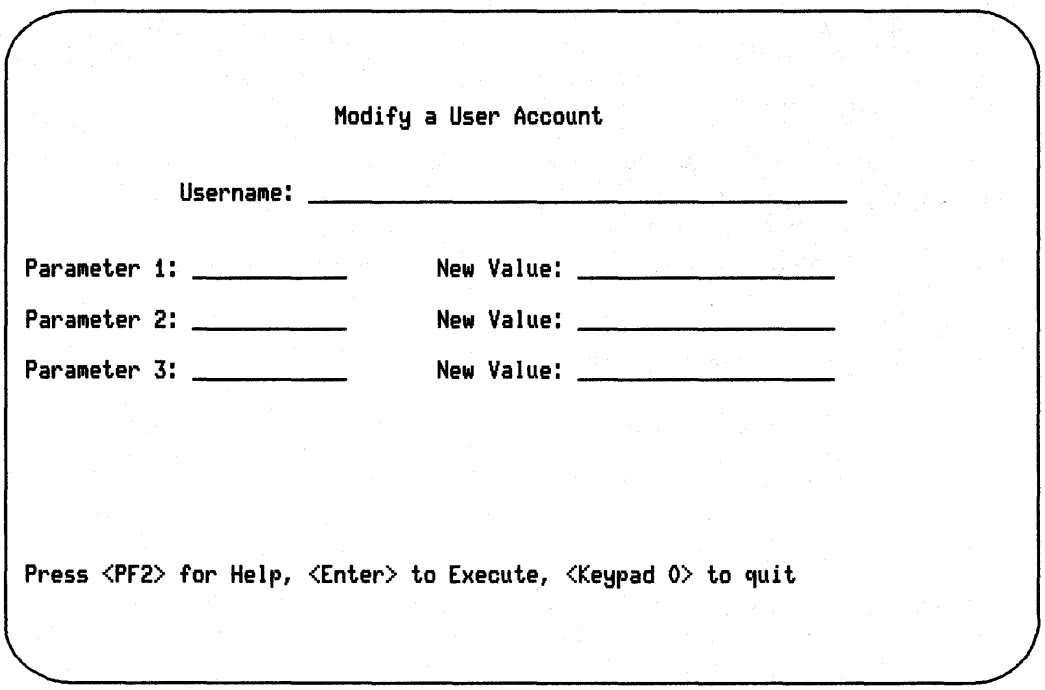
2.6.4 Modifying a User Account

To modify a user account, select the MOD option from the VMS Management menu. MANAGER displays the Modify a User Account screen shown in Figure 2–6.

Use this screen from a SYSTEM account to modify up to three parameters for an account.

For information about user-account parameters, see the system tuning information in your system-specific installation guide.

Figure 2–6: Modify a User Account Screen



2.6.5 Changing Account Passwords

To change the password of a user account, select the PASS option from the VMS Management menu. MANAGER does not display a data entry screen in this case. Instead, MANAGER displays prompts for the old password and the new password. Enter both the old and the new passwords, and press RETURN. MANAGER then redisplays the VMS Management menu.

2.6.6 Entering a DCL Command

To enter a DCL command from within the Manager Utility, select the CMD option from the VMS Management menu. MANAGER does not display a data entry screen in this case. Instead, MANAGER displays the DCL prompt. Enter the DCL command after the prompt, and press RETURN. When the command completes execution, MANAGER redisplays the VMS Management menu.

2.7 Queue Management Tasks

The following sections describe the Queue Management tasks.

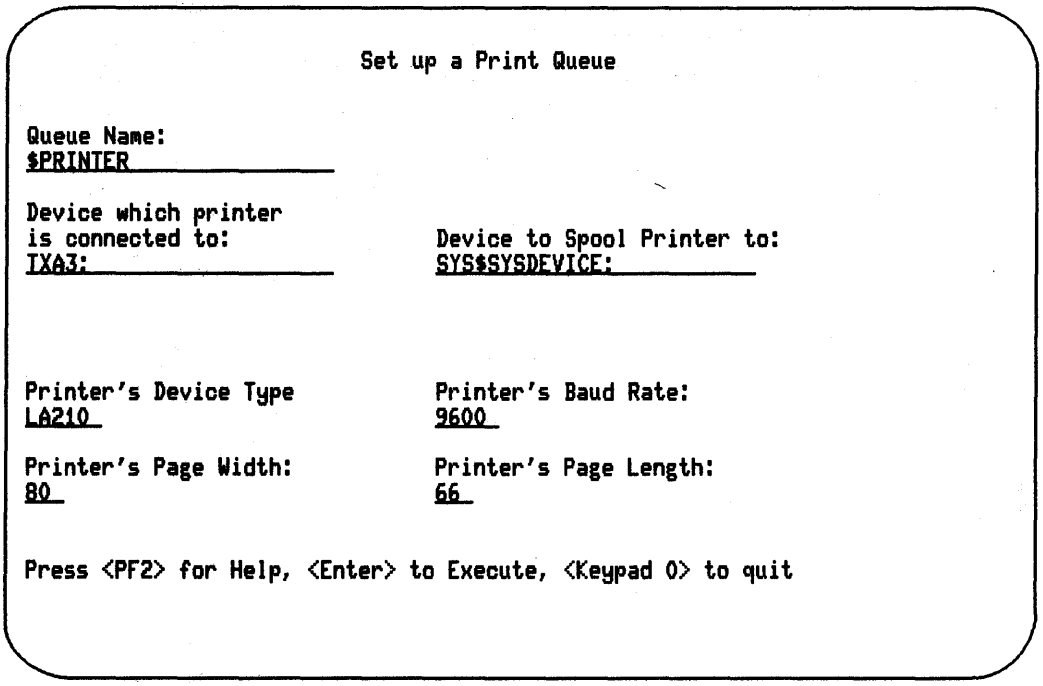
2.7.1 Setting Up a Print Queue

To set up a print queue, select the PSET option from the Queue Management menu. MANAGER displays the Set Up a Print Queue screen shown in Figure 2–7.

Use this screen from a SYSTEM account to set up the characteristics of a print queue and to start the queue.

For further information about setting up a print queue, including optional INITIALIZE/QUEUE qualifiers, see the *Guide to VAX/VMS System Management and Daily Operations*.

Figure 2–7: Set Up a Print Queue Screen



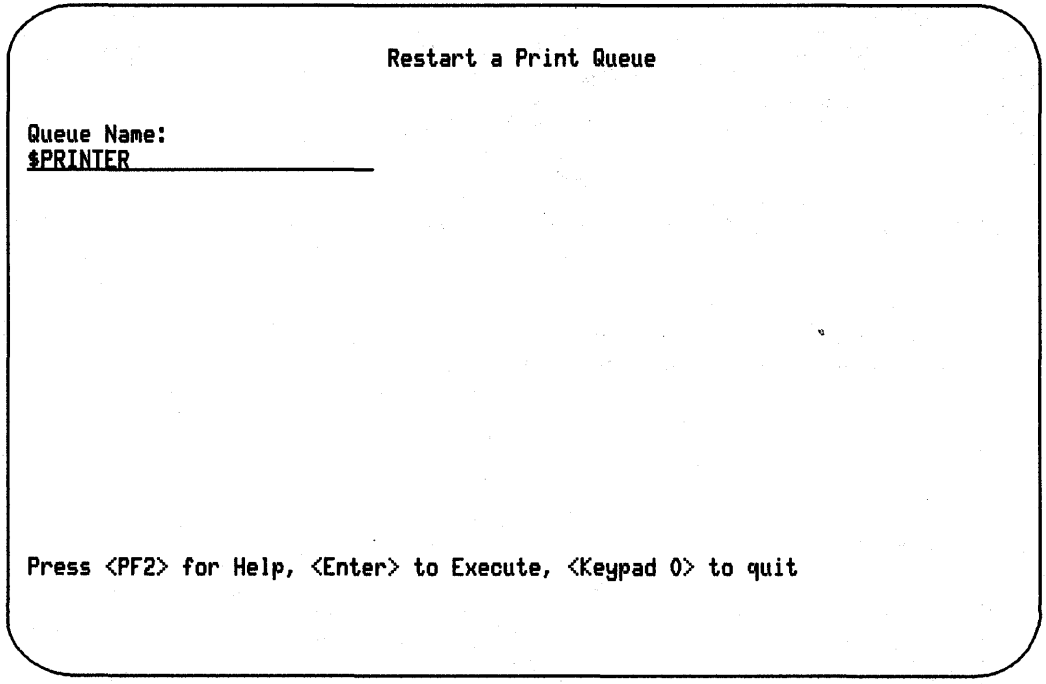
2.7.2 Restarting a Print Queue

To restart a print queue, select the PUP option from the Queue Management menu. MANAGER displays the Restart a Print Queue screen shown in Figure 2–8.

Use this screen from a SYSTEM account to restart a stalled print queue.

After the "Queue Name:" prompt, supply the logical name of the print queue.

Figure 2–8: Restart a Print Queue Screen



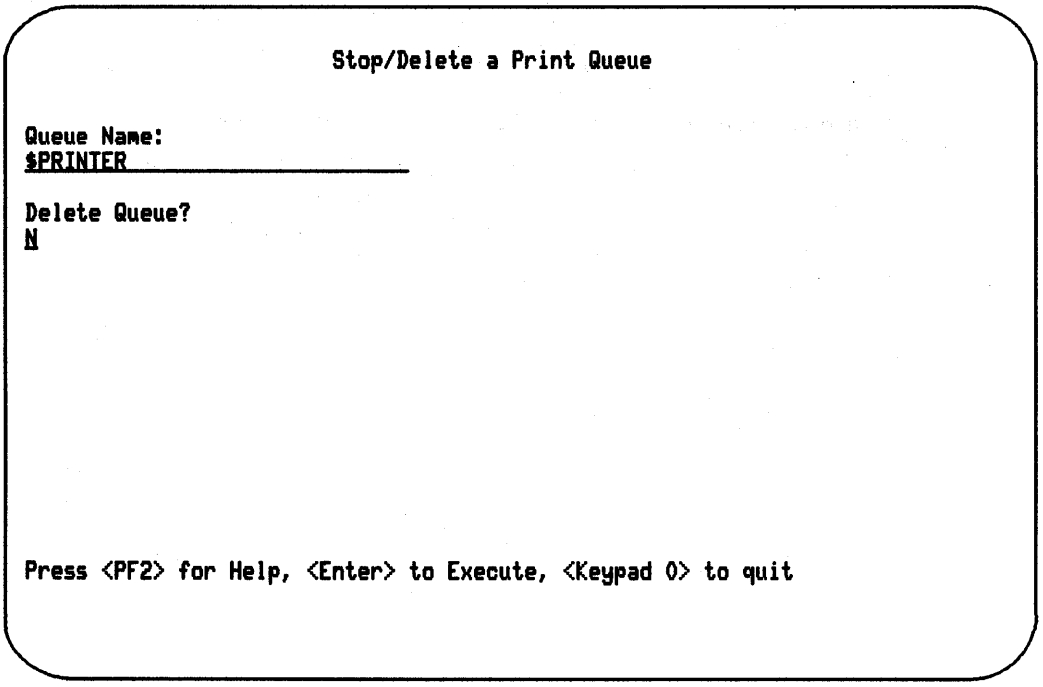
2.7.3 Stopping and Deleting a Print Queue

To stop or delete a print queue, select the PDIE option from the Queue Management menu. MANAGER displays the Stop/Delete a Print Queue screen shown in Figure 2-9.

Use this screen from a SYSTEM account to stop or delete a print queue.

After the "Queue Name:" prompt, enter the logical name of the print queue. After the "Delete Queue?" prompt, enter Y (yes) to delete the queue from the list of valid system queues. Enter N (no) to stop the queue. Stopping the queue does not delete entries from the queue and does not delete the queue from the system.

Figure 2–9: Stop/Delete a Print Queue Screen



2.7.4 Setting Up a Batch Queue

To set up a batch queue, select the BSET option from the Queue Management menu. MANAGER displays the Set Up a Batch Queue screen shown in Figure 2–10.

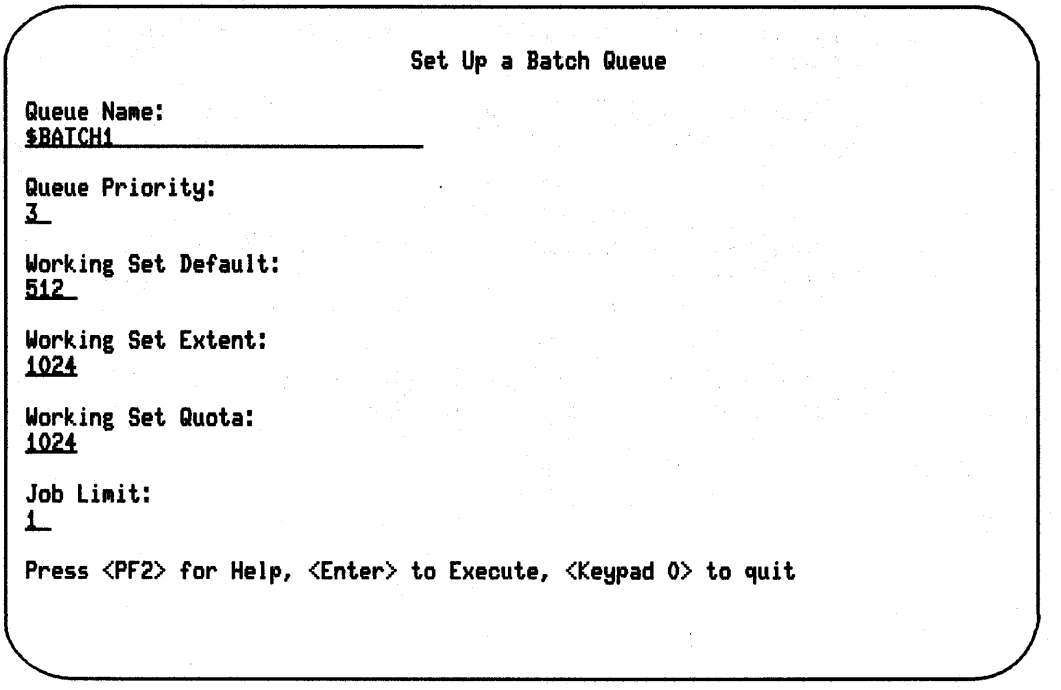
Use this screen from a SYSTEM account to set up the characteristics for a batch queue, and to start the queue.

NOTE

The screen contains default values. DO NOT change the numerical values unless you are sure of what you are doing. Changing these values can have a severe impact on system performance.

For further information about managing batch queues, see the *Guide to VAX/VMS System Management and Daily Operations*.

Figure 2-10: Set Up a Batch Queue Screen



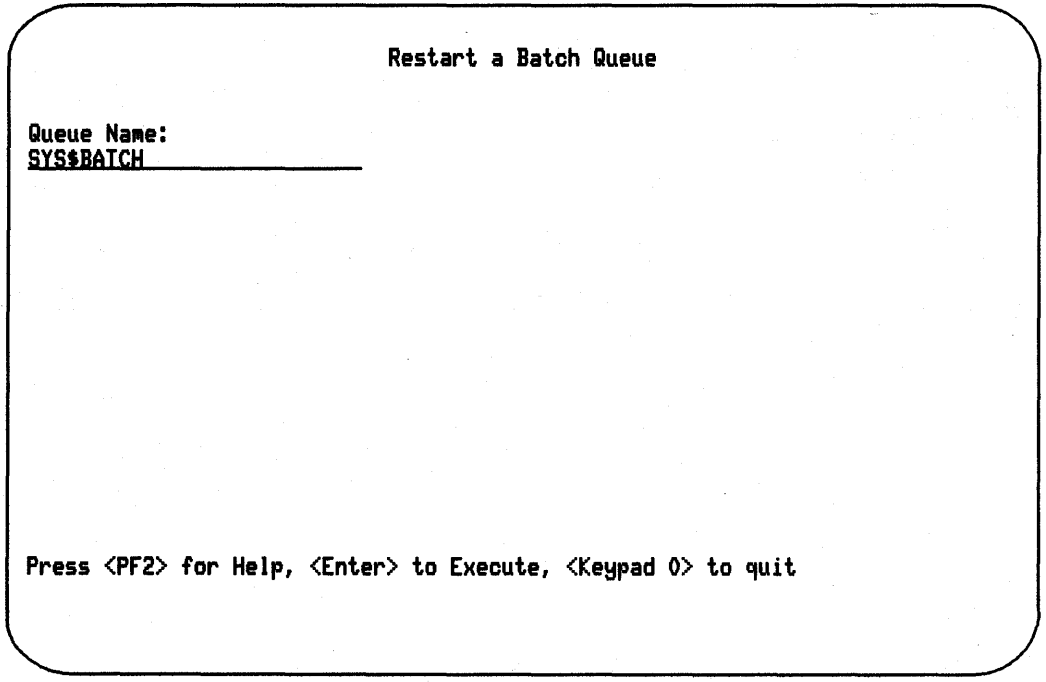
2.7.5 Restarting a Batch Queue

To restart a batch queue, select the BUP option from the Queue Management menu. MANAGER displays the Restart a Batch Queue screen shown in Figure 2–11.

Using this screen from a SYSTEM account, you can restart a stalled batch queue.

After the "Queue Name:" prompt, supply the logical name of the batch queue.

Figure 2–11: Restart a Batch Queue Screen



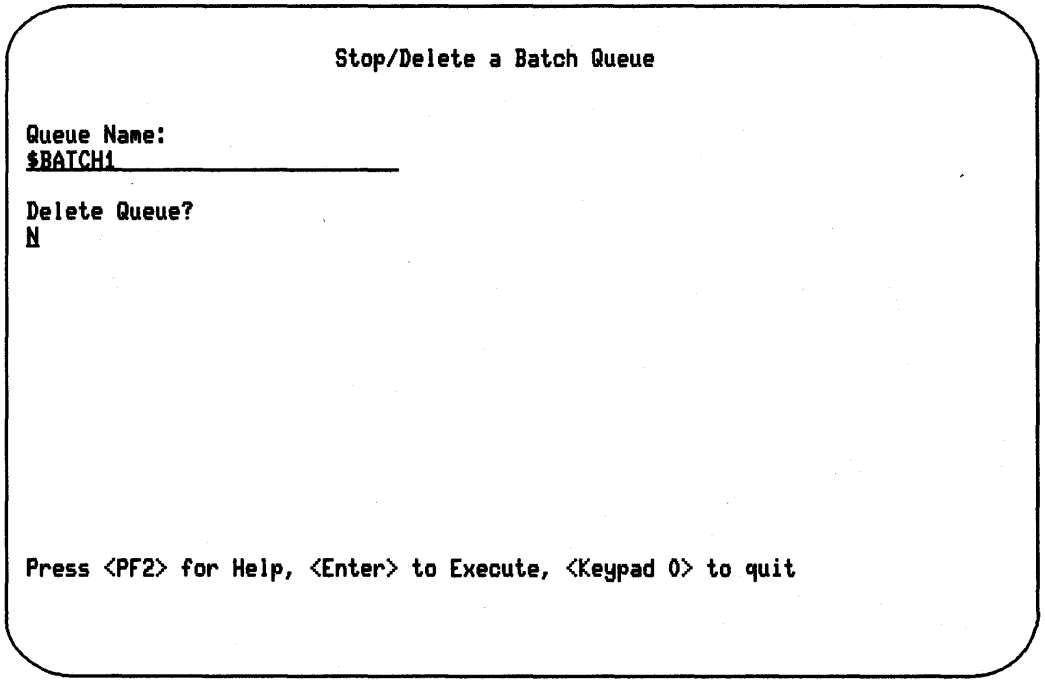
2.7.6 Stopping and Deleting a Batch Queue

To stop or delete a batch queue, select the BDIE option from the Queue Management menu. MANAGER displays the Stop/Delete a Batch Queue screen shown in Figure 2–12.

Use this screen from a SYSTEM account to stop or delete a batch queue.

After the "Queue Name:" prompt, enter the logical name of the batch queue. After the "Delete Queue?" prompt, enter Y (yes) to delete the queue from the list of valid system queues. Enter N (no) to stop the queue. Stopping the queue does not delete entries from the queue, and does not delete the queue from the system.

