VAX LISP/VMS System Access Guide

Order Number: AA–GH75B–TE

July 1989

This guide describes VAX LISP facilities for interacting with the programming interface to the VMS operating system. Formal definitions of functions and macros introduced in this guide are contained in the *VAX LISP/VMS Object Reference Manual.*

Revision/Update Information: This manual supersedes the *VAX LISP/VMS System Access Programming Guide*, AA–GH75A–TE, for VAX LISP/VMS Version 2.0.

Operating System and Version: VMS Version 5.1

Software Version: VAX LISP Version 3.0

digital equipment corporation
maynard, massachusetts

First Printing, May 1986
Updated, July 1987
Revised, July 1989

AI VAXstation	PDP	VAX LISP/ULTRIX
DEC	ULTRIX	VAX LISP/VMS
DECnet	ULTRIX–11	VAXstation
DECUS	ULTRIX–32	VAXstation II
MicroVAX	UNIBUS	VMS
MicroVAX II	VAX	digital™
MicroVMS	VAX LISP	

S833

This document was prepared using VAX DOCUMENT, Version 1.1.

Contents

Preface

The *VAX LISP / VMS System Access Guide* provides information that lets you, as a LISP programmer, make use of the programming interface of the VMS operating system. The routines included with the operating system give you access to capabilities not normally accessible from the LISP environment.

Intended Audience

This guide is intended for programmers with a good knowledge of both LISP and the programming interface to the VMS operating system.

Structure

An outline of the organization and chapter content of this guide follows:

- Chapter 1 provides an overview of the VAX LISP system access facilities.
- Chapter 2 explains how to use file specifications in LISP.
- Chapter 3 describes how to get information about the current state of the operating system and the process running LISP.
- Chapter 4 shows how to define an external (system) routine, how to call it from LISP, and how to call back to LISP from the routine.
- Chapter 5 explains alien structures, which allow you to exchange data between LISP and routines written in other languages.
- Chapter 6 describes interrupt functions, which you can use to handle asynchronous events in the operating system.
- Chapter 7 shows how you can control the execution of keyboard functions and interrupt functions by assigning them interrupt levels. You can also protect sections of code against interruption and cause your program to wait until an event occurs or some needed information becomes available.

Associated Documents

The following documents are relevant to VAX LISP/VMS programming:

- *VAX LISP / VMS Program Development Guide*
- *Common LISP: The Language*
- *VMS Linker Utility Reference Manual*
- *Introduction to VMS System Routines*

- *VMS Utility Routines Manual*
- *VMS System Services Reference Manual*
- *VMS RTL Library (LIB$) Manual*
- *VMS Record Management Services Manual*
- *VAX Architecture Handbook*

For a complete list of VAX/VMS software documents, see the *Overview of VMS Documentation*.

Conventions

The following conventions are used in this guide:

Convention	Meaning
UPPERCASE	DCL commands and qualifiers and VMS file names are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The examples directory (SYS$SYSROOT:[VAXLISP.EXAMPLES] by default) contains sample LISP source files.
`UPPERCASE TYPEWRITER`	Defined LISP functions, macros, variables, constants, and other symbol names are printed in uppercase `TYPEWRITER` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The `CALL-OUT` macro calls a defined external routine . . . .
`lowercase typewriter`	LISP forms are printed in the text in lowercase `typewriter` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: `(setf example-1 (make-space))`
SANS SERIF	Format specifications of LISP functions and macros are printed in a sans serif typeface. For example: CALL-OUT *external-routine* &REST *routine-arguments*
italics	Lowercase *italics* in format specifications and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters. For example: The *routine-arguments* must be compatible with the arguments defined in the call to the `DEFINE-EXTERNAL-ROUTINE` macro.
( )	Parentheses used in examples of LISP code and in format specifications indicate the beginning and end of a LISP form. For example: `(setq name lisp)`

Convention	Meaning
[ ]	Square brackets in format specifications enclose optional elements. For example: *[doc-string]* Square brackets do not indicate optional elements when they are used in the syntax of a directory name in a VMS file specification. Here, the square bracket characters must be included in the syntax. For example: `(pathname "MIAMI::DBA1:[SMITH]LOGIN.COM;4")`
{ }	In function and macro format specifications, braces enclose elements that are considered one unit of code. For example: *{keyword value}*
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered one unit of code, which can be repeated zero or more times. For example: *{keyword value}**
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments that follow it are optional. For example: PPRINT *object* &OPTIONAL *stream* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.
&REST	In function and macro format specifications, the word &REST indicates that an indefinite number of arguments may appear. For example: CALL-OUT *external-routine* &REST *routine-arguments* Do not specify &REST when you invoke a function or macro whose definition includes &REST.
&KEY	In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: COMPILE-FILE *input-pathname* &KEY :LISTING :MACHINE-CODE :OPTIMIZE :OUTPUT-FILE :VERBOSE :WARNINGS Do not specify &KEY when you invoke a function or macro whose definition includes &KEY.
. . .	A horizontal ellipsis in a format specification means that the element preceding the ellipsis can be repeated. For example: *function-name . . .*
. . .	A vertical ellipsis in a code example indicates that all the information that the system would display in response to the function call is not shown; or, that all the information a user is to enter is not shown.

Convention	Meaning
Return	A word inside a box indicates that you press a key on the keyboard. For example: Return or Tab In code examples, carriage returns are implied at the end of each line. However, Return is used in some examples to emphasize carriage returns.
Ctrl/*x*	Two key names enclosed in a box indicate a control key sequence in which you hold down Ctrl while you press another key. For example: Ctrl/C or Ctrl/S
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	By default, MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. You can rebind the mouse buttons.