Figure 2–12: Stop/Delete a Batch Queue Screen

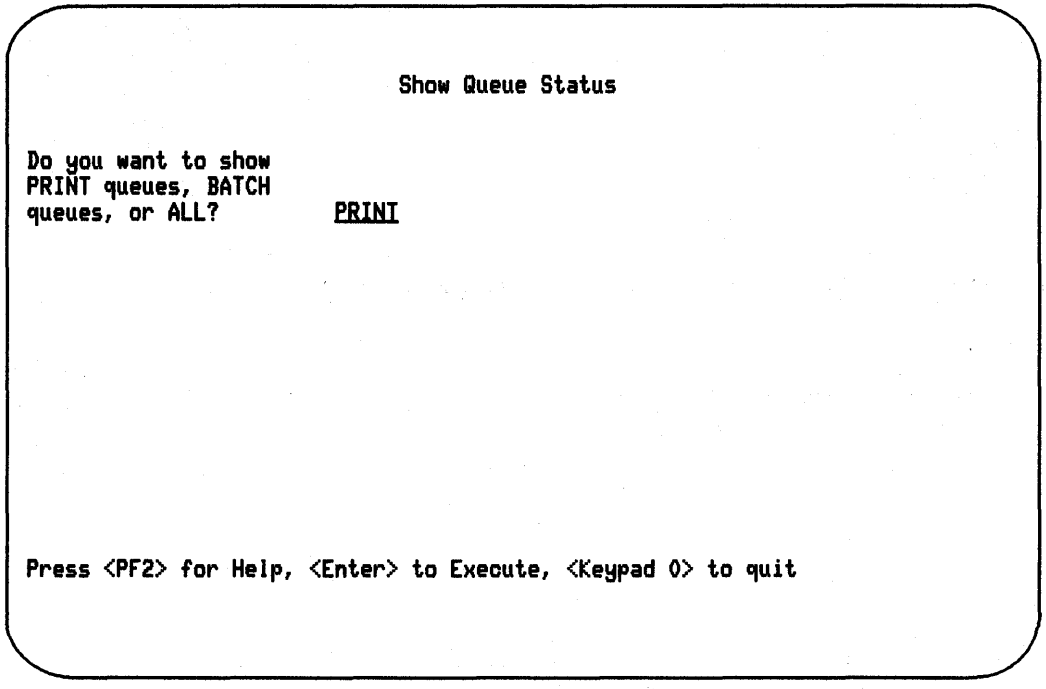


2.7.7 Showing Queue Status

To show the status of a queue, select the SQUE option from the Queue Management menu. MANAGER displays the Show Queue Status screen shown in Figure 2–13.

Use this screen to display information about a print or batch queue, or all print or batch queues.

Figure 2–13: Show Queue Status Screen



2.8 DECnet Management Tasks

The following sections describe the DECnet management tasks.

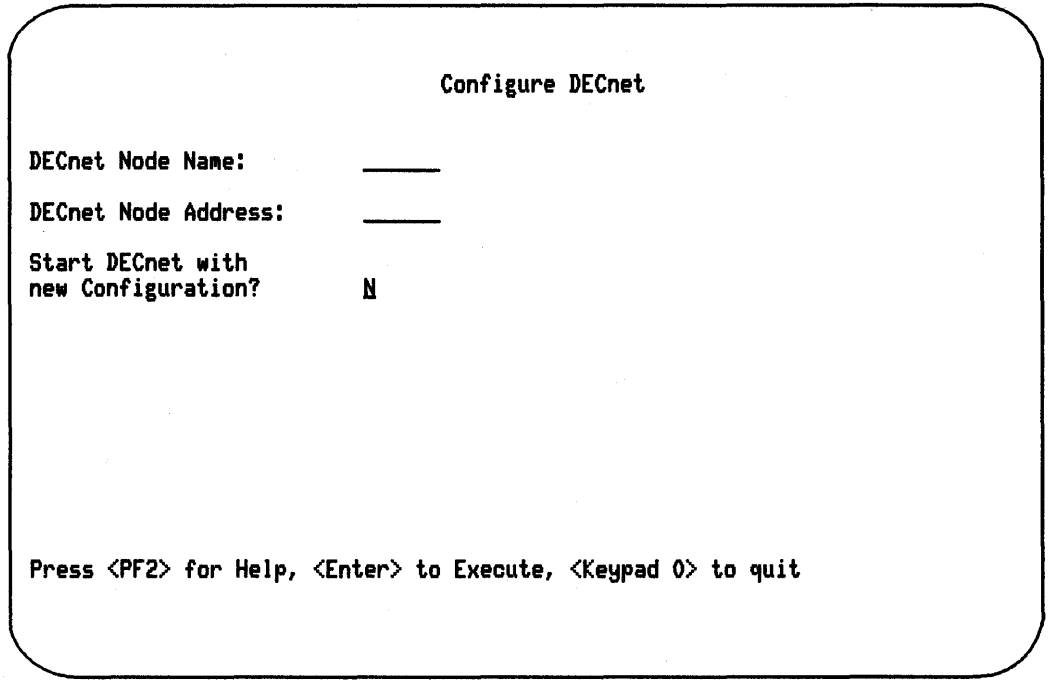
2.8.1 Configuring DECnet

To configure DECnet, select the CONF option from the DECnet Management menu. MANAGER displays the Configure DECnet screen shown in Figure 2–14.

Use this screen from a SYSTEM account to configure a network database in accordance with the VMS-supplied command file NETCONFIG.COM, which is located in the SYS$MANAGER directory.

After the "DECnet Node Address:" prompt, enter both the area number and the node number separated by a period, for example, 5.12. After the prompt "Start DECnet with new Configuration?", enter Y (yes) to start DECnet with the new configuration. Entering N (no) does not make the connection to DECnet.

Figure 2–14: Configure DECnet Screen

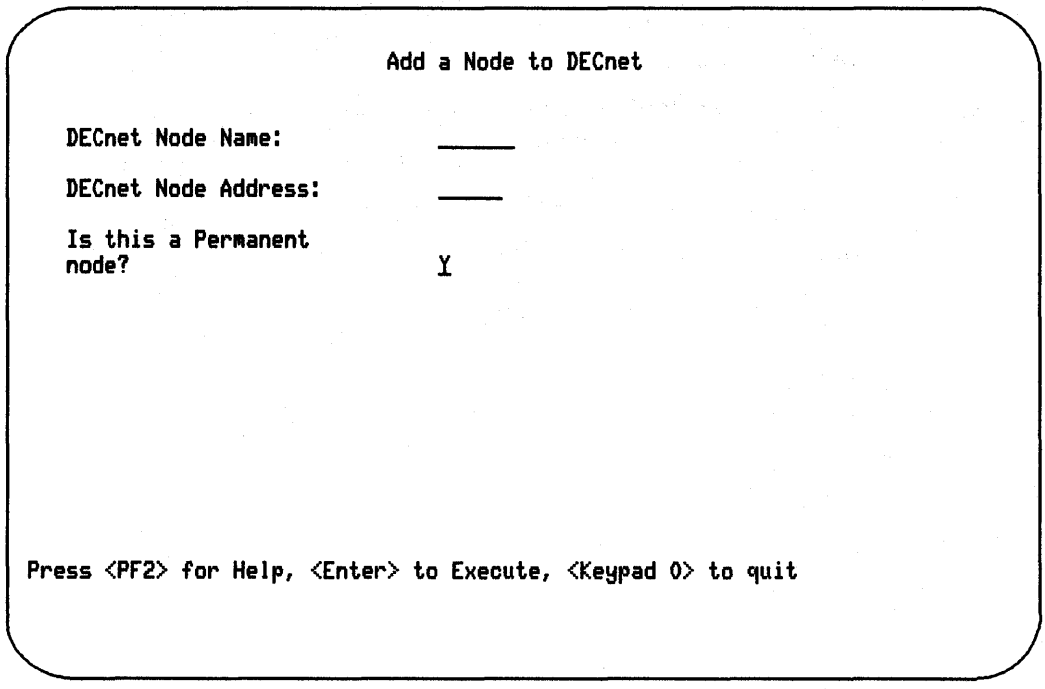


2.8.2 Adding a Node to DECnet

To add a node to DECnet, select the DADD option from the DECnet Management menu. MANAGER displays the Add a Node to DECnet screen shown in Figure 2-15.

Use this screen from a SYSTEM account to add an existing DECnet node to your system's permanent or volatile DECnet database. When you add the node address to the permanent database, that node is always on your system's network. If you add the node address to the volatile database, the node is on your system's network only until your system is turned off or your connection to the network is turned off.

Figure 2–15: Add a Node to DECnet Screen



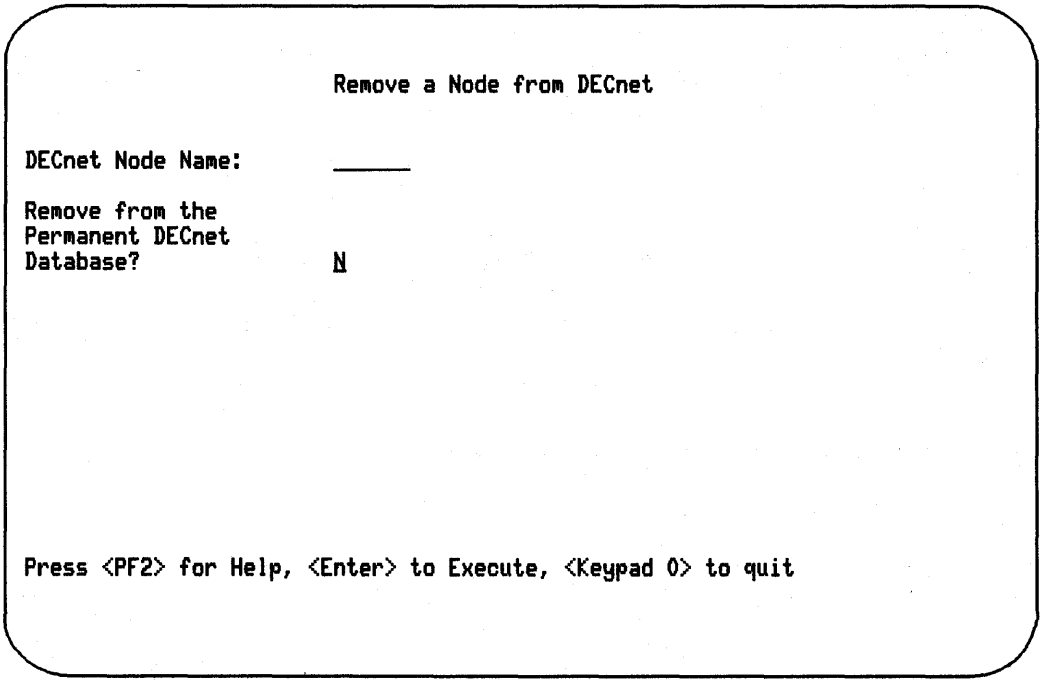
2.8.3 Removing a Node from DECnet

To remove a node from DECnet, select the REMV option from the DECnet Management menu. MANAGER displays the Remove a Node from DECnet screen shown in Figure 2–16.

Use this screen from a SYSTEM account to remove a specific DECnet node from your system's volatile or permanent network files.

After the "Remove from the Permanent DECnet Database?" prompt, enter Y (yes) to remove the node from the permanent and volatile database. Enter N (no) to remove it from the volatile database only.

Figure 2–16: Remove a Node from DECnet Screen

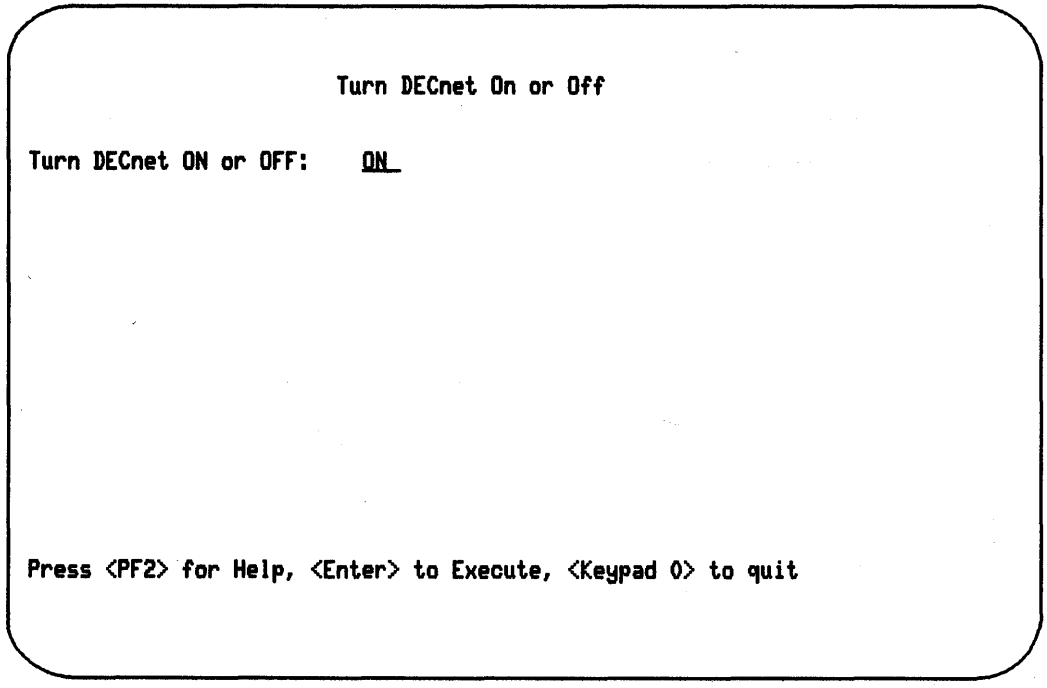


2.8.4 Turning DECnet On or Off

To turn DECnet on or off, select the TURN option from the DECnet Management menu. MANAGER displays the Turn DECnet On or Off screen shown in Figure 2–17.

Use this screen from a SYSTEM account to start up or shut down your system's connection to DECnet.

Figure 2–17: Turn DECnet On or Off Screen

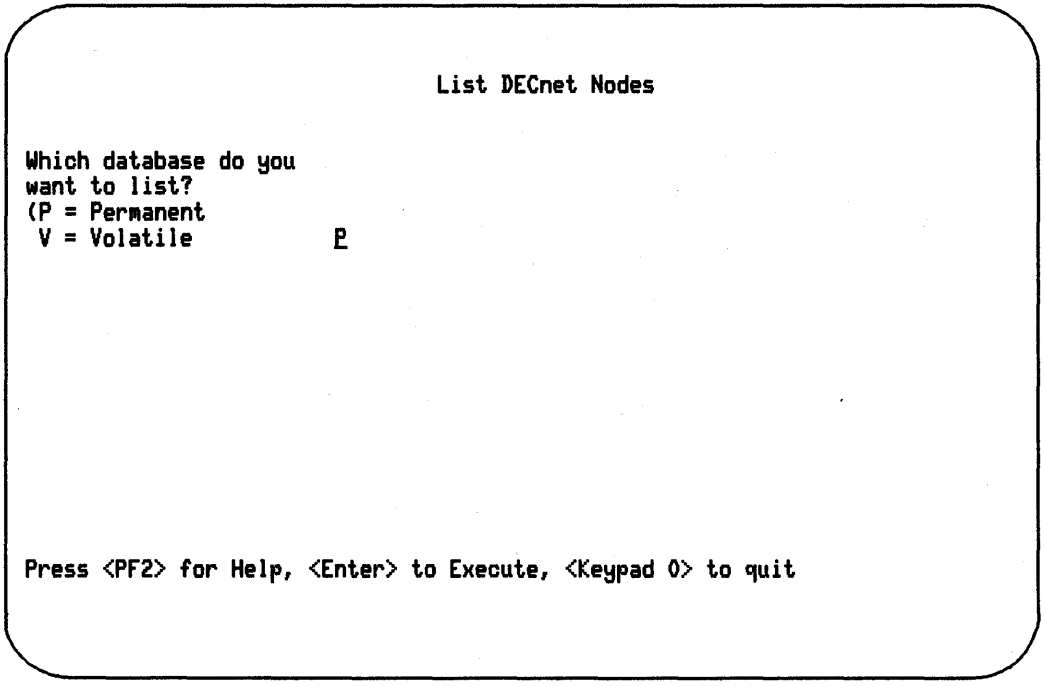


2.8.5 Listing DECnet Nodes

To list DECnet nodes, select the LNET option from the DECnet Management menu. MANAGER displays the List DECnet Nodes screen shown in Figure 2–18.

Use this screen to display a list of the DECnet nodes your system has in either its permanent or volatile database.

Figure 2–18: List DECnet Nodes Screen



2.9 Device Management Tasks

The following sections describe the Device Management tasks.

2.9.1 Mounting a Device

To mount a device, select the MNT option from the Device Management menu. MANAGER displays the Mount a Device screen shown in Figure 2–19.

Use this screen from a SYSTEM account to mount a device, that is, to make the disk or tape on that device available for processing.

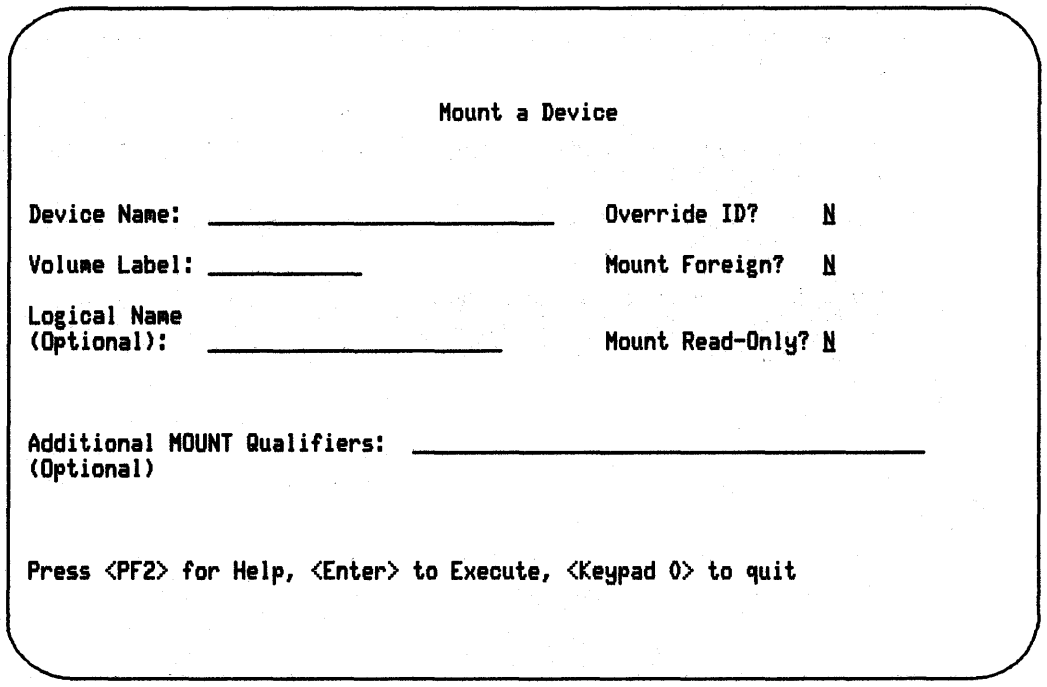
The "Device Name:" prompt accepts either the device's physical or logical name.

After the "Volume Label:" prompt, you can provide a label or leave the field blank. If you leave it blank, you must use either the override option or the foreign option. Use the override option when you do not know the tape label. Use the foreign option when the disk or tape is not in standard format.

Mount the device as read-only to protect the material on the disk or tape, and to ensure that the material is not overwritten.

For further information about mounting a device, including optional MOUNT qualifiers, see the *Guide to VAX/VMS System Management and Daily Operations*.

Figure 2–19: Mount a Device Screen



2.9.2 Initializing a Device

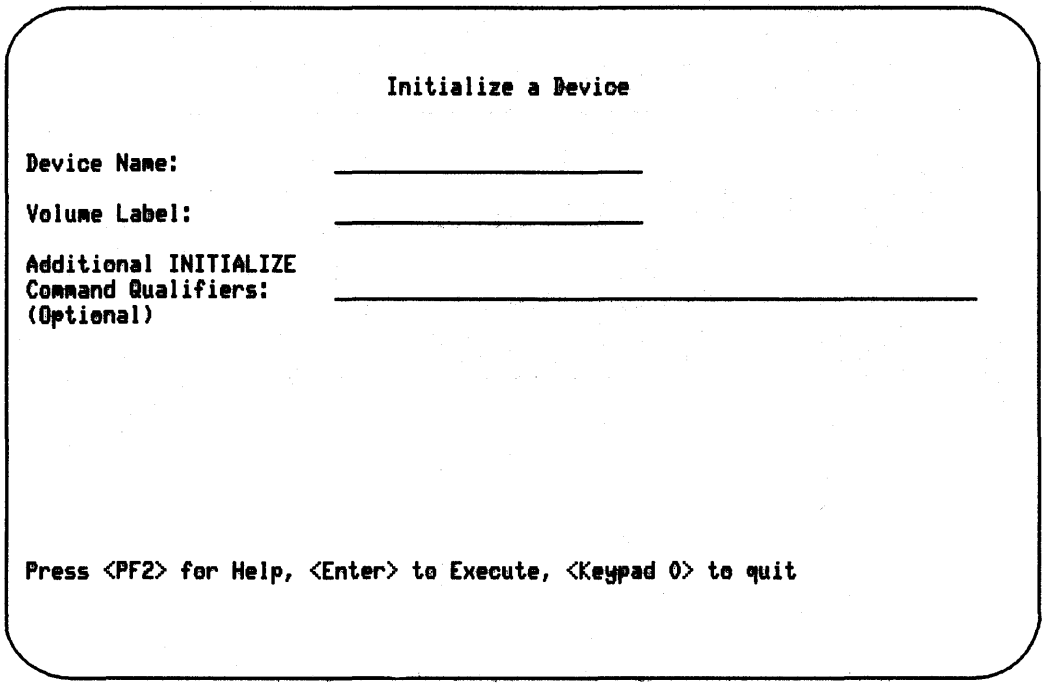
To initialize a device, select the INIT option from the Device Management menu. MANAGER displays the Initialize a Device screen shown in Figure 2–20.