Red print	In interactive examples, user input is shown in red. For example:
	`Lisp> (cdr '(a b c))`
	`(B C)`
	`Lisp>`

Chapter 1

Overview of System Access Facilities

VAX LISP is layered on the VMS operating system. VAX LISP provides various means of access to the facilities of the operating system. This chapter provides a broad overview of those means of access. The remainder of this guide describes them in detail.

The VMS operating system offers the following general facilities to any programmer, including the LISP programmer:

- System services and other system routines. The routines are shipped with the operating system. Some routines provide an interface to operating system capabilities, such as I/O, scheduling, and notification of external events. Other routines set or retrieve parameters about a process or the entire system. There is a large family of math routines and a group of routines that can manage the screen of a video terminal.
- A multilanguage programming environment. Routines written in a language that conforms to the VAX Procedure Calling Standard can be called by and return values to routines written in other languages. For example, a LISP program can call a numeric analysis routine written in FORTRAN. The external routine can also call back to VAX LISP.

The remainder of this chapter briefly describes each of the facilities that let you work with operating system facilities. The chapters that follow describe each facility in greater detail.

1.1 File System Interface

Common LISP includes several functions and variables for dealing with the file system of "typical" operating system environments. Chapter 2 provides information on interfacing with the VMS file system, using functions and variables defined in both Common LISP (such as `DIRECTORY` and `*DEFAULT-PATHNAME-DEFAULTS*`) and VAX LISP (that is, `DEFAULT-DIRECTORY` and `*MODULE-DIRECTORY*`).

1.2 Operating System Information

Chapter 3 describes VAX LISP facilities for getting information from the operating system about the environment in which your program is running. For example, the `COMMAND-LINE-ENTITY-P` and `COMMAND-LINE-ENTITY-VALUE` functions return information about the way VAX LISP was invoked.

1.3 Callout and Callback Facilities

As a VAX LISP programmer, the callout and callback facilities are your primary means of access to routines that are external to VAX LISP.

To use the callout facility, you must first identify a system routine that you want to use, or write and debug a routine in another language. Information about system routines is in *VMS System Services Reference Manual* and *VMS Utility Routines Manual*. This documentation has information about the arguments that each routine expects, its effects, and the value, if any, that it returns.

If you write a routine in another language, you must be aware of the VAX data types and passing mechanisms of the routine's arguments.

Once you have identified or written an external routine, you must define it, using the DEFINE-EXTERNAL-ROUTINE macro. This macro informs LISP of the location and arguments of an external routine and sets up a mechanism whereby arguments expressed in LISP data types can be converted to the proper VAX data types for the external routine.

The CALL-OUT macro calls a defined external routine, passing it the arguments you specify and returning a value if the external routine returns a value.

The callback facility allows routines written in another VAX language (that follows the VAX Procedure Calling Standard) to invoke LISP functions. The LISP image must be started first, and the external routine must be called from LISP with the CALL-OUT macro.

1.4 Alien Structures

The DEFINE-EXTERNAL-ROUTINE macro can specify arguments for most common VAX data types. However, to pass more complex data you must define an alien structure that corresponds to the structure of the data in an external routine. An alien structure definition has two general purposes:

- To define a precise layout for a portion of memory
- To instruct LISP how to interpret fields in that memory, allowing you to access those fields by using LISP data types

An alien structure definition provides a template for instances of that structure, similar to a Common LISP structure definition created by the DEFSTRUCT macro. The DEFINE-ALIEN-STRUCTURE macro defines an alien structure and may also provide a constructor function, field accessor functions, a type-checking predicate, and so on, depending on the options with which it is called.

For example, you can pass an instance of an alien structure to an external routine, using CALL-OUT. The external routine can access or modify fields in the structure. When CALL-OUT returns, the modified structure is again available for LISP to interpret as LISP data.

1.5 Interrupt Functions

Normally, LISP is a synchronous environment; that is, events in LISP programs occur at times that can be predicted from the code and the data. Events such as garbage collections that interrupt the normal flow of program execution do so in a way that is transparent to user programs.

However, all events do not happen in a synchronous fashion. Some events are asynchronous; that is, they occur at unpredictable points in the program, although you can predict that they will eventually occur. For example:

- An I/O request is issued. Later, at an unpredictable point in the execution of the program, the I/O operation completes.
- A timer is set. The time of its expiration can be predicted but not the program state at that time.

A number of routines initiate activities that complete asynchronously. These routines start the activity and then return; they do not wait for the activity to complete. All these routines allow you to request notification of completion. In VAX LISP, this notification takes the form of an interrupt function.

An interrupt function is a function that you write and that is designed to execute as the result of an asynchronous event. Once you have written the interrupt function, you make it known to VAX LISP by using the INSTATE-INTERRUPT-FUNCTION function, which returns an identifier for the interrupt function. You then use CALL-OUT to pass this identifier, along with a VAX LISP constant, to a system routine that initiates an asynchronous activity. When the activity completes, your interrupt function will execute.

1.6 Controlling Interruptions and Synchronizing Execution

VAX LISP lets you control the way functions can interrupt each other. You can also synchronize program execution by causing the program to wait until an event occurs or information becomes available.

A function that is specified with BIND-KEYBOARD-FUNCTION or INSTATE-INTERRUPT-FUNCTION can also have an interrupt level specified. The interrupt level is an integer. When the function is called on to execute, it can do so only if its interrupt level is higher than the level at which VAX LISP is operating. By using interrupt levels, you can ensure that functions that must interrupt other functions can do so.

Some functions, such as those that modify shared data structures, must never be interrupted. You can use the CRITICAL-SECTION macro to protect such code from any interruption.

If your program has to wait for the execution of a keyboard function or an interrupt function, VAX LISP provides the WAIT function. The WAIT function halts normal LISP execution until a testing function that you specify returns non-NIL.

Chapter 2

File System Interface

This chapter describes the system-dependent aspects of the VAX LISP/VMS file system interface. It explains how to use VAX LISP pathnames as VMS file specifications. It also describes the functions and variables associated with directories in VAX LISP.

2.1 Pathnames and Namestrings

In VAX LISP, file names can be represented by pathnames, namestrings, symbols, or streams. This section covers pathnames and namestrings.

In Common LISP, a pathname is a LISP data object that represents a file specification. A namestring also represents a file specification. However, it provides the necessary translation between pathnames, which are implementation independent, and file specifications, which are implementation dependent.

The section is divided as follows:

- Section 2.1.1 describes the relationship between logical names and pathnames.
- Section 2.1.2 tells you when to use pathnames.
- Section 2.1.3 describes the fields of a Common LISP pathname.
- Section 2.1.4 lists the values of VAX LISP pathname fields.
- Section 2.1.5 shows you three ways to create pathnames.
- Section 2.1.6 tells you what LISP functions to use to compare pathnames.
- Section 2.1.7 describes the purpose of namestrings.
- Section 2.1.8 shows you how to convert pathnames into namestrings.
- Section 2.1.9 describes the `*DEFAULT-PATHNAME-DEFAULTS*` variable and how to change it.

2.1.1 Relationship Between Logical Names and Pathnames

In VAX LISP/VMS, logical names are translated into file specifications at the time a pathname is created to allow pathnames to be merged properly. Translation of logical names is not normally a problem unless the logical name has multiple translations. In general, strings (rather than pathnames) for file specifications improve the use of logical names with multiple translations. Some functions that accept pathnames or strings as arguments (such as `OPEN` and `PROBE-FILE`) can be passed a string including a reference to such a logical name, and the appropriate translation will be used. Other functions, however (such as LOAD and COMPILE-FILE), convert string arguments to pathnames to apply the default file type and directory, if they are not specified. In that case, the first translation for which the device and directory exist is used. Providing a complete file specification in the string argument to LOAD or COMPILE-FILE allows all the translations of included logical names to be used.

If a file specification includes a reference to a remote node, logical names are not translated in the resulting pathname.

2.1.2 When to Use Pathnames

Pathnames do not replace the traditional ways of representing a file in LISP. Instead, the pathnames add a new way of representing a file to make LISP programs portable between systems with different file-naming conventions.

Pathnames, however, do not have to refer to an existing file or give complete file specifications; pathnames can exist as data objects and can be used as arguments to pathname functions (see Section 2.2 and *Common LISP: The Language*).

Several pathname functions and most functions that deal with the file system can take either pathnames, namestrings, symbols, or streams as their arguments. However, the values of the following variable and argument must be pathnames:

- The *DEFAULT-PATHNAME-DEFAULTS* variable
- The defaults argument in a call to the PARSE-NAMESTRING function

See Section 2.1.9 and *Common LISP: The Language* for a description of the preceding variable and function.

2.1.3 Fields of a Common LISP Pathname

A Common LISP pathname is a LISP data object composed of six fields. Each field represents one of the following aspects of a file specification:

Host	file system
Device	file structure or physical or logical device on which files are stored
Directory	group of related files
Name	file name
Type	file extension
Version	number incremented every time the file is modified

2.1.4 Values of VAX LISP Pathname Fields

For a description of VAX LISP file specifications, see Chapter 1 of the *VAX LISP/VMS Program Development Guide*. The following examples show how the components of a VAX LISP file specification are mapped into the fields of a VAX LISP pathname. The first example shows a VAX LISP file specification:

```
MIAMI::DBA1:[SMITH]LOGIN.COM;4
```

The second example shows the pathname that represents that file specification:

```
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME "LOGIN" :TYPE "COM" :VERSION 4)
```

Table 2–1 names the fields of a VAX LISP pathname, the VMS file components that correspond to those fields, and the VAX LISP data type each field accepts.

Table 2–1: VAX LISP Pathname Fields

Field Name	VMS Component	Field Value
`:HOST`	node	String, integer, or `NIL`. If you specify a string, the field value can include an access control string, and you must omit the final double colon (::). Examples of host field values are `0`, `"0"`, `"HOST"`, `"A::B::C"`, and `"A\"NAME password\""`.
`:DEVICE`	device	String or `NIL`. If you specify a string, you must omit the final colon (:). An example of a device field value is `"DBA1"`.
`:DIRECTORY`	directory	String, `NIL`, or the `:WILD` keyword. The `:WILD` keyword is translated to the VMS wildcard symbol, the asterisk (*). If you specify a string, you must omit the square brackets ([ ]) or angle brackets (<>). Examples of directory field values are `"SMITH"`, `"SMITH.COMMAND"`, and `"SMITH . . . "`.
`:NAME`	filename	String, `NIL`, or the `:WILD` keyword. The `:WILD` keyword is translated to the VMS wildcard symbol, the asterisk (*). If you specify a string, you must omit the period (.) that follows the name. Examples of name field values are `"LISP"` and `"L*SP"`.
`:TYPE`	filetype	String, `NIL`, or the `:WILD` keyword. The `:WILD` keyword is translated to the VMS wildcard symbol, the asterisk (*). If you specify a string, you must omit the period (.) that precedes the type. Examples of type field values are `"LSP"` and `"FAS"`.
`:VERSION`	version	String, integer, `NIL`, or keyword. An integer can be positive, negative, or zero. Zero represents the newest version of a file, and minus one (–1) represents the previous version of a file. The following keywords can be specified: `:NEWEST` equivalent to 0; `:PREVIOUS` equivalent to –1; `:WILD` equivalent to "*". If you specify a string, you must omit the initial semicolon (;). Examples of version field values are `0`, `-14`, `"2%"`, and `"4*"`.