Use this screen to initialize a disk or tape, that is, to erase everything on the disk or tape. Then, prepare the disk or tape for use by formatting it and writing a label for it.

When you initialize a disk or tape, the recommended procedure is to allocate the device first.

For further information about initializing a device, including optional INITIALIZE qualifiers, see the *Guide to VAX/VMS System Management and Daily Operations*.

Figure 2–20: Initialize a Device Screen



2.9.3 Dismounting a Device

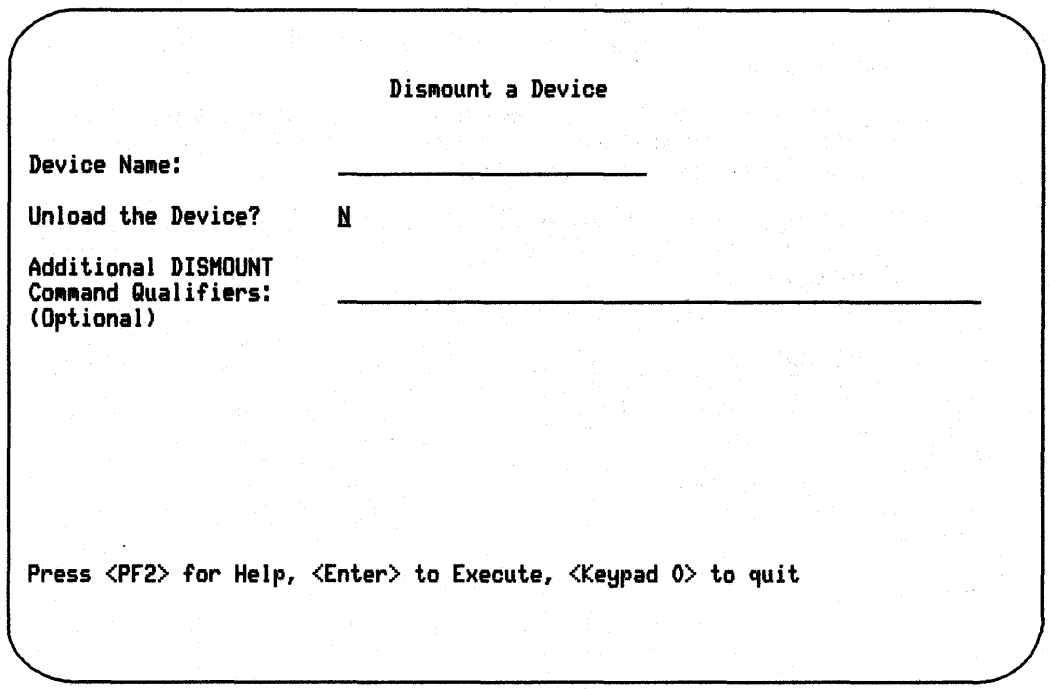
To dismount a device, select the DISM option from the Device Management menu. MANAGER displays the Dismount a Device screen shown in Figure 2–21.

Use this screen to dismount a device, that is, to release the disk drive or tape drive that was previously mounted.

To unload a device means to make the device ready for use by other users, for example, to unload the tape from a tape drive. A Y (yes) response to the "Unload the Device?" prompt means that the device is dismounted and made ready. An N (no) response means the device is dismounted, but not physically unloaded.

For further information about dismounting a device, including optional DISMOUNT qualifiers, see the *Guide to VAX/VMS System Management and Daily Operations*.

Figure 2–21: Dismount a Device Screen



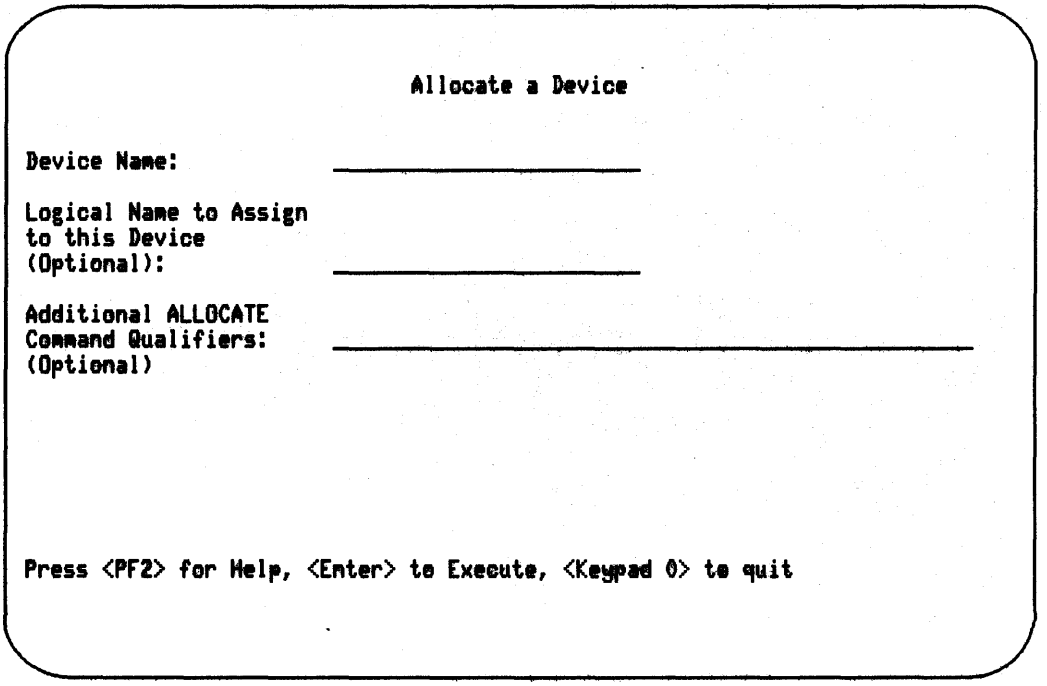
2.9.4 Allocating a Device

To allocate a device, select the ADEV option from the Device Management menu. MANAGER displays the Allocate a Device screen shown in Figure 2–22.

Use this screen to allocate a device to the process you are currently logged in to. When you allocate a device, you provide your process with exclusive access to the device.

For further information about allocating a device, including optional ALLOCATE qualifiers, see the *Guide to VAX/VMS System Management and Daily Operations*.

Figure 2–22: Allocate a Device Screen

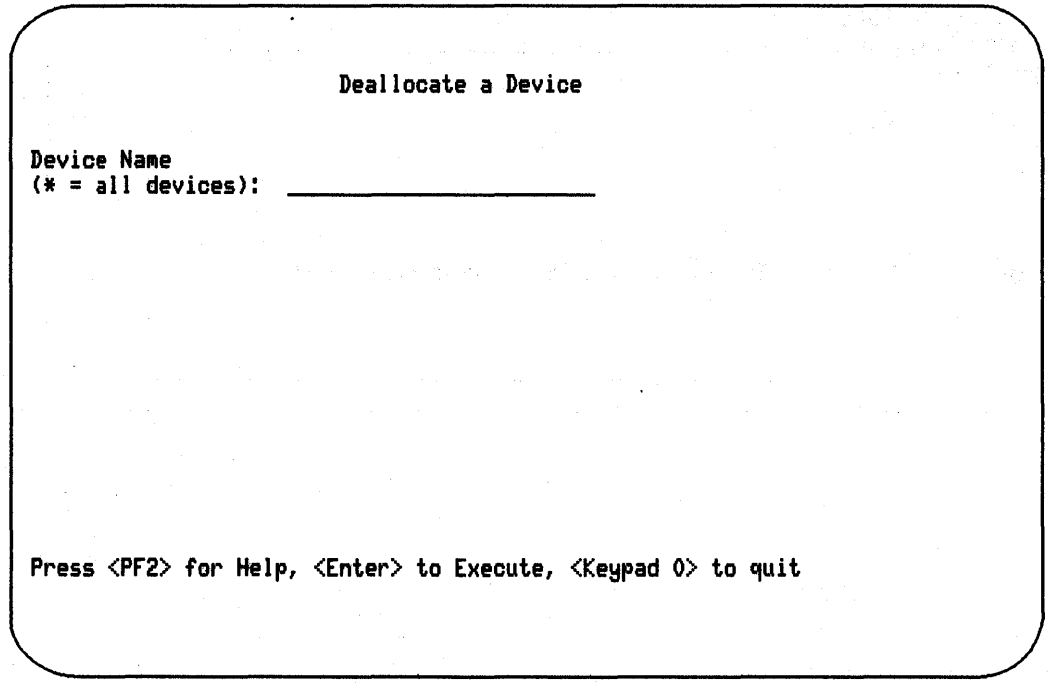


2.9.5 Deallocating a Device

To deallocate a device, select the DDEV option from the Device Management menu. MANAGER displays the Deallocate a Device screen shown in Figure 2–23.

Use this screen to deallocate a device, that is, to make the device available to other users.

Figure 2–23: Deallocate a Device Screen



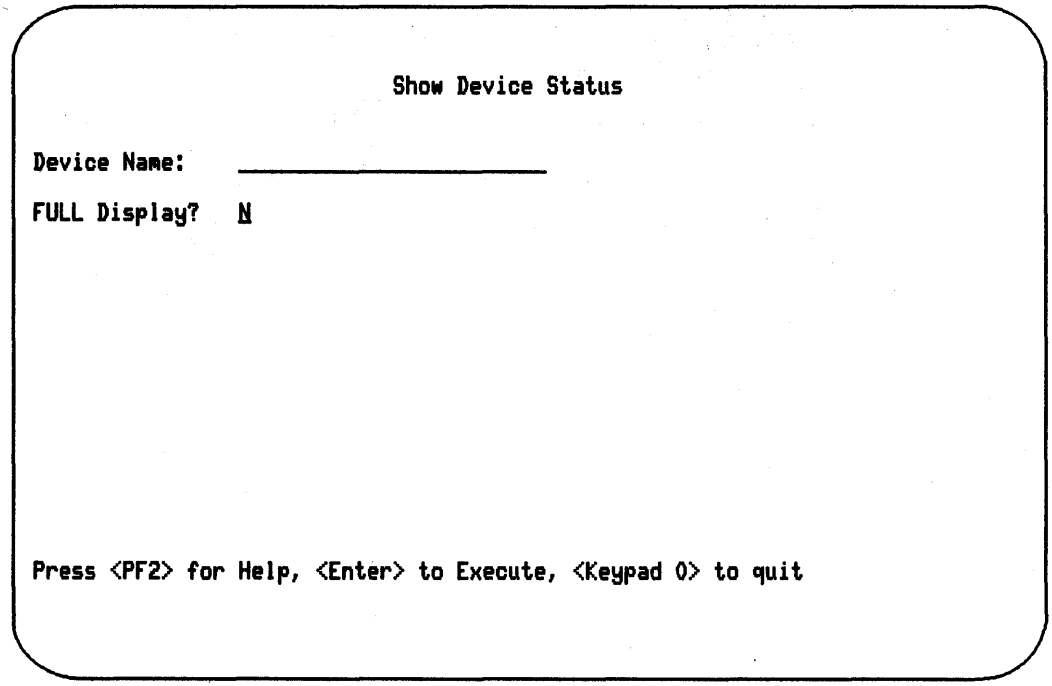
2.9.6 Showing Device Status

To show the status of a device, select the SHOW option from the Device Management menu. MANAGER displays the Show Device Status screen shown in Figure 2–24.

Use this screen to display on your terminal screen information about a device or all devices. You may want to use this screen to check the device status before and after using other device screens.

After the "Device Name:" prompt, you can enter the name of a single device or an asterisk (*) to display all devices. After the "Full Display?" prompt, enter Y (yes) to display all the status information for that device or devices. Press RETURN or enter N (no) to display just the standard information.

Figure 2–24: Show Device Status Screen



2.10 Maintenance Utilities

The following sections describe the Maintenance Utilities.

2.10.1 Using the Backup Utility

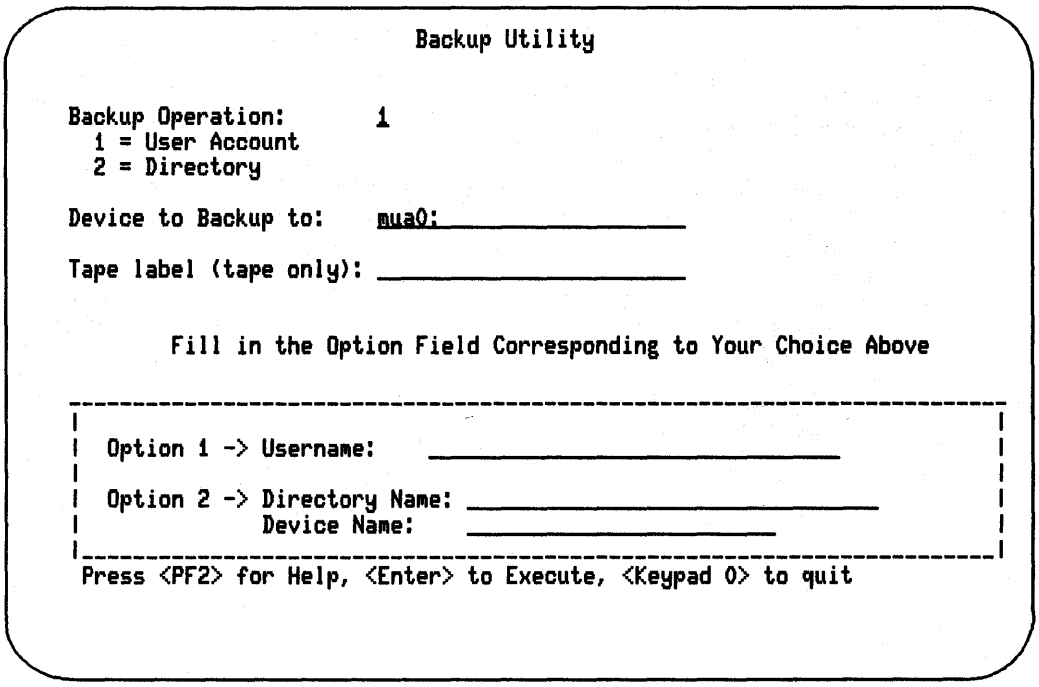
To use the Backup Utility, select the BCK option from the Maintenance Utilities menu. MANAGER displays the Backup Utility screen shown in Figure 2–25.

Use this screen to copy files to permanent offline storage. You can copy an entire user account, or a directory tree that begins with the directory you specify.

Before using this screen, be sure that the backup device is prepared to receive the data.

This screen is a companion to the Restore Utility screen.

Figure 2–25: Backup Utility Screen



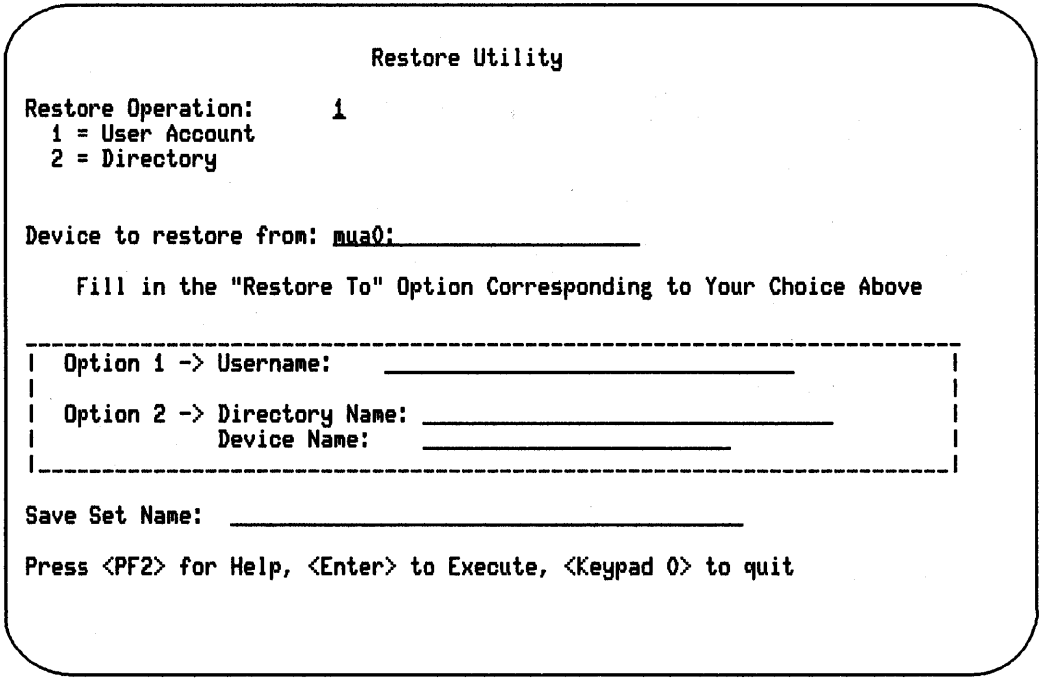
2.10.2 Using the Restore Utility

To use the Restore Utility, select the RST option from the Maintenance Utilities menu. MANAGER displays the Restore Utility screen shown in Figure 2–26.

Use this screen to put back on line the files were put on tape or disk for offline storage.

This screen is a companion to the Backup Utility screen.

Figure 2–26: Restore Utility Screen



Chapter 3

Program Development

This chapter provides an overview of the procedures you use to create, compile, link, and run application programs on a VAXlab system. This chapter also presents information specific to programming languages helpful when developing programs using the VAXlab Software Library (VSL).

Each language section contains sample programs showing how to use the various VSL application routines. The sample programs are shipped with the VSL software and are installed on the system during the installation procedure. Section 3.3, Accessing the VSL Sample Programs, explains how to access and execute the sample programs presented in this document.

3.1 Steps in Program Development

Take the following steps to create and run a VSL application program:

1. **Create the program source code.**

   You can write a VSL application program using any programming language that supports the VAX/VMS Calling Standard. See the language-specific reference manual for information about the VAX/VMS Calling Standard. You can use any VMS editing utility on your VAXlab system when you create your source code files.

   When you create a source code file, be sure to supply the file name with an extension that appropriately identifies the programming language used in the file. If you supply the appropriate file extension when you create the file, you can omit it when you enter the command that compiles the program.

2. **Compile or assemble the source program to produce an object file, a file with an OBJ extension.**

   MACRO source code uses the VAX MACRO assembler. Each of the other programming languages uses its own compiler to create the object files. Object files are not executable, but they contain references to other programs and subroutines that are required to run the program. Compiling a program generally involves establishing these workable references to other programs or to other routines so that they can be combined, or linked, with the program before it is executed.

   In the following command line, the FORTRAN compiler is accessed to compile the program source code in two FORTRAN files, MAIN.FOR and SUB1.FOR. In this example, the subroutine, SUB1.FOR, is contained in a file separate from the main calling program, MAIN.FOR.

   ```
   $ FORTRAN MAIN,SUB1 RETURN
   ```

   The FORTRAN compiler creates an object file (filename.OBJ) for each source code file. The files MAIN.OBJ and SUB1.OBJ are created during the compilation. These object files must be linked together and then linked to the VSL routines before the program is ready to run. The linking is accomplished in step 3.

3. **Link the object files to the VSL libraries to produce an executable image, a file with an EXE extension.**

   You do not have to specify the routine libraries; they are defined during the installation procedure. The LINK command accesses them automatically, as needed.

   You use the LINK command to link together all object files, regardless of the programming language in which the source code files are written.

   Continuing the FORTRAN example, the following command links MAIN.OBJ and SUB1.OBJ with any routines needed from the installed VSL libraries.

   ```
   $ LINK MAIN,SUB1 RETURN
   ```

   The LINK command creates a single executable image file. The file name is the name of the first file specified in the LINK command, and the file extension is EXE. The LINK command shown creates MAIN.EXE.

4. **Execute the image**, using the RUN command with the executable file you created.

```
$ RUN MAIN RETURN
```

If your program does not run satisfactorily, review and revise your source code file, as necessary. Then, recompile, relink, and rerun the program. This process is known as debugging your program.

You can also debug a program using the Debugger Utility, which lets you debug interactively. See the *VAX/VMS Debugger Reference Manual* for information about using the Debugger Utility. In addition, each language-specific reference manual contains a section about using the debugger with that language. Be familiar with the material before using the debugger. Then, recompile your source code file with the /DEBUG/NOOPTIMIZE qualifiers, and relink with the /DEBUG qualifier, as follows:

```
$ FORTRAN/DEBUG/NOOPTIMIZE MAIN,SUB1 RETURN
$ LINK/DEBUG MAIN,SUB1 RETURN
```

Now, when you run the MAIN program, the debugger takes control. You can use debugging commands to stop the execution of the program at a particular statement, and then examine or modify data items.

3.2 Programming Language Considerations

The following sections describe certain programming language-specific information helpful when developing programs using the VSL. **These sections are not intended to be read sequentially.** You need to read only those sections relevant to the programming language you are using.

Information is presented for the following programming languages:

- VAX Ada
- VAX BASIC
- VAX C
- VAX FORTRAN
- VAX PASCAL

You may, in fact, be using a programming language that is not described here. If this is the case, see the reference manual for your language, and become familiar with the appropriate conventions used to:

- Include symbolic definition files
- Declare data types and variables
- Declare and dimension arrays
- Declare external routines
- Default routine call arguments
- Check routine call status
- Use AST routines

3.2.1 Developing Programs in VAX Ada

Be familiar with the information contained in the following sections before you begin developing VSL application programs using VAX Ada. For further information about VAX Ada programming concepts and techniques not covered in this guide, see the *VAX Ada Language Reference Manual.*

3.2.1.1 Including Symbolic Definition Files

The VSL symbolic definition files define the Laboratory I/O (LIO), Laboratory Signal-Processing (LSP), and Laboratory Graphics Package (LGP) error code symbols, the LGP plotting attribute symbols, the LIO set parameter code symbols, the LSP spectral window types, and the entry points containing language-specific interface constructs for the VSL routines. You need to include symbolic definition files in your VSL application programs so that these symbols can be recognized by the programming language you are using.

Table 3–1 lists the symbolic definition files provided for use with VAX Ada.

Table 3–1: VAX Ada Symbolic Definition Files

File Name	Defines:
LGPATTDEF.ADA	LGP plotting attribute symbols
LGPDEF.ADA	LGP error code symbols
LIOERRS.ADA	LIO error code symbols
LIOSET.ADA	LIO set parameter code symbols
LSPDEF.ADA	LSP error code symbols
LSPSET.ADA	LSP spectral window types
VSL.ADA	Entry points containing Ada pragma interface constructs for the VSL routines[1]

[1]Include this symbolic definition file in all VSL application programs written in VAX Ada.

You use the following routine line to include a symbolic definition file in a user program:

```
with filename;
```

where

filename is one of the files listed in Table 3–1.

The files you must include in a user program depend on the VSL facilities the program is designed to use. The file VSL.ADA must be included in all VAX Ada VSL application programs because it defines entry points to VAX Ada pragma interface constructs for the VSL routines.

If your program is designed to use only the LIO routines, then you also need to include those files that define LIO symbolic values. If a program, such as the one presented in Example 3–2, uses routines from all the VSL facilities, then you need to include many of the files listed in Table 3–1. If your application programs use routines from all the VSL facilities, it is advisable to include all the files listed in Table 3–1.

3.2.1.2 Declaring Data Types and Variables

Every VAX Ada data item is associated with a particular data type. A data type is a set of values which share certain characteristics. A data type detemines both the range of values a data item can assume and the operations that can be performed on it. In addition, the type determines the storage space required for all the possible values of the data item.

Some data items are used to pass information from a user program to a device that is to perform some function with the information. Other data items are used to return information from a device to a user program. The VSL application routine reference descriptions explain the functions, syntax, and appropriate usage of the VSL routines. Each routine reference description also explains the routine arguments, their data types, and whether an argument is used to pass information to a device, to return information from the device to the user program, or, in some cases, both.

The VSL application routines use INTEGER, FLOAT, and STRING data types. The INTEGER data type can be broken down further into the SHORT_INTEGER (word) subtype. String literals are basic operations applicable to the data type STRING and to any other one-dimensional array type whose component type is a CHARACTER type. Characters are used to form messages and text. The allowed characters and their ordering are those defined for the ASCII character set. Single character literals are enclosed by single quotes. Strings of characters are enclosed by double quotes.

To explicitly type data items, you use a declarative statement to specify the type, range, and precision of your program values. The following program segment shows how to define some of the data types used in Example 3–2.

```
-- Define data types --

-- Define devspec argument of the LIO$ATTACH routine as a character
-- string 5 characters in length

      Device_to_use     : STARLET.DEVICE_NAME_TYPE(1..5) := "AXA0:";

-- Define the range of the A/D data - 12-bit A/D

      AtoD_data = -2048..2047 - 12-bit A/D

            .
            .
            .


-- Define the title argument of the LGP$PLOT routine as a constant
-- character string

      title_string  : constant string := "ADA_EXAMPLE";

-- Define local variables

       STATUS    : CONDITION_HANDLING.COND_VALUE_TYPE;
     -- STATUS returned by LIO calls
       AXV_ID    : INTEGER;          -- LIO-assigned device ID
       AX_DATA_LENGTH   : INTEGER;   -- Number of data bytes to read
       AX_BUFFER_LENGTH: INTEGER;    -- Buffer size
       AX_PARAM_VALUE1 : INTEGER;    -- Parameter value
       AX_DEVICE_SPECIFIC : VSL.INTEGER_ARRAY(1..2); --
       AX_PARAM_CODE : INTEGER;   -- LIO$K_X parameter code to set
       AX_N_VALUES : INTEGER;   -- Number of parameter values to set
       AX_RUNDOWN       : INTEGER;   -- LIO$DETACH rundown argument
       AX_IO_TYPE : INTEGER;   -- I/O type
       WS_NUMBER : INTEGER;   -- work station number for plotting
       LGP_N   : VSL.INTEGER_ARRAY(1..3); -- number of points to plot
       LGP_ILINE : INTEGER;   -- Plotting line type
       DUMMY_STRING : STRING(1..255);
       CHAR_COUNT : NATURAL;
```

3.2.1.3 Declaring and Dimensioning Arrays

An array is a composite object consisting of components that have the same subtype. The name for a component of an array uses one or more index values belonging to specified discrete types. The value of an array is a composite value consisting of the values of its components.

An array is characterized by the number of indices (the **dimensionality** of the array), the type and position of each index, the lower and upper bounds for each index, and the type and possible constraints of the components. The order of the indices is significant.

A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive. This range of values is called the **index range**.

Array definitions can be named in a type declaration. When you declare the array, the lower and upper bounds follow the type declaration in parentheses and define the maximum size of the array, for example:

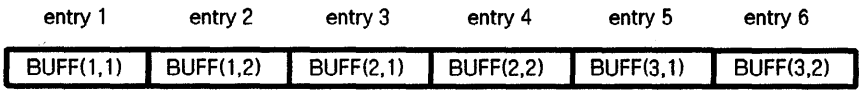
```
-- Define the A/D buffer as a 100-element word array of data in the
-- range of -2048 to 2047.

      AtoD_buffer :  VSL.SHORT_INTEGER_ARRAY(1..100);

-- Define the voltage array as a 100-element single-precision,
-- floating-point (real) array

      plot_buffer :    VSL.FLOAT_ARRAY(1..100); -- data in volts
```

This program segment defines and dimensions the two arrays that are used in Example 3–2. The first array is a short (word or two-byte) integer array of length 100 that is used to contain the 100 raw data values obtained from the analog-to-digital converter on the AXV11-C device. The second array is a single-precision, floating-point (real) array of length 100 that is used to contain the 100 voltages obtained by converting the 100 raw data points to voltages using the LSP routine LSP$FORMAT_TRANSLATE_ADC.

Certain high-level languages have different ways of storing the values associated with two-dimensional arrays. You need to be aware of the way in which VAX Ada stores the values associated with two-dimensional arrays when you declare and dimension arrays in your programs.

VAX Ada stores the values associated with a 2-by-3 two-dimensional array, called BUFF, as a linear one-dimensional array of length six with the storage allocated according to the following indexing system:

Figure 3–1: VAX Ada Array Indexing System



MR-1369-GE

When using two-dimensional arrays in a VAX Ada VSL application program, the leftmost index usually references the buffer number. The rightmost index, which varies faster, can reference consecutive values read from an analog-to-digital converter, or values passed to a multiline plotting routine call.

3.2.1.4 Declaring External Routines

The symbolic definition file VSL.ADA defines the entry points containing the Ada interface pragma constructs for the VSL routines. Including this file in your VSL application programs ensures that the VSL routines are declared external and that their respective arguments are typed appropriately. See Section 3.2.1.1, Including Symbolic Definition Files, for information about including VSL.ADA, as well as other required symbolic definition files, in your VSL application programs.

3.2.1.5 Defaulting Routine Call Arguments

The reference descriptions of the VSL routines describe each routine's syntax and argument list in detail. Some VSL routine arguments are required. A required argument must always be included in the routine's argument list. Some VSL routine arguments are optional. Optional arguments can be included in the routine's argument list at the programmer's discretion. These arguments pass or return information that may or may not be useful to a particular application program. Some optional arguments are useful only with certain VAXlab devices.

Most VSL routine call arguments are assigned default values that are used if a user-supplied value is not included in a routine call argument list. To use a default value supplied by VSL, or to signal the omission of an optional argument in a routine call argument list, you must account for the argument in the routine call argument list by specifying the argument type and appending the 'NULL_PARAMETER attribute to it, for example:

```
VSL.LGP_PLOT    ( STATUS          => STATUS,
                  WS_NUMBER       => WS_NUMBER,
                  MODE_STRING     => MODE_STRING,
                  XARRAY          => VSL.FLOAT_ARRAY'NULL_PARAMETER,
                  YARRAY          => PLOT_BUFFER,
                  N               => LGP_N,
                  XLABEL          => XLABEL_STRING,
                  YLABEL          => YLABEL_STRING,
                  STATUS          => STATUS,
                  ILINE           => LGP_ILINE,
                  IGRID           => INTEGER'NULL_PARAMETER,
                  XCONTROL        => VSL.FLOAT_ARRAY'NULL_PARAMETER,
                  YCONTROL        => VSL.FLOAT_ARRAY'NULL_PARAMETER,
                  COLOR           => VSL.FLOAT_ARRAY'NULL_PARAMETER,
                  TITLE           => TITLE_STRING,
                  METAFLAG        => INTEGER'NULL_PARAMETER,
                  METAFILE_NAME   => STRING'NULL_PARAMETER);
```

In the LGP$PLOT routine call argument list above, the arguments are passed as shown in the following table.

Argument	Value
ws_number	WS_NUMBER
mode_string	MODE_STRING
xarray	VSL.FLOAT_ARRAY'NULL_PARAMETER[1]
yarray	PLOT_BUFFER
n	LGP_N
xlabel	XLABEL_STRING
ylabel	YLABEL_STRING
status	STATUS
iline	LGP_ILINE
igrid	INTEGER'NULL_PARAMETER[1]
xcontrol	VSL.FLOAT_ARRAY'NULL_PARAMETER[1]
ycontrol	VSL.FLOAT_ARRAY'NULL_PARAMETER[1]
color	VSL.FLOAT_ARRAY'NULL_PARAMETER[1]
title	TITLE_STRING
metaflag	INTEGER'NULL_PARAMETER
metafile_name	STRING'NULL_PARAMETER

[1]Uses default value.

3.2.1.6 Checking Routine Call Status

You use the VMS Run-Time Library routine LIB$SIGNAL to signal the status of VSL routine calls. LIB$SIGNAL generates a signal indicating that an exception has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program exits.

The following program segment calls the LIO$READ routine which returns the status of the operation in the variable **STATUS**.

```
VSL.LIO_READ  (   STATUS          => STATUS,
                  DEVICE_ID     => AXV_ID,
                  BUFFER        => AtoD_buffer,
                  BUFFER_LENGTH   => AX_BUFFER_LENGTH,
                  DATA_LENGTH     => AX_DATA_LENGTH,
                  DEVICE_SPECIFIC => AX_DEVICE_SPECIFIC);

   if NOT SUCCESS (STATUS) then
            STOP (STATUS);
   end if;
```

The NOT SUCCESS function tests whether **STATUS** is true or false. If the function returns true, then **STATUS** is odd (bit zero set to one), and program execution continues. If the function returns false, then **STATUS** is even (bit zero set to zero), and the condition is signaled.

The error-handling mechanisms used by the LIO, LSP, and LGP routines are documented in the *Guide to the VAXlab Laboratory I/O Routines*, the *Guide to the VAXlab Laboratory Signal-Processing Routines*, and the *Guide to the VAXlab Laboratory Graphics Package*, respectively. See the appropriate document for complete information about the ways in which each VSL facility performs error handling. These documents also contain detailed information about the error codes returned by each facility and suggested user actions to recover from errors.

3.2.1.7 Using AST Routines

Before you attempt to set up and use AST routines within the context of your VSL applications, be familiar with the information about AST routines discussed in the *Guide to the VAXlab Laboratory I/O Routines*. Once you are familiar with that material, read the remainder of this section carefully to become familiar with the way in which you write your VAX Ada programs to include the use of AST routines.

Example 3–1 contains:

1. Part of a program header that defines the external AST routine.
2. Part of a program header that redefines the LIO$SET routine and the LIO$K_AST_RTN parameter to pass the address of the AST routine by value.
3. The actual AST routine.

Example 3–1: An AST Routine Written in VAX Ada

```
-- Define AST Routine (must reside in your ADA library)

procedure EXAMPLE_ADA_AST(
        STATUS                  : in CONDITION_HANDLING.COND_VALUE_TYPE;
        AX_DEVICE_ID            : in  INTEGER;
        AX_BUFFER               : in  VSL.SHORT_INTEGER_ARRAY;
        AX_BUFFER_LENGTH        : in  INTEGER;
        AX_DATA_LENGTH          : in  INTEGER;
        AX_BUFFER_INDEX         : in  INTEGER;
        AX_DEVICE_SPECIFIC      : in out VSL.INTEGER_ARRAY);

   pragma INTERFACE (ADA, EXAMPLE_ADA_AST);

   pragma IMPORT_PROCEDURE (
              INTERNAL => EXAMPLE_ADA_AST, -- ADA NAME OF PROC
              PARAMETER_TYPES => (CONDITION_HANDLING.COND_VALUE_TYPE,
                                  INTEGER,
                                  VSL.SHORT_INTEGER_ARRAY,
                                  INTEGER,
                                  INTEGER,
                                  INTEGER,
                                  VSL.INTEGER_ARRAY),
              MECHANISM =>       ( reference, reference,reference,
                                   reference, reference,
                                   reference, reference) );

-- Redefine LIO_SET_AST to pass the AST routine address by value

    procedure LIO_SET_AST (
        STATUS          : out CONDITION_HANDLING.COND_VALUE_TYPE;
        DEVICE_ID       : in  INTEGER;
        PARAM_CODE      : in  INTEGER;
        N_VALUES        : in  INTEGER;
        PARAM_VALUE1    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE2    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE3    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE4    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE5    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE6    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE7    : in  INTEGER             := INTEGER'NULL_PARAMETER;
        PARAM_VALUE8    : in  INTEGER             := INTEGER'NULL_PARAMETER);

    pragma INTERFACE (ADA, LIO_SET_AST);
```

Example 3–1 Cont'd. on next page

Example 3–1 (Cont.): An AST Routine Written in VAX Ada

```
    pragma INPORT_VALUED_PROCEDURE (LIO_SET_AST, "LIO$SET_I",
        (CONDITION_HANDLING.COND_VALUE_TYPE, INTEGER, INTEGER, INTEGER,
         INTEGER, INTEGER, INTEGER, INTEGER, INTEGER, INTEGER, INTEGER,
        INTEGER),
        (VALUE, REFERENCE, REFERENCE, REFERENCE, VALUE, REFERENCE,
        REFERENCE, REFERENCE, REFERENCE, REFERENCE, REFERENCE, REFERENCE));

                        .
                        .
                        .

-- Code fragment to "SET" AST routine
-- SIGNAL COMPLETION VIA AST

        AX_PARAM_CODE    := LIOSET.LIO_K_AST_RTN;
        AX_N_VALUES      := 1;

        LIO_SET_AST    (  STATUS => STATUS,
                          DEVICE_ID         => AXV_ID,
                          PARAM_CODE        => AX_PARAM_CODE,
                          N_VALUES           => AX_N_VALUES,
        PARAM_VALUE1   => SYSTEM.TO_INTEGER(EXAMPLE_ADA_AST'address));

           if not SUCCESS (status) then
                 STOP (status);
           end if;

                        .
                        .
                        .

--++ FILE: EXAMPLE_ADA_AST.ADA
--
with TEXT_IO; use TEXT_IO;
with SYSTEM,STARLET;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with VSL;
with LIOSET;
with LIOERRS;
with LSPDEF;
with LGPDEF;

-- AST ROUTINE
```

Example 3–1 Cont'd. on next page

Example 3–1 (Cont.): An AST Routine Written in VAX Ada

```
procedure EXAMPLE_ADA_AST (
        STATUS                  : in CONDITION_HANDLING.COND_VALUE_TYPE;
        AX_DEVICE_ID            : in  INTEGER;
        AX_BUFFER               : in  VSL.SHORT_INTEGER_ARRAY;
        AX_BUFFER_LENGTH        : in  INTEGER;
        AX_DATA_LENGTH          : in  INTEGER;
        AX_BUFFER_INDEX         : in  INTEGER;
        AX_DEVICE_SPECIFIC      : in out VSL.INTEGER_ARRAY) is

        AX_STATUS               : CONDITION_HANDLING.COND_VALUE_TYPE;
        AX_EFN                  : INTEGER;
        AX_FIRST_POINT          : SHORT_INTEGER;

begin

-- STATUS

         if not SUCCESS (STATUS) then
              STOP (STATUS);
         end if;

  .
  .
  .

--- DO SOMETHING
  .
  .
  .

end EXAMPLE_ADA_AST;

  pragma EXPORT_PROCEDURE (EXAMPLE_ADA_AST);
```

Example 3–2: Sample VAX Ada Program Using the VSL Routines

```
--++ FILE: ADA_EXAMPLE.ADA
--
with TEXT_IO; use TEXT_IO;
with SYSTEM,STARLET;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with CONDITION_HANDLING; use CONDITION_HANDLING;
with VSL;
with LIOSET;
with LIOERRS;
with LSPDEF;
with LGPDEF;

procedure ADA_EXAMPLE is

-- This program reads 100 values from channel 2 of the AXV11-C then
-- displays the data in a graph on the screen.

-- This is a simple application using the VSL libraries.

-- This program can be compiled, linked, and run as follows:

--        ACS SET LIBRARY [.YOUR_ADA_LIBRARY_DIRECTORY]
--        COPY VSL, LIO, LSP, and LGP include files from
--             SYS$LIBRARY to your local directory.
--        ADA VSL
--        ADA LIOSET
--        ADA LIOERRS
--        ADA LGPDEF
--        ADA LSPDEF
--        ADA ADA_EXAMPLE
--        ACS LINK ADA_EXAMPLE
--        RUN ADA_EXAMPLE

-- Define data types --

-- Define devspec argument of the LIO$ATTACH routine as a character
-- string 2 characters in length

      Device_to_use : STARLET.DEVICE_NAME_TYPE(1..2) := "AX";

-- Define the range of the A/D data - 12-bit A/D

      AtoD_data = -2048..2047 - 12-bit A/D

-- Define the A/D buffer as a 100-element word array of data in the
-- range of -2048 to 2047.

      AtoD_buffer : VSL.SHORT_INTEGER_ARRAY(1..100);

-- Define the voltage array as a 100-element single-precision,
-- floating-point (real) array
```

Example 3–2 Cont'd. on next page

Example 3–2 (Cont.): Sample VAX Ada Program Using the VSL Routines

```
        plot_buffer : VSL.FLOAT_ARRAY(1..100); -- data in volts

-- Define the range array as a 2-element single-precision,
-- floating point (real) array

        range_array : ARRAY(1..2) OF FLOAT;

-- Define the mode_string argument of the LGP$PLOT routine as a
-- constant character string

        mode_string : constant string := "IXSY";

-- Define the xlabel argument of the LGP$PLOT routine as a
-- constant character string

 xlabel_string  : constant string := "Time";

-- Define the ylabel argument of the LGP$PLOT routine as a
-- constant character string

        ylabel_string  : constant string := "Voltage";

-- Define the title argument of the LGP$PLOT routine as a
-- constant character string

        title_string  : constant string := "ADA_EXAMPLE";

-- Define local variables

        STATUS    : CONDITION_HANDLING.COND_VALUE_TYPE;
      -- STATUS returned by LIO calls
        AXV_ID    : INTEGER;        -- LIO-assigned device ID
        AX_DATA_LENGTH   : INTEGER;  -- Number of data bytes to read
        AX_BUFFER_LENGTH: INTEGER;   -- Buffer size
        AX_PARAM_VALUE1 : INTEGER;   -- Parameter value
        AX_DEVICE_SPECIFIC : VSL.INTEGER_ARRAY(1..2); --
        AX_PARAM_CODE : INTEGER;  -- LIO$K_X parameter code to set
        AX_N_VALUES : INTEGER;  -- Number of parameter values to set
        AX_RUNDOWN       : INTEGER;  -- LIO$DETACH rundown argument
        AX_IO_TYPE : INTEGER;  -- I/O type
        WS_NUMBER : INTEGER;  -- work station number for plotting
        LGP_N  : VSL.INTEGER_ARRAY(1..3); -- number of points to plot
        LGP_ILINE : INTEGER;  -- Plotting line type
        DUMMY_STRING : STRING(1..255);
        CHAR_COUNT : NATURAL;


-- Type pretty message

      put_line ("ADA_EXAMPLE Read data, convert it, plot it");
```