2.1.5 Creating Pathnames

You can create a pathname in one of three ways, using the following functions:

- **The MAKE-PATHNAME function**

```
Lisp> (make-pathname :host "miami"
                     :device "dba1"
                     :directory "smith"
                     :name "test"
                     :type "lsp"
                     :version 1)
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME  "TEST" :TYPE "LSP" :VERSION 1)
```

- **The PATHNAME function**

  ```
  Lisp> (pathname "miami::dba1:[smith]login.com;4")
  #S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
              :NAME "LOGIN" :TYPE "COM" :VERSION 4)
  ```

  The PARSE-NAMESTRING function

  ```
  Lisp> (parse-namestring "miami::dba1:[smith]login.com")
  #S(PATHNAME :HOST "MIAMI" :DEVICE "DBAI" :DIRECTORY "SMITH"
              :NAME "LOGIN" :TYPE "COM" :VERSION 4)
  ```

The `MAKE-PATHNAME` function creates a pathname directly from the user-input keywords `:HOST`, `:DIRECTORY`, and so on. On the other hand, the `PATHNAME` function and the `PARSE-NAMESTRING` function create a pathname by:

- Using a pathname, namestring, symbol, or stream as an argument
- Parsing the argument
- Returning a pathname if the parse operation is successful

NOTE

The LISP system does not check whether you have entered an existing or complete file specification when you create a pathname. Therefore, you can create a pathname that is not usable and, when you perform a file operation, it will not succeed. To correct the problem, you must change the pathname to conform to a valid file specification. See Chapter 1 of the *VAX LISP/VMS Program Development Guide* for a description of VMS file specifications and Section 2.1.4 for a description of the field values in a VAX LISP pathname.

You can specify any valid DECnet–VAX node specification in the host field of a pathname when you are calling a parsing function. Each host name in the specification must be followed by two colons (::) as shown in the following example:

```
Lisp> (pathname "first::second::third::dba1:[smith]pathname")
#S(PATHNAME :HOST "FIRST::SECOND::THIRD" :DEVICE "DBA1"
            :DIRECTORY "SMITH" :NAME "PATHNAME" :TYPE NIL
            :VERSION NIL)
```

The `PATHNAME` function concatenates the nodes, `FIRST`, `SECOND`, and `THIRD`, into a single string in the pathname's host field.

If the namestring argument in a call to the `PATHNAME` or the `PARSE-NAMESTRING` function is a logical name, the logical name is translated.

The `PATHNAME` and `PARSE-NAMESTRING` functions both return a pathname if the parse operation succeeds. They return different values in the case of error. `PATHNAME` signals an error if the operation fails. `PARSE-NAMESTRING` either returns `NIL` or signals an error, depending on the value of the `:JUNK-ALLOWED` keyword.

Descriptions of the `MAKE-PATHNAME`, `PATHNAME`, and `PARSE-NAMESTRING` functions are provided in *Common LISP: The Language*.

2.1.6 Comparing Similar Pathnames

You should use the EQUAL function to compare pathnames with the same field entries. This function is sensitive to keywords and their equivalent symbols (that is, :WILD is equivalent to "*"), but ignores case. For example, if the MAKE-PATHNAME and PARSE-NAMESTRING functions each create a pathname for TEST.*;, you can use the EQUAL function to compare the pathnames (see *Common LISP: The Language*). The following calls to the SETF macro set the pathnames created by the MAKE-PATHNAME and PARSE-NAMESTRING functions to the variables *x* and *y*:

```
Lisp> (setf x (make-pathname :name "Test"
                             :type :wild
                             :version 0))
#S(PATHNAME :HOST "MIAMI" :DEVICE NIL :DIRECTORY NIL
            :NAME "Test" :TYPE :WILD :VERSION 0)
Lisp> (setf y (parse-namestring "tEST.*;"))
#S(PATHNAME :HOST "MIAMI" :DEVICE NIL :DIRECTORY NIL
            :NAME "TEST" :TYPE "*" :VERSION :NEWEST)
Lisp> (equal x y)
T
```

The EQUAL function returns T to indicate that the pathname values of x and y are equal.

2.1.7 Purpose of Namestrings

Because operating systems such as VMS and ULTRIX have different ways of formatting file names, Common LISP uses namestrings to translate between pathnames (implementation-independent names) and file names (implementation-dependent names).

A namestring is a string naming a file in an implementation-dependent form customary for the file system. A VAX LISP namestring is a string containing a valid VMS file specification. For example, if a file in the VMS file system is called SYS$LOGIN:LOGIN.COM;4, the equivalent namestring would be displayed as "SYS$LOGIN:LOGIN.COM;4".

File system functions, such as LOAD, accept pathnames but internally convert them to namestrings. For more information on namestrings, see Section 2.1.8.

2.1.8 Converting Pathnames into Namestrings

You can convert a pathname into a namestring by specifying the pathname in a call to the NAMESTRING function.