Example 3–2 Cont'd. on next page

Example 3–2 (Cont.): Sample VAX Ada Program Using the VSL Routines

```
-- Attach the AXV11-C to use mapped (polled) I/O.  This routine
-- returns the LIO-assigned device ID for the device.

    AX_IO_TYPE := LIOSET.LIO_K_MAP;

    VSL.LIO_ATTACH (   STATUS      => STATUS,
                       DEVICE_ID   => AXV_ID,
                       DEVSPEC     => Device_to_use,
                       IO_TYPE     => AX_IO_TYPE);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Set up the AXV11-C to use the synchronous I/O interface:

        AX_PARAM_CODE  := LIOSET.LIO_K_SYNCH;
        AX_N_VALUES := 0;
        AX_PARAM_VALUE1 := 0;

    VSL.LIO_SET_I    (  STATUS       => STATUS,
                        DEVICE_ID    => AXV_ID,
                        PARAM_CODE   => AX_PARAM_CODE,
                        N_VALUES     => AX_N_VALUES,
                        PARAM_VALUE1 => AX_PARAM_VALUE1);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Set up AXV11-C channel 2 for input:

        AX_PARAM_CODE    := LIOSET.LIO_K_AD_CHAN;
        AX_N_VALUES      := 1;
        AX_PARAM_VALUE1 := 2;

    VSL.LIO_SET_I    (  STATUS       => STATUS,
                        DEVICE_ID    => AXV_ID,
                        PARAM_CODE   => AX_PARAM_CODE,
                        N_VALUES     => AX_N_VALUES,
                        PARAM_VALUE1      => AX_PARAM_VALUE1);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Set up a channel gain of 1:
```

Example 3–2 Cont'd. on next page

Example 3–2 (Cont.): Sample VAX Ada Program Using the VSL Routines

```
        AX_PARAM_CODE   := LIOSET.LIO_K_AD_GAIN;
        AX_N_VALUES     := 1;
        AX_PARAM_VALUE1 := 1;

    VSL.LIO_SET_I   ( STATUS      => STATUS,
                      DEVICE_ID   => AXV_ID,
                      PARAM_CODE  => AX_PARAM_CODE,
                      N_VALUES    => AX_N_VALUES,
                      PARAM_VALUE1     => AX_PARAM_VALUE1);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Trigger on LIO$READ and fill buffer as fast as possible:

        AX_PARAM_CODE  := LIOSET.LIO_K_TRIG;
        AX_N_VALUES := 1;
        AX_PARAM_VALUE1 := LIOSET.LIO_K_INN_BURST;

    VSL.LIO_SET_I   ( STATUS      => STATUS,
                      DEVICE_ID   => AXV_ID,
                      PARAM_CODE  => AX_PARAM_CODE,
                      N_VALUES    => AX_N_VALUES,
                      PARAM_VALUE1     => AX_PARAM_VALUE1);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Get a raw_data buffer of 100 values.  Use LIO$READ to read the 100
-- A/D values.  Note that the length of the buffer is specified in
-- bytes as is the returned data length.

    AX_BUFFER_LENGTH := 100;

    VSL.LIO_READ  (   STATUS      => STATUS,
                      DEVICE_ID   => AXV_ID,
                      BUFFER      => AtoD_buffer,
                      BUFFER_LENGTH  => AX_BUFFER_LENGTH,
                      DATA_LENGTH => AX_DATA_LENGTH,
                      DEVICE_SPECIFIC  => AX_DEVICE_SPECIFIC);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Detach from the A/D

    AX_RUNDOWN := 0;
```

Example 3–2 Cont'd. on next page

Example 3–2 (Cont.): Sample VAX Ada Program Using the VSL Routines

```
    VSL.LIO_DETACH (   STATUS     => STATUS,
                       DEVICE_ID  => AXV_ID,
                       RUNDOWN    => AX_RUNDOWN);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Convert the raw data to voltages using LSP$FORMAT_TRANSLATE_ADC

    VSL.LSP_FORMAT_TRANSLATE_ADC  ( STATUS => STATUS,
        I           => AtoD_buffer,
        OU          => plot_buffer,
        N           => AX_BUFFER_LENGTH,
        CONTROL_I   => VSL.INTEGER_ARRAY'NULL_PARAMETER,
        RANG        => VSL.FLOAT_ARRAY'NULL_PARAMETER,
        OPT_STATUS => STATUS);

        if NOT SUCCESS (STATUS) then
                 put_line ("problem in LSP_FORMAT_TRANSLATE_ADC");
               STOP (STATUS);
        end if;

-- Plot the data using LGP$PLOT to plot the voltages on the
-- terminal screen.

        WS_NUMBER := 1;

        LGP_N(1) := AX_DATA_LENGTH;
        LGP_N(2) := 0;
        LGP_N(3) := 0;
        LGP_ILINE := 1;
```

Example 3–2 Cont'd. on next page

Example 3-2 (Cont.): Sample VAX Ada Program Using the VSL Routines

```
    VSL.LGP_PLOT    ( STATUS => STATUS,
        WS_NUMBER => WS_NUMBER,
        MODE_STRING => MODE_STRING,
        XARRAY  => VSL.FLOAT_ARRAY'NULL_PARAMETER,
        YARRAY  => PLOT_BUFFER,
        N  => LGP_N,
        XLABEL  => XLABEL_STRING,
        YLABEL  => YLABEL_STRING,
        OPT_STATUS => STATUS,
        ILINE  => LGP_ILINE,
        IGRID  => INTEGER'NULL_PARAMETER,
        XCONTROL => VSL.FLOAT_ARRAY'NULL_PARAMETER,
        YCONTROL => VSL.FLOAT_ARRAY'NULL_PARAMETER,
        COLOR  => VSL.FLOAT_ARRAY'NULL_PARAMETER,
        TITLE  => TITLE_STRING,
        METAFLAG => INTEGER'NULL_PARAMETER,
        METAFILE_NAME => STRING'NULL_PARAMETER);

        if NOT SUCCESS (STATUS) then
                  STOP (STATUS);
        end if;

-- Wait for a carriage return before deleting plot.

        put_line("Type carriage return to exit");
        get_line(DUMMY_STRING, CHAR_COUNT);

-- Terminate the plot.

    VSL.LGP_TERMINATE_PLOT    ( STATUS => STATUS,
                                WS_NUMBER => WS_NUMBER);

       if NOT SUCCESS (STATUS) then
                 STOP (STATUS);
       end if;

end ADA_EXAMPLE;
```

3.2.2 Developing Programs in VAX BASIC

Be familiar with the information contained in the following sections before you begin developing VSL application programs using VAX BASIC. For further information about VAX BASIC programming concepts and techniques not covered in this guide, see the *VAX BASIC Reference Manual.*

3.2.2.1 Including Symbolic Definition Files

The VSL symbolic definition files define the Laboratory I/O (LIO), Laboratory Signal-Processing (LSP), and Laboratory Graphics Package (LGP) error code symbols, the LGP plotting attribute symbols, the LIO set parameter code symbols, the LSP spectral window types, and the entry points containing language-specific interface constructs for the VSL routines. You need to include symbolic definition files in your VSL application programs so that these symbols can be recognized by the programming language you are using.

Table 3–2 lists the symbolic definition files provided for use with VAX BASIC.

Table 3–2: VAX BASIC Symbolic Definition Files

File Name	Defines:
LGPATTDEF.BAS	LGP plotting attribute symbols
LGPDEF.BAS	LGP error code symbols
LIOERRS.BAS	LIO error code symbols
LIOSET.BAS	LIO set parameter code symbols
LSPDEF.BAS	LSP error code symbols
LSPSET.BAS	LSP spectral window types
VSL.BAS	Entry points containing BASIC interface constructs for the VSL routines[1]

[1]Include this symbolic definition file in all VSL application programs written in VAX BASIC.

You use the following routine line to include a symbolic definition file in a user program:

```
%INCLUDE "SYS$LIBRARY:filename.BAS"
```

where

filename is one of the files listed in Table 3–2.

The files you must include in a user program depend on the VSL facilities the program is designed to use. The file VSL.BAS must be included in all VAX BASIC VSL application programs because it defines entry points containing VAX BASIC interface constructs for the VSL routines.

If your program is designed to use only the LIO routines, then you also need to include those files that define LIO symbolic values. If a program, such as the one presented in Example 3–4, uses routines from all the VSL facilities, then you need to include many of the files listed in Table 3–2. If your application programs use routines from all the VSL facilities, it is advisable to include all the files listed in Table 3–2.

3.2.2.2 Declaring Data Types and Data Items

All data in a VAX BASIC program has a specific **data type** that determines how many bits of storage to consider as a unit and how to interpret and manipulate the unit. **Data items** are named quantities whose values can change during program execution. Each data item name refers to a location in the program's storage area.

Some data items are used to pass information from a user program to a device that is to perform some function with the information. Other data items are used to return information from a device to a user program. The VSL application routine reference descriptions explain the functions, syntax, and appropriate usage of the VSL routines. Each routine reference description also explains the routine arguments, their data types, and whether an argument is used to pass information to a device, to return information from the device to the user program, or, in some cases, both.

The VSL application routines use INTEGER, REAL, and STRING data types. The INTEGER data type can be broken down further into the BYTE, WORD, and LONG subtypes. You can specify data-type defaults in a program module using the OPTION TYPE = EXPLICIT statement. This statement means that any program value not explicitly declared causes VAX BASIC to signal an error. Explicit data typing makes programs easier to understand and maintain because the data type of all program values is explicitly spelled out in the program and is not dependent upon compilation defaults that may change.

To explicitly type data items, you use a declarative statement to specify the type, range, and precision of your program values, for example:

```
DATA_ITEM_DECLARATIONS:

! Declare dummy data item to be a character string:
        DECLARE                 !The DECLARE statement!                 &
            STRING              !Data type = string!                    &
                dummy           !Data item to accept carriage return!
```

```
! Declare local data items to be longword integers:
        DECLARE                 !The DECLARE statement!             &
            LONG                !Data type = Longword integer!      &
                S_STATUS        !STATUS returned by LIO calls!      &
                AXV_ID          !LIO-assigned device ID!            &
                DATA_LENGTH     !Number of data bytes to read!
```

This program segment is excerpted from Example 3–4 and shows the explicitly typed data items used in that sample program.

If a variable or an array is to be shared by the main program and a subroutine or an AST routine, use a COMMON statement to declare them. The COMMON statement defines a named, shared storage area called a COMMON block or program segment (PSECT). All BASIC program modules can access the values stored in the COMMON block by specifying a COMMON statement with the same name. See Section 3.2.2.7, Using AST Routines for further information.

3.2.2.3 Declaring and Dimensioning Arrays

An array is a set of data ordered in any number of dimensions. A one-dimensional array, such as BUFFER(9), is called a list or vector. A two-dimensional array, such as BUFFER(9,9) is called a matrix. VAX BASIC always allocates element zero, so specify the bounds of your array to be one less than the actual number you require. For example, BUFFER(9) contains 10 elements, and BUFFER(9,9) contains 100 elements.

Dimensioning an array means to specify the name, data type, and maximum size of the array. Subscripts follow the data item name in parentheses and define its position in the array. When you create an array, bounds follow the array name in parentheses and define the maximum size of the array, for example:

```
!Declare data buffers:
    DIMENSION
            WORD
                RAW_DATA(99) !100 point raw data buffer!
            REAL
                VOLTAGES(99) !100 voltages to plot!
```

This program segment dimensions the two arrays that are used in Example 3–4. The first array, RAW_DATA (99), is a word (two-byte) integer array of length 100 that is used to contain 100 raw data values obtained from the analog-to-digital converter on the AXV11-C device. The second array, VOLTAGES(99), is a single-precision, floating-point

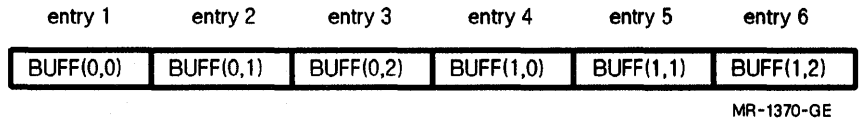
(real) array of length 100 that is used to contain the 100 voltages obtained by converting the 100 raw data points to voltages using the LSP routine LSP$FORMAT_TRANSLATE_ADC.

Certain high-level languages have different ways of storing the values associated with two-dimensional arrays. You need to be aware of the way in which VAX BASIC stores the values associated with two-dimensional arrays when you declare and dimension arrays in your programs. For example, suppose you need to declare a 2-by-3 two-dimensional array, called BUFF. You declare the array as follows:

```
DECLARE LONG BUFF(1,2)
```

VAX BASIC stores the values associated with the 2-by-3 two-dimensional array called BUFF, as a linear one-dimensional array of length six with the storage allocated according to the following indexing system:

Figure 3–2: VAX BASIC Array Indexing System



MR-1370-GE

When using two-dimensional arrays in a VAX BASIC VSL application program, the leftmost index usually references the buffer number. The rightmost index, which varies faster, can reference consecutive values read from an analog-to-digital converter, or values passed to a multiline plotting routine call.

3.2.2.4 Declaring External Routines

The symbolic definition file VSL.BAS defines the entry points containing the BASIC interface language constructs for the VSL routines. Including this file in your VSL application programs ensures that the VSL routines are declared external and that their respective arguments are typed appropriately. See Section 3.2.2.1, Including Symbolic Definition Files for information about including VSL.BAS, as well as other required symbolic definition files, in your VSL application programs.

If your VSL application programs use VMS library routines, such as LIB$SIGNAL, these routines must be declared external within the program context. The following program segment, excerpted from Example 3–4 declares the VMS Run-Time Library routine, LIB$SIGNAL, as an external longword function. This routine and other VMS routines are declared as functions because they return a status value that a program needs to check.

```
! Declare the VMS routine as an external longword function

    EXTERNAL LONG FUNCTION LIB$SIGNAL
```

See Section 3.2.2.6, Checking Routine Call Status for information about how the LIB$SIGNAL routine is used to check routine call status.

3.2.2.5 Defaulting Routine Call Arguments

The reference descriptions of the VSL routines describe each routine's syntax and argument list in detail. Some VSL routine arguments are required. A required argument must always be included in the routine's argument list. Some VSL routine arguments are optional. Optional arguments can be included in the routine's argument list at the programmer's discretion. These arguments pass or return information that may or may not be useful to a particular application program. Some optional arguments are useful only with certain VSL devices.

Most VSL routine call arguments are assigned default values that are used if a user-supplied value is not included in a routine call argument list. To use a default value supplied by VSL, or to signal the omission of an optional argument in a routine call argument list, you must account for the argument in the routine call argument list by including a comma in place of the actual argument, for example:

```
CALL LGP$PLOT(1%, "IXSY", , VOLTAGES() BY REF, 100%, "Time",        &
              "Voltage", , 1%, , , , , "BASIC_EXAMPLE")
```

In the LGP$PLOT routine call argument list above, the arguments are passed as shown in the following table.

Argument	Value
ws_number	1%
mode_string	"IXSY"
xarray	Comma[1]
yarray	VOLTAGES() BY REF
n	100%
xlabel	"Time"
ylabel	"Voltage"
status	Comma[1]
iline	1%
igrid	Comma[1]
xcontrol	Comma[1]
ycontrol	Comma[1]
color	Comma[1]
title	"BASIC_EXAMPLE"
metaflag	Omitted
metafile_name	Omitted

[1]Uses default value.

For LSP and LGP routine calls, you can omit optional arguments at the end of a routine call argument list. You can terminate the list after the last user-supplied value of an optional argument is specified.

For LIO routine calls, you must explicitly specify or explicitly default optional arguments regardless of the argument's place in the routine call argument list. You cannot terminate the list after the last user-supplied value of an optional argument is specified.

3.2.2.6 Checking Routine Call Status

You use the VMS Run-Time Library routine LIB$SIGNAL to signal the status of VSL routine calls. LIB$SIGNAL generates a signal that indicates that an exception condition has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program exits.

The following program segment calls the LIO$READ routine as a function which returns the status of the operation in the variable S_STATUS.

```
S_STATUS = LIO$READ(AXV_ID, RAW_DATA() BY REF, 200%, DATA_LENGTH, )
          CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%
```

The value of S_STATUS is ANDed with 1. If this logical operation produces a 1, then status is odd (bit zero set to one), and program execution continues. If this operation does not produce a 1, then status is even (bit zero set to zero), and the condition is signaled.

NOTE

When using the bitwise AND, as is the case here when checking routine call status, the value of both bits must be 1 for the result to be 1.

The error-handling mechanisms used by the LIO, LSP, and LGP routines are documented in the *Guide to the VAXlab Laboratory I/O Routines*, the *Guide to the VAXlab Laboratory Signal-Processing Routines*, and the *Guide to the VAXlab Laboratory Graphics Package*, respectively. See the appropriate document for complete information about the ways in which each VSL facility performs error handling. These documents also contain detailed information about the error codes returned by each facility and suggested user actions to recover from errors.

3.2.2.7 Using AST Routines

Before you attempt to set up and use AST routines within the context of your VSL applications, be familiar with the information about AST routines discussed in the *Guide to the VAXlab Laboratory I/O Routines*. Once you are familiar with that material, read the remainder of this section carefully to become familiar with the way in which you write your VAX BASIC programs to include the use of AST routines.

Example 3-3 is an AST routine written in VAX BASIC that receives completed buffers from a device, processes them, and requeues them to the device. See the online sample program LIO_ADV_AST.BAS for a complete VAX BASIC VSL application program that uses this subroutine.

Example 3-3: An AST Routine Written in VAX BASIC

```
!Main program sets up the AST routine
!Declare the common used to communicate with the AST routine.
COMMON (ast) LONG done_flag       !AST routine done flag!          &
             LONG buffer_count    !Numbers of buffer processed!    &
             LONG ast_status      !AST routine status!
      .
      .
      .
EXTERNAL LONG adv_ast     ! Declare the AST routine as an external
                          ! longword procedure
      .
      .
      .
! Supply an AST routine to be called when a buffer is complete
s_status = LIO$SET_I(DEVICE_ID, LIO$K_AST_RTN, 1%, ADV_AST)
      .
      .
      .
! The AST routine is called by LIO when a buffer completes.
!
! This subroutine receives completed buffers from the ADV11-D
! device. Each buffer is processed by the AST routine as it is
! received from the ADV11-D device and is then requeued to the
! device for further use. A total of five buffers are received
! and processed by this AST routine.
```

Example 3-3 Cont'd. on next page

Example 3–3 (Cont.): An AST Routine Written in VAX BASIC

```
5000    SUB adv_ast(LONG s_status, LONG device_id, buffer() BY REF,   &
            LONG buffer_length, LONG data_length, LONG buffer_index,  &
            LONG device_specific)

        ! Note that the AST routine dummy arguments are similar to the
        ! arguments to the LIO$DEQUEUE routine.  These arguments are:
        !
        ! s_status          Returns the status of the I/O operation.
        ! device_id         Specifies the LIO-assigned device ID of
        !                     the ADV11-D.
        ! buffer            The actual buffer, NOT the buffer address
        !                     as in the LIO$DEQUEUE routine call.
        ! buffer_length     The length of the buffer, in bytes.
        ! data_length       The length of the data in the buffer, in bytes.
        ! buffer_index      The buffer index, if one is supplied in the
        !                     LIO$ENQUEUE routine call.
        ! device_specific   A dummy argument here.  The ADV11-D device
        !                     does not support a device-specific argument.
        ! Declare the common used to communicate with the main program.
        ! These COMMON definitions must also be declared in the main program.
        !
        COMMON (ast) LONG done_flag      !AST routine done flag!        &
                     LONG buffer_count   !Numbers of buffer processed!  &
                     LONG ast_status     !AST routine status!

6000    FUNCTIONS:
        ! Declare the VMS functions
        !
        EXTERNAL LONG FUNCTION LIB$SIGNAL

7000    EXECUTE:
        ! Save the I/O status where the main program can check if an
        ! error condition occurs.

        ast_status = s_status

        ! Increment the buffer count

        buffer_count = buffer_count + 1
        ! If the buffer count is less than 5, and if no error condition
        ! has occured, requeue the buffer to the ADV11-D.  Otherwise,
        ! set the I/O done flag and exit.

        IF (buffer_count < %5) AND (s_status and 1%) THEN
           status = LIO$ENQUEUE(device_id, buffer() BY REF, buffer_length, &
                    , , , )
             CALL LIB$SIGNAL BY VALUE (s_status) IF (s_status AND 1%) = 0%
```

Example 3–3 Cont'd. on next page

Example 3–3 (Cont.): An AST Routine Written in VAX BASIC

```
    ELSE
         done_flag = 1%
    END IF

    END SUB
```

Example 3–4: Sample VAX BASIC Program Using the VSL Routines

```
10  REM BASIC_EXAMPLE.BAS
       This program reads 100 values from channel 2 of the AXV11-C then
       displays the data in a graph on the screen.

       This is a simple application using the VSL libraries.

       This program can be compiled, linked, and run as follows:
              BASIC BASIC_EXAMPLE
              LINK BASIC_EXAMPLE
              RUN BASIC_EXAMPLE

1000    BEGINNING:

              OPTION
                     TYPE = EXPLICIT

              %INCLUDE "SYS$LIBRARY:LIOSET.BAS"
              %INCLUDE "SYS$LIBRARY:LIOERRS.BAS"
              %INCLUDE "SYS$LIBRARY:LSPDEF.BAS"
              %INCLUDE "SYS$LIBRARY:LGPDEF.BAS"
              %INCLUDE "SYS$LIBRARY:VSL.BAS"

       DATA_ITEM_DECLARATIONS:

       ! Declare dummy data item
              DECLARE                                                  &
                  STRING                                               &
                      dummy      !Variable to accept carriage return!

       ! Declare local data items
              DECLARE                                                  &
                  LONG                                                 &
                      S_STATUS,      !STATUS returned by LIO calls!    &
                      AXV_ID,        !LIO-assigned device ID!          &
                      DATA_LENGTH,   !Number of data bytes to read!
```

Example 3–4 Cont'd. on next page

Example 3–4 (Cont.): Sample VAX BASIC Program Using the VSL Routines

```
! Declare data buffers
        DIMENSION                                                   &
                WORD                                                &
                        RAW_DATA(99),   !100 pt raw data buffer!    &
                REAL                                                &
                        VOLTAGES(99)    !100 voltages to plot!

! Declare the LIO routines
        EXTERNAL LONG FUNCTION LIB$SIGNAL

SET_UP_THE_AXV:

! Type pretty message
!
        PRINT "BASIC_EXAMPLE: Read data, convert it, plot it"
        PRINT

! Attach the AXV11-C to use mapped (polled) I/O.  This routine
! call returns an LIO-assigned device ID for the device.
!
        S_STATUS = LIO$ATTACH (AXV_ID, "AXAO:", LIO$K_MAP)
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%

! Set up the AXV11-C to use the synchronous I/O interface:
        S_STATUS = LIO$SET_I (AXV_ID, LIO$K_SYNCH, 0%)
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%

! Set up AXV11-C channel 2 for input:
        S_STATUS = LIO$SET_I (AXV_ID, LIO$K_AD_CHAN, 1%, 2%)
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%

! Set up a channel gain of 1:
        S_STATUS = LIO$SET_I (AXV_ID, LIO$K_AD_GAIN, 1%, 1%)
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%

! Trigger on LIO$READ and fill buffer as fast as possible:
        S_STATUS = LIO$SET_I (AXV_ID, LIO$K_TRIG, 1%, LIO$K_IMM_BURST)
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%
```

Example 3–4 Cont'd. on next page

Example 3–4 (Cont.): Sample VAX BASIC Program Using the VSL Routines

```
GET_AND_DISPLAY_DATA:

! Get a RAW_DATA buffer of 100 values.
! This program uses LIO$READ to read the 100 A/D values.
! Note that the length of the buffer is in bytes, as is the
! returned data_length.

        S_STATUS = LIO$READ(AXV_ID, RAW_DATA() BY REF, 200%,    &
                DATA_LENGTH, )
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%

! Detach from the A/D:
        S_STATUS = LIO$DETACH(AXV_ID, 0%)
        CALL LIB$SIGNAL BY VALUE (S_STATUS) IF (S_STATUS AND 1%) = 0%

! Convert the raw data to voltages using LSP$FORMAT_TRANSLATE_ADC:
        CALL LSP$FORMAT_TRANSLATE_ADC(RAW_DATA() BY REF,        &
                VOLTAGES() BY REF,  100%, , , )

! Plot the data using LGP$PLOT to plot the voltages on the
! terminal screen:
        CALL LGP$PLOT(1%, "IXSY", , VOLTAGES() BY REF, 100%,    &
             "Time", "Voltage", , 1%, , , , , "BASIC_EXAMPLE")

! Wait for a carriage return before deleting plot:
        PRINT "Type carriage return to exit";
        INPUT dummy

! Terminate the plot:
        CALL LGP$TERMINATE_PLOT(1%)

        END
```

3.2.3 Developing Programs in VAX C

Be familiar with the information contained in the following sections before you begin developing VSL application programs using VAX C. For further information about VAX C programming concepts and techniques not covered in this guide, see the *Guide to VAX C*.

3.2.3.1 Including Symbolic Definition Files

The VSL symbolic definition files define the Laboratory I/O (LIO), Laboratory Signal-Processing (LSP), and Laboratory Graphics Package (LGP) error code symbols, the LGP plotting attribute symbols, the LIO set parameter code symbols, the LSP spectral window types, and the entry points containing language-specific interface constructs for the VSL routines. You need to include symbolic definition files in your VSL application programs so that these symbols can be recognized by the programming language you are using.

Table 3–3 lists the symbolic definition files provided for use with VAX C.

Table 3–3: VAX C Symbolic Definition Files

File Name	Defines:
LGPATTDEF.H	LGP plotting attribute symbols
LGPDEF.H	LGP error code symbols
LIOERRS.H	LIO error code symbols
LIOSET.H	LIO set parameter code symbols
LSPDEF.H	LSP error code symbols
LSPSET.H	LSP spectral window types
VSL.H	Entry points containing C interface constructs for the VSL routines[1]

[1]Include this symbolic definition in all VSL application programs written in VAX C.

You use the following routine line to include a symbolic definition file in a user program:

```
#INCLUDE <filename.H>
```

where

filename is one of the files listed in Table 3–3.

The files you must include in a user program depend on the VSL facilities the program is designed to use. The file VSL.H must be included in all VAX C VSL application programs because it defines entry points containing VAX C interface constructs for the VSL routines.

If your program is designed to use only the LIO routines, then you also need to include those files that define LIO symbolic values. If a program, such as the one presented in Example 3–6, uses routines from all the VSL facilities, then you need to include many of the files listed in Table 3–3. If your application programs use routines from all the VSL facilities, it is advisable to include all the files listed in Table 3–3.

3.2.3.2 Declaring Data Types and Variables

You can represent data in a VAX C program using constants. A constant is a primary expression with a defined value that does not change during program execution. You can represent a constant in a literal form, which contains the explicit numbers, letters, and operators that comprise the constant. Or, you can define a symbol to represent the constant value. Constants have **data types**, as do all data in VAX C. The data type determines the amount of storage needed and how to interpret the stored constant value. The compiler determines the data type of constants by the way in which their values are represented in the program source code.

You can also represent data in VAX C using variables, whose values can change during program execution. You must explicitly **declare** all variables used in a program. When you declare a variable, you specify the data type of the stored **object**. In VAX C, an object is a value requiring storage.

Some data items are used to pass information from a user program to a device that is to perform some function with the information. Other data items are used to return information from a device to a user program. The VSL application routine reference descriptions explain the functions, syntax, and appropriate usage of the VSL routines. Each routine reference description also explains the routine arguments, their data types, and whether an argument is used to pass information to a device, to return information from the device to the user program, or, in some cases, both.

Integer variables are declared with the keywords **int** (longword), **long** (longword), **short** (word), and **char** (byte). Single-precision, floating-point variables are declared with the keyword **float**. The following code segment shows how to declare some of the local variables used in Example 3–6.

```
.
.
.
main()
{
/* Declare local variables */
       int STATUS          /* STATUS returned by LIO routine calls */
           ,axv_id         /* LIO-assigned device ID */
           ,data_length    /* number of data bytes to read */
           ;
.
.
.
```

All VAX C character strings are passed by descriptor. You must declare the string descriptors for character-string values. For the purposes of string descriptor initialization, VAX C provides a simple preprocessor macro in the **#include** text library module *descrip*. This macro is named $DESCRIPTOR. It takes two arguments, which it uses in a standard VAX C structure declaration. The first argument is an identifier specifying the name of the descriptor to be declared and initialized. The second argument is a pointer to the data byte to be used as the value of the descriptor.

The following code segment shows how to declare the string descriptors for the character-string constant values used in Example 3–6.

```
#include descrip                        /*Define the $DESCRIPTOR macro*/
       .
       .
       .
$DESCRIPTOR(dev_type, "AXA0:") ;     /* AXV11-C device type */
$DESCRIPTOR(mode_string, "ISXY") ; /* LGP$PLOT mode string value */
$DESCRIPTOR(xlabel, "Time") ;        /* LGP$PLOT x-axis label */
$DESCRIPTOR(ylabel, "Voltage") ;     /* LGP$PLOT y-axis label */
$DESCRIPTOR(title, "C_EXAMPLE");     /* LGP$PLOT graph title */
```

The VSL application routines expect most arguments to be passed by reference. This means that the argument list contains the address of the argument rather than its value. In VAX C, you can use the ampersand (&) operator to pass an argument by reference, that is, the ampersand operator causes the argument's **address** to be passed. Note that an array name in an argument list always results in passing the address of the array.

3.2.3.3 Declaring and Dimensioning Arrays

VAX C arrays are data structures composed of identically typed members called elements. Declaring an array means to specify the name, data type, and maximum size of the array. VAX C supports both one-dimensional and multidimensional arrays. Subscripts follow the array name in square brackets and define an element's position in the array.

The following code segment shows how to declare the arrays used in Example 3–6.

```
/*
Declare data buffer for raw data in LSP$FORMAT_TRANSLATE_ADC. This
is a word (16-bit) array containing 100 elements.
*/
   short int raw_data[100];
/*
Declare data buffer for voltages in LSP$FORMAT_TRANSLATE_ADC and
LGP$PLOT routines.  This is a single-precision, floating-point
array containing 100 elements.
*/
    float voltages[100];
```

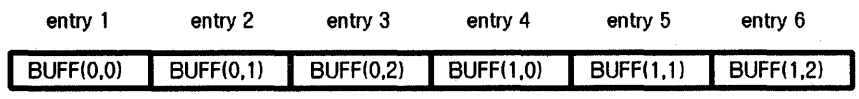
The first array, raw_data[100], is a word (16-bit) integer array of length 100 used to contain the 100 raw data values obtained from the analog-to- digital converter on the AXV11-C. The second array, voltages[100], is a single-precision, floating-point array of length 100 that is used to contain the 100 voltages obtained by converting the 100 raw data points to voltages using the LSP routine LSP$FORMAT_TRANSLATE_ADC.

Certain high-level languages have different ways of storing the values associated with two-dimensional arrays. You need to be aware of the way in which VAX C stores the values associated with two-dimensional arrays when you declare and dimension arrays in your programs. For example, suppose you need to declare a 2-by-3 two-dimensional array, called BUFF. You declare the array as follows:

```
int buff[2][3]
```

VAX C stores the values associated with the 2-by-3 two-dimensional array, called BUFF, as a linear one-dimensional array of length six with the storage allocated according to the following indexing system:

Figure 3–3: VAX C Array Indexing System



When using two-dimensional arrays in a VAX C VSL application program, the leftmost index usually references the buffer number. The rightmost index, which varies faster, can reference consecutive values read from an analog-to-digital converter, or values passed to a multiline plotting routine call.

3.2.3.4 Declaring External Routines

The symbolic definition file VSL.H defines the entry points containing the C interface language constructs for the VSL routines. Including this file in your VSL application programs ensures that the VSL routines are declared external and that their respective arguments are typed appropriately. See Section 3.2.3.1, Including Symbolic Definition Files, for information about including VSL.H, as well as other required symbolic definition files, in your VSL application programs.

3.2.3.5 Defaulting Routine Call Arguments

The reference descriptions of the VSL routines describe each routine's syntax and argument list in detail. Some VSL routine arguments are required. A required argument must always be included in the routine's argument list. Some VSL routine arguments are optional. Optional arguments can be included in the routine's argument list at the programmer's discretion. These arguments pass or return information that may or may not be useful to a particular application program. Some optional arguments are useful only with certain VSL devices.

Most VSL routine call arguments are assigned default values that are used if a user-supplied value is not included in a routine call argument list. To use a default value supplied by VSL, or to signal the omission of an optional argument in a routine call argument list, you must account for the argument in the routine call argument list by including a 0 in place of the actual argument, for example:

```
LGP$PLOT(&1, &mode_string, 0, voltages, &100, &xlabel, &ylabel, 0,
        &1, 0, 0, 0, 0, &title);
```

In the LGP$PLOT routine call argument list above, the arguments are passed as shown in the following table.

Argument	Value
ws_number	&1
mode_string	&mode_string
xarray	0[1]
yarray	voltages
n	&100
xlabel	&xlabel
ylabel	&ylabel
status	0[1]
iline	&1
igrid	0[1]
xcontrol	0[1]
ycontrol	0[1]
color	0[1]
title	&title
metaflag	Omitted
metafile_name	Omitted

[1]Uses the default value.

For LSP and LGP routine calls, you can omit optional arguments at the end of a routine call argument list. You can terminate the list after the last user-supplied value of an optional argument is specified.

For LIO routine calls, you must explicitly specify or explicitly default optional arguments regardless of the argument's place in the routine call argument list. You cannot terminate the list after the last user-supplied value of an optional argument is specified.

3.2.3.6 Checking Routine Call Status

You use the VMS Run-Time Library routine LIB$SIGNAL to signal the status of VSL routine calls. LIB$SIGNAL generates a signal indicating that an exception condition has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program exits.

The following program segment calls the LIO$READ routine as a function which returns the status of the operation in the variable **STATUS**.

```
STATUS = LIO$READ(&axv_id, raw_data, &200, &data_length, 0);
if (!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);
```

The value of **STATUS** is ANDed with the value STS$M_SUCCESS, which is a 1. If this operation produces a 1, then status is odd (bit zero set to one), and program execution continues. If this operation does not produce a 1, then status is even (bit zero set to zero), and the condition is signaled.

NOTE

When using the bitwise AND (&), as is the case here when checking routine call status, the value of both compared bits must be 1 for the result to be 1.

The error-handling mechanisms used by the LIO, LSP, and LGP routines are documented in the *Guide to the VAXlab Laboratory I/O Routines*, the *Guide to the VAXlab Laboratory Signal-Processing Routines*, and the *Guide to the VAXlab Laboratory Graphics Package*, respectively. See the appropriate document for complete information about the ways in which each VSL facility performs error handling. These documents also contain detailed information about the error codes returned by each facility and suggested user actions to recover from errors.

3.2.3.7 Using AST Routines

Before you attempt to set up and use AST routines within the context of your VSL applications, be familiar with the information about AST routines discussed in the *Guide to the VAXlab Laboratory I/O Routines*. Once you are familiar with that material, read the remainder of this section carefully to become familiar with the way in which you write your VAX C programs to include the use of AST routines.

Example 3–5 is an AST routine written in VAX C that receives completed buffers from a device, processes them, and requeues them to the device. See the online sample program LIO_ADV_AST.C for a complete VAX C VSL application program that uses this subroutine.

Example 3–5: An AST Routine Written in VAX C

```
main()       /* Main program sets up the AST routine */
             .
             .
             .
     /* Declare the external AST routine */
     int     adv_ast ();
             .
             .
             .
     /* Supply an AST routine to be called when a buffer is complete */
     status = LIO$SET_I(&device_id, &LIO$K_AST_RTN, &1, adv_ast);
     /* Enqueue the buffer.  This buffer will be passed to the
        AST routine when it completes. */
             .
             .
             .
     /*
       This subroutine receives completed buffers from the ADV11-D
       device. Each buffer is processed by the AST routine as it is
       received from the ADV11-D device and is then requeued to the
       device for further use. A total of five buffers are received
       and processed by this AST routine.
     */
     adv_ast(status_ptr, device_id_ptr, buffer, buffer_length_ptr,
             data_length_ptr, buffer_index_ptr, device_specific_ptr)
```

Example 3–5 Cont'd. on next page

Example 3–5 (Cont.): An AST Routine Written in VAX C

```
/*
  Note that the AST routine dummy arguments are similar to the
  arguments to the LIO$DEQUEUE routine.  These arguments are:
*/
 int *status_ptr           /* Returns the status of the I/O operation. */
     ,*device_id_ptr       /* Specifies the LIO-assigned device ID of
                              the ADV11-D. */
     ,buffer[]             /* The actual buffer, NOT the buffer address
                              as in the LIO$DEQUEUE routine call. */
     ,*buffer_length_ptr   /* The length of the buffer, in bytes.
     ,*data_length_ptr     /* The length of the data in the buffer,
                              in bytes. */
     ,*buffer_index_ptr    /* The buffer index, if one is supplied in the
                              LIO$ENQUEUE routine call. */
     ,*device_specific_ptr /* A dummy argument here.  The ADV11-D device
                              does not support this argument. */
     ;
{
/* Declare the common used to communication with the main program */
extern int
         done_flag        /* AST routine done flag */
         ,buffer_count    /* Numbers of buffer processed */
         ,ast_status      /* AST routine status */
         ;
/* execution */
/*
Save the I/O status where the main program can check if an
error condition occurs.
*/
ast_status = *status_ptr;

/* Increment the buffer count */

buffer_count++;

/*
 If the buffer count is less than 5, and if no error condition
 has occurred, requeue the buffer to the ADV11-D.  Otherwise,
 set the I/O done flag and exit.
*/
```

Example 3–5 Cont'd. on next page

Example 3–5 (Cont.): An AST Routine Written in VAX C

```
    if (buffer_count < 5 && (*status_ptr & STS$M_SUCCESS))
            {
            ast_status = LIO$ENQUEUE(device_id_ptr, buffer,
                         buffer_length_ptr, 0, 0, 0, 0);
            if (!(status & STS$M_SUCCESS)) LIB$SIGNAL(ast_status);
            }
    else
            done_flag = 1;
    }
```

Example 3–6: Sample VAX C Program Using the VSL Routines

```
/*
 C_EXAMPLE.C
   This program reads 100 values from channel 2 of the AXV11-C then
   displays the data in a graph on the screen.

   This is a simple application using the VSL libraries.

   This program can be compiled, linked, and run as follows:
        $ CC C_EXAMPLE
        $ LINK C_EXAMPLE, SYS$INPUT/OPT
        SYS$LIBRARY:VAXCRTL.EXE/SHARE
        <CTRL-Z>
        $ RUN C_EXAMPLE
*/

#include <lioset.h>              /* LIO set parameter definitions */
#include <vsl.h>                 /* VSL routine definitions */
#include descrip                 /* string descriptor definitions */
#include stsdef                  /* STATUS value bit definitions */

main()
{
/* Declare local variables */
       int STATUS         /* STATUS returned by LIO routine calls */
           ,axv_id        /* LIO-assigned device ID */
           ,data_length   /* number of data bytes to read */
           ;
```

Example 3–6 Cont'd. on next page

Example 3–6 (Cont.): Sample VAX C Program Using the VSL Routines

```
/* Declare the string descriptors for the string constants */
       $DESCRIPTOR(dev_type, "AXA0:") ;    /* AXV11-C device type */
       $DESCRIPTOR(mode_string, "ISXY") ; /* LGP$PLOT mode string value */
       $DESCRIPTOR(xlabel, "Time") ;       /* LGP$PLOT x-axis label */
       $DESCRIPTOR(ylabel, "Voltage") ;    /* LGP$PLOT y-axis label */
       $DESCRIPTOR(title, "C_EXAMPLE");    /* LGP$PLOT graph title */

/*
Declare data buffer for raw data in LSP$FORMAT_TRANSLATE_ADC. This
is a word (16-bit) array containing 100 elements.
*/
   short int raw_data[100];
/*
Declare data buffer for voltages in LSP$FORMAT_TRANSLATE_ADC and
LGP$PLOT routines.  This is a single-precision, floating-point
array containing 100 elements.
*/
    float voltages[100];

/* Program execution */

/* Set up the AXV11-C */

     printf("C_EXAMPLE, Read data, convert it, plot it\n\n");

/*
Attach the AXV11-C and set up for mapped (polled) I/O. This routine
call returns an LIO-assigned device ID for the device.
*/
        STATUS = LIO$ATTACH (&axv_id, &dev_type, &LIO$K_MAP);
                 if(!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);

/* Set up the AXV11-C to use the synchronous I/O interface. */

        STATUS = LIO$SET_I (&axv_id, &LIO$K_SYNCH, &0);
                 if(!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);

/* Set up AXV11-C channel 2 for input. */

        STATUS = LIO$SET_I (&axv_id, &LIO$K_AD_CHAN, &1, &2);
                 if(!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);

/* Set up a channel gain of 1. */

        STATUS = LIO$SET_I (&axv_id, &LIO$K_AD_GAIN, &1, &1);
                 if(!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);

/* Trigger on LIO$READ and fill buffer as fast as possible. */

        STATUS = LIO$SET_I (&axv_id, &LIO$K_TRIG, &1, &LIO$K_IMM_BURST);
                 if(!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);
```

Example 3–6 Cont'd. on next page

Example 3–6 (Cont.): Sample VAX C Program Using the VSL Routines

```
/*
Get a raw_data buffer of 100 values using LIO$READ to read the A/D values.
Please note that the length of raw_data is specified in bytes and is the
returned data_length.  Arrays are automatically passed by reference, other
arguments were previously specified to be passed by reference.  A zero
(passed by value) defaults an argument in the routine call argument list.
*/

STATUS = LIO$READ(&axv_id, raw_data, &200, &data_length, 0);
         if (!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);

/* Detach from the A/D. */
STATUS = LIO$DETACH(&axv_id, 0);
         if(!(STATUS & STS$M_SUCCESS)) LIB$SIGNAL(STATUS);

/* Convert raw data to voltages using LSP$FORMAT_TRANSLATE_ADC */
LSP$FORMAT_TRANSLATE_ADC(raw_data, voltages, &100, 0, 0, 0);

/*
Plot the data using LGP$PLOT to plot the voltages on the terminal screen.
*/
LGP$PLOT(&1, &mode_string, 0, voltages, &100, &xlabel, &ylabel, 0,
         &1, 0, 0, 0, &title);

/* Wait for a carriage return before deleting plot. */
        printf("Type carriage return to exit");
        while(getchar() != '\n');       /* loop till get carriage return */

/* Terminate the plot. */
LGP$TERMINATE_PLOT(&1);

}
```

3.2.4 Developing Programs in VAX FORTRAN

Be familiar with the information contained in the following sections before you begin developing VSL application programs using VAX FORTRAN. For further information about VAX FORTRAN programming concepts and techniques not covered in this guide, see *Programming in VAX FORTRAN*.