If the argument you specify contains the name of a host, the function invokes DECnet–VAX to perform network operations whether or not the specified host is the current host. To avoid using DECnet–VAX, the VAX LISP implementation of the NAMESTRING function removes the host value if the value is the same as the translation value of SYS$NODE. The following call to the TRANSLATE-LOGICAL-NAME function shows that the current node is MIAMI:

```
Lisp> (translate-logical-name "sys$node")
("_MIAMI::")
```

If you use the PATHNAME function to create a pathname whose host field is the current node, NAMESTRING does not include the host in the namestring it returns. For example, suppose the host is still MIAMI. If you use the SETF macro to set a variable called THIS-PATHNAME to the pathname created by the PATHNAME function, a subsequent call to NAMESTRING does not include the host:

```
Lisp> (setf this-pathname
            (pathname "miami::dba1:[smith]login.com;4"))
#S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME "LOGIN" :TYPE "COM" :VERSION 4)
Lisp> (namestring this-pathname)
"DBA1:[SMITH]LOGIN.COM;4"
```

Now, suppose you use the PATHNAME function to create a pathname called THAT-PATHNAME whose host field is BOSTON. In this case, a call to the SETF macro sets THAT-PATHNAME to the pathname that is created with the PATHNAME function, and a subsequent call to NAMESTRING does include the host. This is because the host BOSTON is not the current node.

```
Lisp> (setf that-pathname
            (pathname "boston::dba1:[smith]login.com;4"))
#S(PATHNAME :HOST "BOSTON" :DEVICE "DBA1" :DIRECTORY "SMITH"
            :NAME "LOGIN" :TYPE "COM" :VERSION 4)
Lisp> (namestring that-pathname)
"boston::dba1:[smith]login.com;4"
```

If you want to invoke DECnet–VAX and specify the current host, specify the host with an access control string or zero. For example:

```
Lisp> (setf that-pathname
            (pathname "0::thatdevice:[smith]login.com"))
#S(PATHNAME :HOST "0" :DEVICE "THATDEVICE" :DIRECTORY "SMITH"
            :NAME "LOGIN" :TYPE "COM" :VERSION NIL)
Lisp> (namestring that-pathname)
"0::thatdevice:[smith]login.com"
```

Table 2–1 notes that in VAX LISP the host field of a pathname can include an access control string. If the NAMESTRING function is called with a pathname argument whose host field includes an access control string, the namestring that is returned includes the host, even if the value in the pathname's host field is the same as the current node.

Assume that the current host is MIAMI. The following SETF expression sets THIS-PATHNAME to the pathname that is created with the PATHNAME function:

```
Lisp> (setf this-pathname
            (pathname
              "miami\"smith mypassword\"::thisdevice:[smith]file"))
#S(PATHNAME :HOST "MIAMI:\SMITH mypassword\"" :DEVICE "THISDEVICE"
            :DIRECTORY "SMITH" :NAME "FILE" :TYPE NIL VERSION: NIL)
```

The host field of the pathname that is created contains the host MIAMI and the access control string SMITH MYPASSWORD. The NAMESTRING function, when called with THIS-PATHNAME as its argument, returns a namestring that includes all the pathname field values:

```
Lisp> (namestring this-pathname)
"miami\"smith mypassword\"::thisdevice:[smith]file"
```

2.1.9 Using the *DEFAULT-PATHNAME-DEFAULTS* Variable

The value of the *DEFAULT-PATHNAME-DEFAULTS* variable is used by some pathname functions to fill pathname fields not specified in their arguments. The default value of this variable is a pathname whose host, device, and directory fields indicate the current directory and whose name, type, and version fields contain NIL.

In VAX LISP, you can change the value of the *DEFAULT-PATHNAME-DEFAULTS* variable as follows:

- **The SETF macro**

  The following example illustrates using the SETF macro to change a pathname's directory from [SMITH] to [SMITH.TEST]:

  ```
  Lisp> (setf *default-pathname-defaults*
          (make-pathname :directory "[smith.test]"))
  #S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
              :DIRECTORY "[SMITH.TEST]" :NAME NIL
              :TYPE NIL :VERSION NIL)
  ```

- **The DEFAULT-DIRECTORY function**

  The value of the *DEFAULT-PATHNAME-DEFAULTS* variable is set to the value of your default directory when LISP starts and when you change your directory with the form (SETF (DEFAULT-DIRECTORY) . . .). To check the value of your default directory, call the DEFAULT-DIRECTORY function. For example:

  ```
  Lisp> (default-directory)
  #S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
              :DIRECTORY "SMITH" :NAME NIL
              :TYPE NIL :VERSION NIL)
  ```

  The pathname returned in this example indicates that the default directory is SMITH on host MIAMI. In this case, each time a pathname function fills a pathname field with a default value, the corresponding value in the directory SMITH is used.

  To change the value of your default directory, set it with the SETF macro. The following example illustrates how to change a default directory from SMITH to SMITH.TEST:

  ```
  Lisp> (setf (default-directory) "[.test]")
  "[.TEST]"
  ```

  The next example illustrates that when the directory is changed, the DEFAULT-DIRECTORY function returns a new pathname referring to the new default directory:

  ```
  Lisp> (default-directory)
  #S(PATHNAME :HOST "MIAMI" :DEVICE "DBA1"
              :DIRECTORY "SMITH.TEST" :NAME NIL
              :TYPE NIL :VERSION NIL)
  ```

NOTE

The value of the *DEFAULT-PATHNAME-DEFAULTS* variable must be a pathname. Do not set this variable to a namestring, symbol, or stream.

2.2 Directories

VAX LISP provides one function and one variable for accessing directories, in addition to the `DIRECTORY` function described in *Common LISP: The Language*.

- **The DEFAULT-DIRECTORY function**

  The `DEFAULT-DIRECTORY` function returns a pathname that refers to the current directory. You can use this function with the `SETF` macro to change your default directory.