3.2.4.1 Including Symbolic Definition Files

The VSL symbolic definition files define the Laboratory I/O (LIO), Laboratory Signal-Processing (LSP), and Laboratory Graphics Package (LGP) error code symbols, the LGP plotting attribute symbols, the LIO set parameter code symbols, the LSP spectral window types, and the entry points containing language-specific interface constructs for the VSL routines. You need to include symbolic definition files in your VSL application programs so that these symbols can be recognized by the programming language you are using.

Table 3–4 lists the symbolic definition files provided for use with VAX FORTRAN.

Table 3–4: VAX FORTRAN Symbolic Definition Files

File Name	Defines:
LGPATTDEF.FOR	LGP plotting attribute symbols
LGPDEF.FOR	LGP error code symbols
LIOERRS.FOR	LIO error code symbols
LIOSET.FOR	LIO set parameter code symbols
LSPDEF.FOR	LSP error code symbols
LSPSET.FOR	LSP spectral window types
VSL.FOR	Entry points containing FORTRAN interface constructs for the VSL routines[1]

[1]Include this symbolic definition file in all VSL application routines written in VAX FORTRAN.

You use the following routine line to include a symbolic definition file in a user program:

```
INCLUDE 'SYS$LIBRARY:filename.FOR'
```

where

filename is one of the files listed in Table 3–4.

The files you must include in a user program depend on the VSL facilities the program is designed to use. The file VSL.FOR must be included in all VAX FORTRAN VSL application programs because it defines entry points containing VAX FORTRAN interface constructs for the VSL routines.

If your program is designed to use only the LIO routines, then you also need to include those files that define LIO symbolic values. If a program, such as the one presented in Example 3–8, uses routines from all the VSL facilities, then you need to include many of the files listed in Table 3–4. If your application programs use routines from all the VSL facilities, it is advisable to include all the files listed in Table 3–4.

3.2.4.2 Declaring Data Types and Data Items

All data in a VAX FORTRAN program has a specific **data type** that determines how many bits of storage to consider as a unit and how to interpret and manipulate the unit. In VAX FORTRAN, a numeric storage unit generally corresponds to four bytes of memory. In some cases, a numeric storage unit can correspond to two bytes of memory. A character storage unit corresponds to one byte of memory. **Data items** are named quantities whose values can change during program execution. Each data item name refers to a location in the program's storage area.

Some data items are used to pass information from a user program to a device that is to perform some function with the information. Other data items are used to return information from a device to a user program. The VSL application routine reference descriptions explain the functions, syntax, and appropriate usage of the VSL routines. Each routine reference description also explains the routine arguments, their data types, and whether an argument is used to pass information to a device, to return information from the device to the user program, or, in some cases, both.

The VSL application routines use BYTE, INTEGER, REAL*4, COMPLEX*8, and CHARACTER*len data types. The INTEGER data type can be broken down further into the INTEGER*2 (two-byte or word) and INTEGER*4 (four-byte or longword) quantities. The default is INTEGER*4. Single-precision, floating-point data items are declared as REAL*4. Complex single-precision, floating-point data items are declared as COMPLEX*8: a pair of REAL*4 values that represent a complex number. The first value represents the real part of that number, and the second represents the imaginary part of that number.[1]

[1] Complex numbers are used only with certain VSL signal-processing routines. See the *Guide to the VAXlab Signal-Processing Routines* for more information.

You can specify data-type defaults in a program module using data-type declaration statements to explicitly type data items. Explicit data typing makes programs easier to understand and maintain because the data type of all program values is explicitly spelled out in the program and is not as dependent upon compilation defaults that may change.

To explicitly type data items, you use declaration statements to specify the type, range, and precision of your program values, for example:

```
C Declare local data items:

        INTEGER*4 STATUS          !STATUS returned by LIO calls
        INTEGER*4 axv_id          !LIO-assigned device ID
        INTEGER*4 data_length     !Number of bytes of data to read
        BYTE      dummy           !Variable to accept carriage return
```

This program segment is excerpted from Example 3–8 and shows the explicitly typed data items used in that sample program.

3.2.4.3 Declaring and Dimensioning Arrays

An array is a group of contiguous storage locations associated with a single symbolic name, the array name. The individual storage locations, called array elements, are referred to by a subscript appended to the array name. A one-dimensional array, such as BUFFER(10), contains a single column of figures. An array containing more than one column of figures, such as BUFFER(10,10), is called a two-dimensional array. The array BUFFER(10) contains 10 array elements; the array BUFFER(10,10) contains 100 array elements.

Dimensioning an array means to specify the name, data type, and maximum size of the array. When programming in VAX FORTRAN, you can use data-type declaration statements, the DIMENSION statement, and the COMMON statement to declare arrays.[2] These statements contain array declarators that define the name of the array, the number of dimensions in the array, and the number of array elements in each dimension, for example:

```
C Declare data buffers:
          INTEGER*2 raw_data(100) !100 point raw data buffer
          REAL*4    voltages(100) !100 voltages to plot
```

[2] The nature of your application program may determine which method is most appropriate.

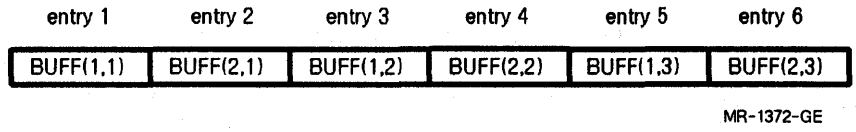
This program segment uses data-type declaration statements to dimension the two arrays that are used in Example 3–8. The first array, raw_data(100), is a word (two-byte) integer array of length 100 that is used to contain 100 raw data values obtained from the analog-to-digital converter on the AXV11-C. The second array, voltages(100), is a single-precision, floating-point (real) array of length 100 that is used to contain the 100 voltages obtained by converting the 100 raw data points to voltages using the LSP routine LSP$FORMAT_TRANSLATE_ADC.

Certain high-level languages have different ways of storing the values associated with two-dimensional arrays. You need to be aware of the way in which VAX FORTRAN stores the values associated with two-dimensional arrays when you declare and dimension arrays in your programs. For example, suppose you need to declare a 2-by-3 two-dimensional array, called BUFF. You declare the array as follows:

```
INTEGER*2 BUFF(3,2)
```

VAX FORTRAN stores the values associated with the 2-by-3 two-dimensional array, called BUFF, as a linear one-dimensional array of length six with the storage allocated according to the following indexing system:

Figure 3–4: VAX FORTRAN Array Indexing System



When using two-dimensional arrays in a VAX FORTRAN VSL application program, the leftmost index, which varies faster, can reference consecutive values read from an analog-to-digital converter, or values passed to a multiline plotting routine call. The rightmost index references the buffer number.

3.2.4.4 Declaring External Routines

The symbolic definition file VSL.FOR defines the entry points containing the FORTRAN interface language constructs for the VSL routines. Including this file in your VSL application programs ensures that the VSL routines are declared external and that their respective arguments are typed appropriately. See Section 3.2.4.1, Including Symbolic Definition Files, for information about including VSL.FOR, as well as other required symbolic definition files, in your VSL application programs.

3.2.4.5 Defaulting Routine Call Arguments

The reference descriptions of the VSL routines describe each routine's syntax and argument list in detail. Some VSL routine arguments are required. A required argument must always be included in the routine's argument list. Some VSL routine arguments are optional. Optional arguments can be included in the routine's argument list at the programmer's discretion. These arguments pass or return information that may or may not be useful to a particular application program. Some optional arguments are useful only with certain VSL devices.

Most VSL routine call arguments are assigned default values that are used if a user-supplied value is not included in a routine call argument list. To use a default value supplied by VSL, or to signal the omission of an optional argument in a routine call argument list, you must account for the argument in the routine call argument list by including a comma in place of the actual argument, for example:

```
   CALL LGP$PLOT(1, 'IXSY', , VOLTAGES, 100, 'Time', 'Voltage',
1  STATUS, 1, , , , , 'FORTRAN_EXAMPLE')
```

In the LGP$PLOT routine call argument list above, the arguments are passed as shown in the following table.

Argument	Value
ws_number	1
mode_string	'IXSY'
xarray	Comma[1]
yarray	VOLTAGES
n	100
xlabel	'Time'
ylabel	'Voltage'
status	STATUS
iline	1
igrid	Comma[1]
xcontrol	Comma[1]
ycontrol	Comma[1]
color	Comma[1]
title	'FORTRAN_EXAMPLE'
metaflag	Omitted
metafile_name	Omitted

[1]Uses default value.

For LSP and LGP routine calls, you can omit optional arguments at the end of a routine call argument list. You can terminate the list after the last user-supplied value of an optional argument is specified.

For LIO routine calls, you must explicitly specify or explicitly default optional arguments regardless of the argument's place in the routine call argument list. You cannot terminate the list after the last user-supplied value of an optional argument is specified.

3.2.4.6 Checking Routine Call Status

You use the VMS Run-Time Library routine LIB$SIGNAL to signal the status of VSL routine calls. LIB$SIGNAL generates a signal indicating that a exception condition has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program exits.

The following program segment calls the LIO$READ routine as a function which returns the status of the operation in the variable **STATUS**.

```
STATUS = LIO$READ(axv_id, raw_data, 200, data_length, )
        IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS)
```

The .NOT. logical operator tests whether **STATUS** is true or false. If the operation returns true, then **STATUS** is odd (bit zero set to one), and program execution continues. If the function returns false, then **STATUS** is even (bit zero set to zero), and the condition is signaled.

The error-handling mechanisms used by the LIO, LSP, and LGP routines are documented in the *Guide to the VAXlab Laboratory I/O Routines*, the *Guide to the VAXlab Laboratory Signal-Processing Routines*, and the *Guide to the VAXlab Laboratory Graphics Package*, respectively. See the appropriate document for complete information about the ways in which each VSL facility performs error handling. These documents also contain detailed information about the error codes returned by each facility and suggested user actions to recover from errors.

3.2.4.7 Using AST Routines

Before you attempt to set up and use AST routines within the context of your VSL applications, be familiar with the information about AST routines discussed in the *Guide to the VAXlab Laboratory I/O Routines*. Once you are familiar with that material, read the remainder of this section carefully to become familiar with the way in which you write your VAX FORTRAN programs to include the use of AST routines.

Example 3-7 is an AST routine written in VAX FORTRAN that receives completed buffers from a device, processes them, and requeues them to the device. See the online sample program LIO_ADV_AST.FOR for a complete VAX FORTRAN VSL application program that uses this subroutine.

Example 3–7: An AST Routine Written in VAX FORTRAN

```
      PROGRAM MAIN_PROGRAM
C Declare COMMON definitions

      COMMON /ast/done_flag,buffer_count,ast_status
      LOGICAL * 1 done_flag         ! Set to .TRUE. when done
      INTEGER     buffer_count      ! Increment on each buffer until 5
      INTEGER     ast_status        ! AST routine status
            .
            .
            .

      EXTERNAL ADV_AST          ! Declare the AST routine as external

            .
            .
            .

C Supply an AST routine to be called when a buffer is complete

      status = LIO$SET_I(device_id, LIO$K_AST_RTN, 1, ADV_AST)
            .
            .
            .

C This subroutine receives completed buffers from the ADV11-D
C device. Each buffer is processed by the AST routine as it is
C received from the ADV11-D device and is then requeued to the
C device for further use. A total of five buffers are received
C and processed by this AST routine.

       SUBROUTINE ADV_AST(status, device_id, buffer, buffer_length,
  1               data_length, buffer_index,  device_specific)

C Note that the AST routine dummy arguments are similar to the
C arguments to the LIO$DEQUEUE routine.  These arguments are:
C
       status             ! Returns the status of the I/O operation.
       device_id          ! Specifies the LIO-assigned device ID of
                          !   the ADV11-D.
       buffer             ! The actual buffer, NOT the buffer address
                          !   as in the LIO$DEQUEUE routine call.
       buffer_length      ! The length of the buffer, in bytes.
       data_length        ! The length of the data in the buffer, in bytes.
       buffer_index       ! The buffer index, if one is supplied in the
                          !   LIO$ENQUEUE routine call.
      device_specific     ! A dummy argument here.  The ADV11-D device
                          !   does not support a device-specific argument.
```

Example 3–7 Cont'd. on next page

Example 3-7 (Cont.): An AST Routine Written in VAX FORTRAN

```
C Declare the common used to communicate with the main program.
C These COMMON definitions must also be declared in the main program.

      COMMON /ast/done_flag,buffer_count,ast_status
      LOGICAL * 1 done_flag        ! Set to .TRUE. when done
      INTEGER     buffer_count     ! Increment on each buffer until 5
      INTEGER     ast_status       ! AST routine status

C Save the I/O status where the main program can check if an
C error condition occurs.

      ast_status = status

C Increment the buffer count

      buffer_count = buffer_count + 1

C If the buffer count is less than 5, and if no error condition
C has occured, requeue the buffer to the ADV11-D.  Otherwise,
C set the I/O done flag and exit.

     IF ((buffer_count .LT. 5) .AND. (status .AND. 1) THEN
            status = LIO$ENQUEUE(device_id, buffer, buffer_length, , , ,)
               IF (.NOT. status)  CALL LIB$SIGNAL(%VAL(status)
     ELSE
            done_flag = .TRUE.
     END IF

     RETURN
     END
```

Example 3–8: Sample VAX FORTRAN Program Using the VSL Routines

```
        PROGRAM FORTRAN_EXAMPLE
C This program reads 100 values from channel 2 of the AXV11-C then
C displays the data in a graph on the screen.
C
C This is a simple application using the VSL libraries.
C
C This program can be compiled, linked, and run as follows:
C       FORTRAN FORTRAN_EXAMPLE
C       LINK FORTRAN_EXAMPLE
C       RUN FORTRAN_EXAMPLE
C

        INCLUDE 'SYS$LIBRARY:LIOSET.FOR'
        INCLUDE 'SYS$LIBRARY:LIOERRS.FOR'
        INCLUDE 'SYS$LIBRARY:LSPDEF.FOR'
        INCLUDE 'SYS$LIBRARY:LGPDEF.FOR'
        INCLUDE 'SYS$LIBRARY:VSL.FOR'

C Declare local data items:

        INTEGER*4 STATUS          !STATUS returned by LIO calls
        INTEGER*4 axv_id          !LIO-assigned device ID
        INTEGER*4 data_length     !Number of data bytes to read
        BYTE      dummy           !Variable to accept carriage return

C Declare data buffers:

        INTEGER*2 raw_data(100) !100 point raw data buffer
        REAL      voltages(100) !100 voltages to plot

C Type pretty message
C
        TYPE *,'FORTRAN_EXAMPLE, Read data, convert it, plot it'
        TYPE *

C Attach the AXV11-C to use mapped (polled) I/O. This routine
C returns the LIO-assigned device ID for the device.
C
        STATUS = LIO$ATTACH (axv_id, 'AXAO', LIO$K_MAP)
                IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Set up the AXV11-C to use the synchronous I/O interface:
C
        STATUS = LIO$SET_I (axv_id, LIO$K_SYNCH, 0)
                IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))
```

Example 3–8 Cont'd. on next page

Example 3–8 (Cont.): Sample VAX FORTRAN Program Using the VSL Routines

```
C Set up AXV11-C channel 2 for input:
C
        STATUS = LIO$SET_I (axv_id, LIO$K_AD_CHAN, 1, 2)
                IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Set up a channel gain of 1:
C
        STATUS = LIO$SET_I (axv_id, LIO$K_AD_GAIN, 1, 1)
                IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Trigger on LIO$READ and fill buffer as fast as possible:
C
        STATUS = LIO$SET_I (axv_id, LIO$K_TRIG, 1, LIO$K_INN_BURST)
                IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Get a raw_data buffer of 100 values:
C This program uses LIO$READ to read the 100 A/D values.
C Note that the length of the buffer is in bytes, as is the
C returned data_length.
C
        STATUS = LIO$READ(axv_id, raw_data, 200, data_length, )
                 IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Detach from the A/D:
C
        STATUS = LIO$DETACH(axv_id, )
                 IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Convert the raw data to voltages using LSP$FORMAT_TRANSLATE_ADC:
C
        CALL LSP$FORMAT_TRANSLATE_ADC(raw_data, VOLTAGES, 100, , , )
             IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Plot the data using LGP$PLOT to plot the voltages on the
C terminal screen:
C
        CALL LGP$PLOT(1, 'IXSY', , VOLTAGES, 100, 'Time', 'Voltage',
     1  STATUS, 1, , , , 'FORTRAN_EXAMPLE')
             IF(.NOT. STATUS) CALL LIB$SIGNAL(%VAL(STATUS))

C Wait for a carriage return before deleting plot:
C
        TYPE 1000
1000    FORMAT(1X,'Type carriage return to exit'$)
        ACCEPT 1010,dummy
1010    FORMAT(A1)
```

Example 3–8 Cont'd. on next page

Example 3–8 (Cont.): Sample VAX FORTRAN Program Using the VSL Routines

```
C Terminate the plot:
C
        CALL LGP$TERMINATE_PLOT(1)

        STOP 'Done'
        END
```

3.2.5 Developing Programs in VAX PASCAL

Be familiar with the information contained in the following sections before you begin developing VSL application programs using VAX PASCAL. For further information about VAX PASCAL programming concepts and techniques not covered in this guide, see the *VAX PASCAL Reference Manual.*

3.2.5.1 Including Symbolic Definition Files

The VSL symbolic definition files define the Laboratory I/O (LIO), Laboratory Signal-Processing (LSP), and Laboratory Graphics Package (LGP) error code symbols, the LGP plotting attribute symbols, the LIO set parameter code symbols, the LSP spectral window types, and the entry points containing language-specific interface constructs for the VSL routines. You need to include symbolic definition files in your VSL application programs so that these symbols can be recognized by the programming language you are using.

Table 3–5 lists the symbolic definition files provided for use with VAX PASCAL

Table 3–5: VAX PASCAL Symbolic Definition Files

File Name	Defines:
LGPATTDEF.PAS	LGP plotting attribute symbols
LGPDEF.PAS	LGP error code symbols
LIOERRS.PAS	LIO error code symbols
LIOSET.PAS	LIO set parameter code symbols
LSPDEF.PAS	LSP error code symbols
LSPSET.PAS	LSP spectral window types
VSL.PAS	Entry points containing PASCAL interface constructs for the VSL routines[1]

[1]Include this symbolic definition file in all VSL application routines written in VAX PASCAL.

Before the files listed in Table 3–5 can be inherited into a PASCAL program, they must be compiled into environment files. Take the following steps to create the appropriate environment files:

1. Log in to a privileged account, such as the system manager's account.
2. Set default to SYS$LIBRARY.
3. Enter the following command lines:

```
$ PASCAL/ENVIRONMENT/NOOBJECT LIOERRS, LIOSET
$ PASCAL/ENVIRONMENT/NOOBJECT LSPDEF, LSPSET
$ PASCAL/ENVIRONMENT/NOOBJECT LGPDEF, LGPATTDEF
$ PASCAL/ENVIRONMENT/NOOBJECT VSL
$ SET PROT=(W:RE) *.PEN
```

   The PASCAL compiler creates an environment file (filename.PEN) for each of these symbolic definition files.

Then, when you code an application program using PASCAL, you use the following routine line to include a symbolic definition (environment) file in a user program:

```
[INHERIT ('SYS$LIBRARY:filename')]
```

where

filename is one of the files listed in Table 3–5.

The files you must include in a user program depend on the VSL facilities the program is designed to use. The file VSL.PAS must be included in all VAX PASCAL VSL application programs because it defines entry points containing VAX PASCAL interface constructs for the VSL routines.

If your program is designed to use only the LIO routines, then you also need to include those files that define LIO symbolic values. If a program, such as the one presented in Example 3–10, uses routines from all the VSL facilities, then you need to include many of the files listed in Table 3–5. If your programs use routines from all the VSL facilities, it is advisable to include all the files listed in Table 3–5.

3.2.5.2 Declaring Data Types and Data Items

Every PASCAL data item is associated with a particular data type. A data type is a set of values which share certain characteristics. A data type detemines both the range of values a data item can assume and the operations that can be performed on it. In addition, the type determines the storage space required for all the data item's possible values.

Some data items are used to pass information from a user program to a device that is to perform some function with the information. Other data items are used to return information from a device to a user program. The VSL application routine reference descriptions explain the functions, syntax, and appropriate usage of the VSL routines. Each routine reference description also explains the routine arguments, their data types, and whether an argument is used to pass information to a device, to return information from the device to the user program, or, in some cases, both.