- **The DIRECTORY function**

  The `DIRECTORY` function converts its argument to a pathname and merges that pathname with the following elements of a file specification:

  host::device:[directory].*;**

  The values for the *host*, *device*, and *directory* fields are supplied by the `*DEFAULT-PATHNAME-DEFAULTS*` variable (see Section 2.1.9).

- **The *MODULE-DIRECTORY* variable**

  The value of the `*MODULE-DIRECTORY*` variable refers to the directory containing the module that is currently being loaded into the LISP environment due to a call to the `REQUIRE` function. The value is bound during calls to the `REQUIRE` function, and is a pathname.

  After the module is loaded, the `*MODULE-DIRECTORY*` variable is rebound to `NIL`.

See *VAX LISP/VMS Function, Macro and Variable Descriptions* for descriptions of these functions and variable.

Chapter 3

Getting Operating-System Information

This chapter describes how to get information about the current state of the operating system and the process running VAX LISP. The following topics are covered in this chapter:

- Section 3.1 describes how to access the command line used to invoke VAX LISP.
- Section 3.2 tells how to find the version number.
- Section 3.3 shows how to get device information.
- Section 3.4 describes how to get file information.
- Section 3.5 describes how to get process information.
- Section 3.6 tells how to control terminal characteristics.
- Section 3.7 shows how to get system messages.
- Section 3.8 describes how to translate logical names.

3.1 Accessing the Command Line

VAX LISP includes two functions for accessing information about the command line that invoked VAX LISP: the COMMAND-LINE-ENTITY-P function indicates the presence or absence of a given entity on the command line, and the COMMAND-LINE-ENTITY-VALUE function returns the value of a given entity on the command line. These two functions provide an interface to the VMS utility routines CLI$PRESENT and CLI$GET_VALUE, respectively (see the *VMS Utility Routines Manual* for more information on these routines).

The command line functions have the following format:

COMMAND-LINE-ENTITY-P *entity-desc*
COMMAND-LINE-ENTITY-VALUE *entity-desc*

The *entity-desc* argument may be a character string or a symbol. If you supply a symbol, the print name of the symbol is used. (See the description of the *entity-desc* argument to CLI$PRESENT and CLI$GET_VALUE in the *VMS Utility Routines Manual*.)

Each of these functions returns two values. The values returned by `COMMAND-LINE-ENTITY-P` have the following meanings:

First Value	Second Value	Meaning
`T`	`T`	Entity was explicitly specified as present
`T`	`NIL`	Entity was present by default
`NIL`	`T`	Entity was explicitly negated with NO
`NIL`	`NIL`	Entity was absent by default

The values returned by `COMMAND-LINE-ENTITY-VALUE` are:

1. A string containing the first or next value of the specified entity, depending on whether this is the first request for this entity or a subsequent request. If the entity is not present, has no value, or if all values for this entity have been obtained, `NIL` is returned.
2. A character that is the character delimiter that preceded the returned entity. This is normally a comma (`#\,`) but may be a plus sign (`#\+`) to indicate concatenation.

The `COMMAND-LINE-ENTITY-P` and `COMMAND-LINE-ENTITY-VALUE` functions are useful in a user-defined LISP system that is invoked by a defined DCL command. See *VAX LISP/VMS System-Building Guide* for more information on building a LISP system.

3.2 Finding the Version Number

The `SOFTWARE-VERSION-NUMBER` function finds the version number of a given software product or component. The major and minor parts of the version number are returned as multiple values. That is, for a version number in the form *m.n*, `SOFTWARE-VERSION-NUMBER` returns a fixnum for *m* and a fixnum for *n*.

A call to the `SOFTWARE-VERSION-NUMBER` function has the format:

SOFTWARE-VERSION-NUMBER *component*

The *component* is a string containing the name of a software product or component. The possible values are:

```
"VAX LISP"
"VMS"
"UIS"
"CLX"
```

3.3 Getting Device Information

The `GET-DEVICE-INFORMATION` function returns information about a device. The keywords you specify with the function determine the type of information the function returns. The format of the `GET-DEVICE-INFORMATION` function is:

GET-DEVICE-INFORMATION *device* &REST {*keyword*}*

The *device* argument is a string that names the device; *keyword* arguments specify the type of information you want. Do not specify values for the keywords. Table 3–1 lists the keywords and the values they return.

Table 3–1: GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
`:ACP-PID`	An integer that specifies the ACP process ID.
`:ACP-TYPE`	An integer that specifies the ACP type code.
`:BUFFER-SIZE`	An integer that specifies the buffer size.
`:CLUSTER-SIZE`	An integer that specifies the volume cluster size.
`:CYLINDERS`	An integer that specifies the number of cylinders on the device.
`:DEVICE-CHARACTERISTICS`	A vector of 32 bits that specifies the device characteristics. See the *VMS I/O User's Reference Manual: Part I* for information about device characteristics.
`:DEVICE-CLASS`	An integer that specifies the device class.
`:DEVICE-DEPENDENT-0`	A bit vector that specifies device-dependent information.
`:DEVICE-DEPENDENT-1`	A bit vector that specifies device-dependent information.
`:DEVICE-NAME`	A string that specifies the device name.
`:DEVICE-TYPE`	An integer that specifies the device type.
`:ERROR-COUNT`	An integer that specifies the number of errors that have occurred on the device.
`:FREE-BLOCKS`	An integer that specifies the number of free blocks on the device; otherwise, `NIL`.
`:LOGICAL-VOLUME-NAME`	A string that specifies the logical name associated with the volume on the device. This keyword is valid only for disks.
`:MAX-BLOCKS`	An integer that specifies the maximum number of logical blocks that can exist on the device.
`:MAX-FILES`	An integer that specifies the maximum number of files that can exist on the device.
`:MOUNT-COUNT`	An integer that specifies the number of times the device has been mounted.
`:NEXT-DEVICE-NAME`	A string that specifies the name of the next volume in the volume set.
`:OPERATION-COUNT`	An integer that specifies the number of operations that have been performed on the device.
`:OWNER-UIC`	An integer that specifies the UIC of the owner.
`:PID`	An integer that specifies the process ID of the owner.
`:RECORD-SIZE`	An integer that specifies the blocked record size.
`:REFERENCE-COUNT`	An integer that specifies the number of channels assigned to the device.
`:ROOT-DEVICE-NAME`	A string that specifies the name of the root volume in the volume set.
`:SECTORS`	An integer that specifies the number of sectors per track.
`:SERIAL-NUMBER`	An integer that specifies the serial number.
`:TRACKS`	An integer that specifies the number of tracks per cylinder.
`:TRANSACTION-COUNT`	An integer that specifies the number of files open on the device.

(continued on next page)

Table 3–1 (Cont.): GET-DEVICE-INFORMATION Keywords

Keyword	Return Value
`:UNIT`	An integer that specifies the unit number.
`:VOLUME-COUNT`	An integer that specifies the number of volumes in the volume set.
`:VOLUME-NAME`	A string that specifies the name of the volume on the device.
`:VOLUME-NUMBER`	An integer that specifies the number of the volume on the device.
`:VOLUME-PROTECTION`	A vector of 32 bits that specifies the volume protection mask.

The keywords and their values are returned as a property list in the following format:

({:*keyword value*}*)

The function preserves the order of the keyword–value pairs in the argument list. If you do not specify any keywords, the function returns a list of all the keyword–value pairs. If the device does not exist, the function returns `NIL`.

This function is similar to the VMS system service $GETDVI. For more information on the $GETDVI system service, see the *VMS System Services Reference Manual* and the *VMS I/O User's Reference Manual: Part I.*

3.4 Getting File Information

The `GET-FILE-INFORMATION` function returns information about a file. The syntax is:

GET-FILE-INFORMATION *pathname* &REST {*keyword*}*

The *pathname* argument may be a pathname, namestring, symbol, or stream that represents the name of the file.

The optional keywords let you specify particular types of information about the file. Do not specify values with the keywords. Table 3–2 lists the keywords and the values they return.

Table 3–2: GET-FILE-INFORMATION Keywords

Keyword	Return Value
`:ALLOCATION-QUANTITY`	An integer that specifies the number of blocks allocated for the file.
`:BACKUP-DATE`	The last universal date and time the file was backed up. If the file has not been backed up, the function returns `NIL`.
`:BLOCK-SIZE`	An integer that specifies the block size.
`:CREATION-DATE`	The universal date and time the file was created.

(continued on next page)

Table 3–2 (Cont.): GET-FILE-INFORMATION Keywords

Keyword	Return Value
`:DEFAULT-EXTENSION`	An integer that specifies the number of blocks added to the file's size when the file was extended.
`:END-OF-FILE-BLOCK`	An integer that specifies the block in which the file ends.
`:EXPIRATION-DATE`	The universal date and time the file expires. If an expiration date is not recorded, the function returns `NIL`.
`:FIRST-FREE-BYTE`	An integer that specifies the offset of the first byte in the file's end-of-file block.
`:FIXED-CONTROL-SIZE`	An integer that specifies the fixed control area size.
`:GROUP`	An integer that specifies the owner group number.
`:LONGEST-RECORD-LENGTH`	An integer that specifies the length of the longest record in the file.
`:MAX-RECORD-SIZE`	An integer that specifies the maximum size allowed for a record.
`:MEMBER`	An integer that specifies the owner member number.
`:ORGANIZATION`	An integer that specifies the organization.
`:PROTECTION`	A vector of 16 bits that specifies the protection code.
`:RECORD-ATTRIBUTES`	An integer that specifies the record attributes.
`:RECORD-FORMAT`	An integer that specifies the record format.
`:REVISION`	An integer that specifies the revision number.
`:REVISION-DATE`	The last universal date and time the file was revised.
`:UIC`	An integer that specifies the owner UIC.
`:VERSION-LIMIT`	An integer that specifies the maximum version number the file can have.

These keywords correspond to the fields of the RMS file access block (FAB) and extended attribute block (XAB). See the *VMS Record Management Services Manual* for information on FAB and XAB fields.

The keywords and their values are returned as a property list in the following format:

({:*keyword value*}*)

The function preserves the order of the keyword–value pairs in the argument list. If you do not specify any keywords, the function returns a list of all the keyword–value pairs. If the file does not exist, the function returns `NIL`.

3.5 Getting Process Information

The `GET-PROCESS-INFORMATION` function returns information about a process. If the process is nonexistent, this function returns `NIL`. Its format is:

GET-PROCESS-INFORMATION *process* &REST {*keyword*}*

The *process* argument is the name or process identification (PID) of a process. You can specify a string, an integer, or `NIL`. If you specify a string, the argument is the process name. It must be case sensitive. If you specify an integer, the argument is the PID. If you specify `NIL`, the function returns information on the current process.

The optional keywords determine the type of information the function returns. Do not specify values with the keywords. Table 3–3 lists the keywords and the values they return.