The VSL application routines use WORD, INTEGER, REAL, and PACKED ARRAY [] OF CHAR data types. A fixed-length character string is defined as a packed array of characters with a lower bound of 1. The length of the string is established by the array's upper bound.

To explicitly type data items, you use a declarative statement to specify the type, range, and precision of your program values. The following program segment shows how to define some of the data types used in Example 3–10.

```
{ Define data types }
TYPE
{ Define devspec argument of the LIO$ATTACH routine as a character
  string a maximum of 2 characters in length }

        name_string = PACKED ARRAY [1..2] OF CHAR;  { 'AX' }

{ Define the range of the A/D data - 12-bit A/D }

        AtoD_data = -2048..2047 { 12-bit A/D }

{ Define the A/D buffer as a 100-element word array of data in the
  range of -2048 to 2047. }

        AtoD_buffer = ARRAY [1..100] OF [WORD] AtoD_data; {A/D raw data }

{ Define the voltage array as a 100-element single-precision,
  floating-point (real) array }

        plot_buffer = ARRAY [1..100] OF REAL; { data in volts }

                .
                .
                .

{ Define the title argument of the LGP$PLOT routine as a character
  string a maximum of 14 characters in length }

        title_string = PACKED ARRAY [1..14] OF CHAR; { 'PASCAL_EXAMPLE' }


{ Define local variables }

VAR     STATUS : INTEGER;       { STATUS returned by LIO calls }
        axv_id : INTEGER;       { LIO-assigned device ID }
        data_length : INTEGER; { Number of data bytes to read }
```

3.2.5.3 Declaring and Dimensioning Arrays

An array is a group of components in which all elements have the same data type and share a common identifier. An individual element of an array is referred to by an integer index, or subscript, that designates the element's position or order in the array.

The definition of an array type specifies its dimensions, the bounds of each dimension, and the types of its indexes and components. The arrays used in Example 3–10 were **typed** in the previous section. The following program segment shows how to **declare** the arrays used in Example 3–10. The actual allocation or dimensioning of the arrays does not take place until they are declared as shown below.

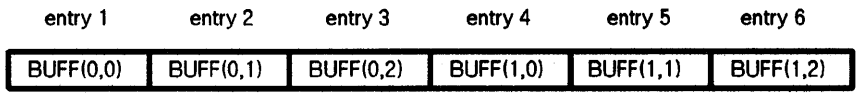
```
{ Declare data buffers }

        raw_data : AtoD_buffer;   { 100 point raw data buffer }
        voltage : plot_buffer;    { 100 voltages to plot }
```

Certain high-level languages have different ways of storing the values associated with two-dimensional arrays. You need to be aware of the way in which VAX PASCAL stores the values associated with two-dimensional arrays when you declare and dimension arrays in your programs. For example, suppose you need to declare a 2-by-3 two-dimensional array, called BUFF. You declare the array as follows:

```
buff = array [0..1] of array [0..2] of [word] AtoD_data;
```

VAX PASCAL stores the values associated with the 2-by-3 two-dimensional array, called buff, as a linear one-dimensional array of length six with the storage allocated according to the following indexing system:

Figure 3–5: VAX PASCAL Array Indexing System



MR-1373-GE

When using two-dimensional arrays in a VAX PASCAL VSL application program, the leftmost index usually references the buffer number. The rightmost index, which varies faster, can reference consecutive values read from an analog-to-digital converter, or values passed to a multiline plotting routine call.

3.2.5.4 Declaring External Procedures and Functions

The symbolic definition file VSL.PAS defines the entry points containing the PASCAL interface language constructs for the VSL routines. Including this file in your VSL application programs ensures that the VSL routines are declared external and that their respective arguments are typed appropriately. See Section 3.2.5.1, Including Symbolic Definition Files, for information about including VSL.PAS, as well as other required symbolic definition files, in your VSL application programs.

If your VSL application programs use VMS library routines, such as LIB$SIGNAL, these routines must be declared external with the program context. The following program segment, excerpted from Example 3–10 shows how to declare an external routine as a function within the program context.

```
{Define VMS library routine}

[EXTERNAL] FUNCTION LIB$SIGNAL
        (
         %IMMED STATUS : INTEGER
        ) : INTEGER; EXTERNAL;
```

See Section 3.2.5.6, Checking Routine Call Status, for information about how the LIB$SIGNAL routine is used to check routine call status.

3.2.5.5 Defaulting Routine Call Arguments

The reference descriptions of the VSL routines describe each routine's syntax and argument list in detail. Some VSL routine arguments are required. A required argument must always be included in the routine's argument list. Some VSL routine arguments are optional. Optional arguments can be included in the routine's argument list at the programmer's discretion. These arguments pass or return information that may or may not be useful to a particular application program. Some optional arguments are useful only with certain VSL devices.

Most VSL routine call arguments are assigned default values that are used if a user-supplied value is not included in a routine call argument list. To use a default value supplied by VSL, or to signal the omission of an optional argument in a routine call argument list, you must account for the argument in the routine call argument list by including a comma in place of the actual argument. To use routine argument default values, you need to declare the arguments you want to default using the %IMMED foreign mechanism specifier. See Section 3.2.5.4, Declaring External Procedures and Functions for further information.

```
LGP$PLOT (1, 'IXSY', , VOLTAGES, 100, 'Time', 'Voltage', STATUS,
          1, , , , , 'PASCAL_EXAMPLE');
```

In the LGP$PLOT routine call argument list above, the arguments are passed as shown in the following table.

Argument	Value
ws_number	1
mode_string	'IXSY'
xarray	Comma[1]
yarray	VOLTAGES
n	100
xlabel	'Time'
ylabel	'Voltage'
status	STATUS
iline	1
igrid	Comma[1]
xcontrol	Comma[1]
ycontrol	Comma[1]
color	Comma[1]
title	'PASCAL_EXAMPLE'
metaflag	Omitted
metafile_name	Omitted

[1]Uses default value.

For LSP and LGP routine calls, you can omit optional arguments at the end of a routine call argument list. You can terminate the list after the last user-supplied value of an optional argument is specified.

For LIO routine calls, you must explicitly specify or explicitly default optional arguments regardless of the argument's place in the routine call argument list. You cannot terminate the list after the last user-supplied value of an optional argument is specified.

3.2.5.6 Checking Routine Call Status

You use the VMS Run-Time Library routine LIB$SIGNAL to signal the status of VSL routine calls. LIB$SIGNAL generates a signal indicating that an exception has occurred in your program. If a condition handler does not take corrective action and the condition is severe, then your program exits.

The following program segment calls the LIO$READ routine as a function which returns the status of the operation in the variable **STATUS**.

```
STATUS := LIO$READ(axv_id, raw_data, 200, data_length, );
          IF NOT ODD (STATUS) THEN LIB$SIGNAL (STATUS);
```

The NOT ODD function tests whether the low bit of **STATUS** is odd or even. If the function returns true, then **STATUS** is odd (bit zero set to one), and program execution continues. If the function returns false, then **STATUS** is even (bit zero set to one), and the condition is signaled.

The error-handling mechanisms used by the LIO, LSP, and LGP routines are documented in the *Guide to the VAXlab Laboratory I/O Routines*, the *Guide to the VAXlab Laboratory Signal-Processing Routines*, and the *Guide to the VAXlab Laboratory Graphics Package*, respectively. See the appropriate document for complete information about the ways in which each VSL facility performs error handling. These documents also contain detailed information about the error codes returned by each facility and suggested user actions to recover from errors.

3.2.5.7 Using AST Routines

Before you attempt to set up and use AST routines within the context of your VSL applications, be familiar with the information about AST routines discussed in the *Guide to the VAXlab Laboratory I/O Routines*. Once you are familiar with that material, read the remainder of this section carefully to become familiar with the way in which you write your VAX PASCAL programs to include the use of AST routines.

Example 3-9 segment is an AST routine written in VAX PASCAL that receives completed buffers from a device, processes them, and requeues them to the device. See the online sample program LIO_ADV_AST.PAS for a complete VAX PASCAL VSL application program that uses this subroutine.

Example 3-9: An AST Routine Written in VAX PASCAL

```
            .
            .
            .
{Define the AST Routine }

[ASYNCHRONOUS, UNBOUND] PROCEDURE ADV_AST(VAR status, device_id : INTEGER;
       buffer: AtoD_buffer; buffer_length, data_length, buffer_index,
       device_specific : INTEGER);

{
This subroutine receives completed buffers from the ADV11-D device.
Each buffer is processed by the AST routine as it is received from the
ADV11-D device and is then requeued to the device for further use. A
total of five buffers are received and processed by this AST routine.
This AST routine must be declared ASYNCHRONOUS and UNBOUND.

Note that the AST routine dummy arguments are similar to the
arguments to the LIO$DEQUEUE routine.  These arguments are:

        status              Returns the status of the I/O operation.
        device_id           Specifies the LIO-assigned device ID of
                              the ADV11-D.
        buffer              The actual buffer, NOT the buffer address
                              as in the LIO$DEQUEUE routine call.
        buffer_length       The length of the buffer, in bytes.
        data_length         The length of the data in the buffer, in bytes.
        buffer_index        The buffer index, if one is supplied in the
                              LIO$ENQUEUE routine call.
        device_specific     A dummy argument here.  The ADV11-D device
                              does not support a device-specific argument.

The following variables are declared in the main routine and are
referenced here as globals:

       done_flag : BOOLEAN        AST routine done flag
       buffer_count : INTEGER     Number of buffers
       ast_status : INTEGER       AST routine status
}
```

Example 3-9 Cont'd. on next page

Example 3–9 (Cont.): An AST Routine Written in VAX PASCAL

```
BEGIN

      {
      Save the I/O status where the main program can check if an
      error condition occurs.
      }
      ast_status := status;

      {
      Increment the buffer count
      }
      buffer_count := buffer_count + 1;

      {
      If the buffer count is less than 5, and if no error condition
      has occured, requeue the buffer to the ADV11-D.  Otherwise,
      set the I/O done flag and exit.
      }

      IF((buffer_count < 5) AND (ODD (status))) THEN
          BEGIN
            ast_status = LIO$ENQUEUE(device_id, buffer, buffer_length, , , ,)
            IF NOT ODD (ast_status) THEN LIB$SIGNAL (ast_status)
          END
      ELSE
            done_flag := TRUE;
      END;

{ Define the routine that sets up the AST routine }

FUNCTION SET_AST(device_id : INTEGER;
         [UNBOUND] PROCEDURE ADV_AST(VAR status, device_id : INTEGER;
           buffer : AtoD_buffer, buffer_length, data_length, buffer_index,
           device_specific : INTEGER)) : INTEGER;
{
This function exists purely to get around PASCAL's strong typing rules.
Its purpose it to create a version of the LIO$SET_I routine call that
sets up the AST routine.  The LIO$SET_I called is declared to accept
integer arguments (this declaration is made in the include file VSL.PAS).
This use of the LIO$SET_I routine call requires it to accept a routine as
an argument, so you need a special declaration of the LIO$SET_I routine
to enable it.
}
```

Example 3–9 Cont'd. on next page

Example 3–9 (Cont.): An AST Routine Written in VAX PASCAL

```
{ Declare special version of the LIO$SET_I routine call }
[EXTERNAL] FUNCTION LIO$SET_I
          (
          %REF device_id : INTEGER;      { LIO-assigned device ID }
          %REF param_code : INTEGER;     { Set parameter code }
          %REF param_val : INTEGER;      { Set parameter code value }
{ AST routine }
          %IMMED [UNBOUND] PROCEDURE ADV_AST(VAR status, device_id : INTEGER,
                 buffer : AtoD_buffer, buffer_length, data_length,
                 buffer_index, device_specific : INTEGER)
          ) : INTEGER; EXTERNAL;

{ Supply AST routine }

BEGIN
          set_ast := LIO$SET_I(device_id, LIO$K_AST_RTN, 1, ADV_AST)
END;
           .
           .
           .
```

Example 3–10: Sample VAX PASCAL Program Using the VSL Routines

```
[INHERIT ('SYS$LIBRARY:LIOSET',   { Defines LIO set parameter codes }
          'SYS$LIBRARY:LIOERRS',  { Defines LIO error codes }
          'SYS$LIBRARY:LSPDEF',   { Defines LSP error codes }
          'SYS$LIBRARY:LGPDEF',   { Defines LGP error codes }
          'SYS$LIBRARY:VSL')]     { Defines VSL routines }

PROGRAM PASCAL_EXAMPLE (INPUT, OUTPUT);
{
 This program reads 100 values from channel 2 of the AXV11-C then
 displays the data in a graph on the screen.

 This is a simple application using the VSL libraries.

 This program can be compiled, linked, and run as follows:
        PASCAL PASCAL_EXAMPLE
        LINK PASCAL_EXAMPLE
        RUN PASCAL_EXAMPLE
}
```

Example 3–10 Cont'd. on next page

Example 3–10 (Cont.): Sample VAX PASCAL Program Using the VSL Routines

```
{ Define data types }

TYPE
{ Define devspec argument of the LIO$ATTACH routine as a character
  string a maximum of 2 characters in length }

        name_string = PACKED ARRAY [1..2] OF CHAR;  { 'AX' }

{ Define the range of the A/D data - 12-bit A/D }

        AtoD_data = -2048..2047 { 12-bit A/D }

{ Define the A/D buffer as a 100-element word array of data in the
  range of -2048 to 2047. }

        AtoD_buffer = ARRAY [1..100] OF [WORD] AtoD_data; {A/D raw data }
{ Define the voltage array as a 100-element single-precision,
  floating-point (real) array }

        plot_buffer = ARRAY [1..100] OF REAL; { data in volts }

{ Define the range array as a 2-element single-precision,
  floating point (real) array }

        range_array = ARRAY [1..2] OF REAL;

{ Define the mode_string argument of the LGP$PLOT routine as a
  character string a maximum of 4 characters in length }

        mode_string = PACKED ARRAY [1..4] OF CHAR; { 'IXSY' }

{ Define the xlabel argument of the LGP$PLOT routine as a character
  string a maximum of 4 characters in length }

        xlabel_string = PACKED ARRAY [1..4] OF CHAR; { 'Time' }

{ Define the ylabel argument of the LGP$PLOT routine as a character
  string a maximum of 7 characters in length }

        ylabel_string = PACKED ARRAY [1..7] OF CHAR; { 'Voltage' }

{ Define the title argument of the LGP$PLOT routine as a character
  string a maximum of 14 characters in length }

        title_string = PACKED ARRAY [1..14] OF CHAR; { 'PASCAL_EXAMPLE' }
```

Example 3–10 Cont'd. on next page

Example 3–10 (Cont.): Sample VAX PASCAL Program Using the VSL Routines

```
{ Define local variables }

VAR     STATUS : INTEGER;       { STATUS returned by LIO calls }
        axv_id : INTEGER;       { LIO-assigned device ID }
        data_length : INTEGER; { Number of data bytes to read }

{ Declare data buffers }

        raw_data : AtoD_buffer;         { 100 point raw data buffer }
        voltages : plot_buffer;         { 100 voltages to plot }


{Define VMS library routines}

[EXTERNAL] FUNCTION LIB$SIGNAL  { Define LIB$SIGNAL as an external function }
        (
         %IMMED STATUS : INTEGER     { %IMMED defaults STATUS argument }
        ) : INTEGER; EXTERNAL;
BEGIN
{ Type pretty message }
        WRITELN('PASCAL_EXAMPLE, Read data, convert it, plot it');
        WRITELN;

{ Attach the AXV11-C to use mapped (polled) I/O.  This routine
  returns the LIO-assigned device ID for the device. }

        STATUS := LIO$ATTACH (axv_id, 'AX', LIO$K_MAP);
                IF NOT ODD (STATUS) THEN LIB$SIGNAL(STATUS);

{ Set up the AXV11-C to use the synchronous I/O interface: }

        STATUS := LIO$SET_I (axv_id, LIO$K_SYNCH, 0);
                IF NOT ODD (STATUS) THEN LIB$SIGNAL( STATUS );

{ Set up AXV11-C channel 2 for input: }

        STATUS := LIO$SET_I (axv_id, LIO$K_AD_CHAN, 1, 2);
                IF NOT ODD (STATUS) THEN LIB$SIGNAL( STATUS );

{ Set up a channel gain of 1: }

        STATUS := LIO$SET_I (axv_id, LIO$K_AD_GAIN, 1, 1);
                IF NOT ODD (STATUS) THEN LIB$SIGNAL( STATUS );

{ Trigger on LIO$READ and fill buffer as fast as possible: }

        STATUS := LIO$SET_I (axv_id, LIO$K_TRIG, 1, LIO$K_IMM_BURST);
                IF NOT ODD (STATUS) THEN LIB$SIGNAL( STATUS );
```

Example 3–10 Cont'd. on next page

Example 3–10 (Cont.): Sample VAX PASCAL Program Using the VSL Routines

```
{ Get a raw_data buffer of 100 values.  Use LIO$READ to read the 100
  A/D values.  Note that the length of the buffer is specified in
  bytes as is the returned data length. }

        STATUS := LIO$READ(axv_id, raw_data, 200, data_length, );
                  IF NOT ODD (STATUS) THEN LIB$SIGNAL( STATUS );

{ Detach from the A/D }

        STATUS := LIO$DETACH(axv_id, 0);
                  IF NOT ODD (STATUS) THEN LIB$SIGNAL( STATUS );

{ Convert the raw data to voltages using LSP$FORMAT_TRANSLATE_ADC }

        LSP$FORMAT_TRANSLATE_ADC(raw_data, voltages, 100, , , );

{ Plot the data using LGP$PLOT to plot the voltages on the
  terminal screen. }

        LGP$PLOT(1, 'IXSY', , voltages, 100, 'Time', 'Voltage', , 1, , , , ,
                 'PASCAL_EXAMPLE');
{ Wait for a carriage return before deleting plot. }

        WRITE('Type carriage return to exit');
        READLN;

{ Terminate the plot. }

        LGP$TERMINATE_PLOT(1);
END.
```

3.3 Accessing the VSL Sample Programs

The VSL sample programs are shipped with the VSL software and are put on line during the VSL installation procedure in a directory with the logical name VSL$EXAMPLES.

To copy a sample program file, in this case ADA_EXAMPLE.ADA, to your directory, enter the following command line:

```
$ COPY VSL$EXAMPLES:ADA_EXAMPLE.ADA *.* RETURN
```

Once you copy a sample file to your directory, follow the procedures for compiling, linking, and running the program.

The *Guide to the VAXlab Laboratory I/O Routines*, the *Guide to the VAXlab Signal-Processing Routines*, and the *Guide to the VAXlab Laboratory Graphics Package* each contains several sample programs showing how to use the routines associated with that VSL facility to acquire, process, and plot data. These programs, as well as many other sample programs which do not appear in the hardcopy documentation, are also shipped with the VSL software, and are put on line during the VSL installation procedure.

Table 3–6 lists the sample program directories created during the installation procedure and the contents of each directory. The logical names of these directories are also defined during the installation procedure by commands in the appropriate startup command file.

Table 3–6: VSL Online Sample Program Directories

Directory	Contains:
LIO$EXAMPLES	LIO sample programs
LSP$EXAMPLES	LSP sample programs
LGP$EXAMPLES	LGP sample programs

See the *Guide to the VAXlab Laboratory I/O Routines* for a complete description of the sample programs contained in the LIO$EXAMPLES directory.

See the *Guide to the VAXlab Laboratory Signal-Processing Routines* for a complete description of the sample programs contained in the LSP$EXAMPLES directory.

See the *Guide to the VAXlab Laboratory Graphics Package* for a complete description of the sample programs contained in the LGP$EXAMPLES directory.

Index

A

B

C

D

E

F

I

L

M

P

Q

R

S

T

V

READER'S COMMENTS

Your comments and suggestions help us to improve the quality of our publications.

For which tasks did you use this manual? (Circle your responses.)

(a) Installation (c) Maintenance (e) Training
(b) Operation/use (d) Programming (f) Other (Please specify.) _____

Did the manual meet your needs? Yes ☐ No ☐ Why? _____

Please rate the manual in the following categories. (Circle your responses.)

	Excellent	Good	Fair	Poor	Unacceptable
Accuracy (product works as described)	5	4	3	2	1
Clarity (easy to understand)	5	4	3	2	1
Completeness (enough information)	5	4	3	2	1
Organization (structure of subject matter)	5	4	3	2	1
Table of Contents, Index (ability to find topic)	5	4	3	2	1
Illustrations, examples (useful)	5	4	3	2	1
Overall ease of use	5	4	3	2	1
Page Layout (easy to find information)	5	4	3	2	1
Print Quality (easy to read)	5	4	3	2	1

What things did you like *most* about this manual? _____

What things did you like *least* about this manual? _____

Please list and describe any errors you found in the manual.

Page	Description/Location of Error
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions for improving this manual: _____

Name _____ Job Title _____
Street _____ Company _____
City _____ Department _____
State/Country _____ Telephone Number _____
Postal (ZIP) Code _____ Date _____

Fold Here and Tape

Affix
Stamp
Here

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
200 FOREST STREET MR01-2/L12
MARLBOROUGH, MA 01752-9101

Fold Here