Table 3–3: GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
`:ACCOUNT`	A string that specifies the account.
`:ACTIVE-PAGE-TABLE-COUNT`	An integer that specifies the active page table count.
`:AST-ACTIVE`	A vector of four bits that specifies the number of access modes that have active asynchronous system traps (ASTs) for the process.
`:AST-COUNT`	An integer that specifies the remaining AST quota.
`:AST-ENABLED`	A vector of four bits that specifies the number of access modes that have enabled ASTs for the process.
`:AST-QUOTA`	An integer that specifies the AST quota.
`:AUTHORIZED-PRIVILEGES`	A vector of 64 bits that specifies the privileges the process is authorized to enable.
`:BASE-PRIORITY`	An integer that specifies the base priority.
`:BATCH`	Either `T` or `NIL`. The function returns `T` if the process is a batch job; otherwise, returns `NIL`.
`:BIO-BYTE-COUNT`	An integer that specifies the remaining buffered I/O byte count quota.
`:BIO-BYTE-QUOTA`	An integer that specifies the buffered I/O byte count quota.
`:BIO-COUNT`	An integer that specifies the remaining buffered I/O operation quota.
`:BIO-OPERATIONS`	An integer that specifies the number of buffered I/O operations the process has performed.
`:BIO-QUOTA`	An integer that specifies the buffered I/O operation quota.
`:CLI-TABLENAME`	A string that specifies the file name of the current command language interpreter table.
`:CPU-LIMIT`	An integer that specifies the CPU time limit of the process in 10-millisecond units.
`:CPU-TIME`	An integer that specifies the accumulated CPU time of the process in 10-millisecond units.
`:CURRENT-PRIORITY`	An integer that specifies the current priority.
`:CURRENT-PRIVILEGES`	A vector of 64 bits that specifies the current privileges.
`:DEFAULT-PAGE-FAULT-CLUSTER`	An integer that specifies the default page fault cluster size.
`:DEFAULT-PRIVILEGES`	A vector of 64 bits that specifies the default privileges.
`:DIO-COUNT`	An integer that specifies the remaining direct I/O operation quota.
`:DIO-OPERATIONS`	An integer that specifies the number of direct I/O operations the process has performed.

(continued on next page)

Table 3–3 (Cont.): GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
:DIO-QUOTA	An integer that specifies the direct I/O operation quota.
:ENQUEUE-COUNT	An integer that specifies the number of lock manager enqueues.
:ENQUEUE-QUOTA	An integer that specifies the lock manager enqueue quota.
:EVENT-FLAG-WAIT-MASK	A vector of 32 bits that specifies the event flag wait mask.
:FIRST-FREE-P0-PAGE	An integer that specifies the first free page at the end of the program region.
:FIRST-FREE-P1-PAGE	An integer that specifies the first free page at the end of the control region.
:GLOBAL-PAGES	An integer that specifies the number of global pages in the working set.
:GROUP	An integer that specifies the group field of the UIC.
:IMAGE-NAME	A string that specifies the current image file name.
:IMAGE-PRIVILEGES	A vector of 64 bits that specifies the privileges with which the current image of the process was installed.
:JOB-SUBPROCESS-COUNT	An integer that specifies the number of subprocesses.
:LOCAL-EVENT-FLAGS	A vector of 32 bits that specifies the local event flags the process has in effect.
:LOGIN-TIME	An integer in internal time that specifies the time the process was created.
:MEMBER	An integer that specifies the member field of the UIC.
:MOUNTED-VOLUMES	An integer that specifies the number of mounted volumes.
:OPEN-FILE-COUNT	An integer that specifies the remaining open file quota.
:OPEN-FILE-QUOTA	An integer that specifies the open file quota.
:OWNER-PID	An integer that specifies the process ID of the owner.
:PAGE-FAULTS	An integer that specifies the number of page faults.
:PAGE-FILE-COUNT	An integer that specifies the number of paging file pages remaining to the process.
:PAGE-FILE-QUOTA	An integer that specifies the paging file quota.
:PAGES-AVAILABLE	An integer that specifies the number of virtual pages available for expansion.
:PID	An integer that specifies the process ID.
:PID-OF-PARENT	An integer that specifies the PID of the parent process. This integer differs from `:OWNER-PID` in that `:PID-OF-PARENT` refers to the top-level process, while `:OWNER-PID` refers to the process immediately above the current process or subprocess.

(continued on next page)

Table 3–3 (Cont.): GET-PROCESS-INFORMATION Keywords

Keyword	Return Value
`:PROCESS-CREATION-FLAGS`	A 32-bit bit-vector that specifies the flags used to create the process.
`:PROCESS-INDEX`	An integer that specifies the index number of the process at a given instant. (Process index numbers are reassigned to different processes over time.)
`:PROCESS-NAME`	A string that specifies the name of the process.
`:SITE-SPECIFIC`	A longword that specifies the contents of the site-specific longword.
`:STATE`	An integer that specifies the state of the process.
`:STATUS`	A vector of 32 bits that specifies the status flags.
`:SUBPROCESS-COUNT`	An integer that specifies the number of subprocesses owned by the process.
`:SUBPROCESS-QUOTA`	An integer that specifies the subprocess quota.
`:TERMINAL`	A string that specifies the name of the terminal with which the process is interacting.
`:TERMINATION-MAILBOX`	An integer that specifies the termination mailbox unit number.
`:TIMER-QUEUE-COUNT`	An integer that specifies the remaining timer queue entry quota.
`:TIMER-QUEUE-QUOTA`	An integer that specifies the timer queue entry quota.
`:UAF-FLAGS`	A 12-bit bit-vector that specifies the UAF flags of the user who owns the process.
`:UIC`	An integer that specifies the UIC.
`:USERNAME`	A string that specifies the user name.
`:VIRTUAL-ADDRESS-PEAK`	An integer that specifies the peak virtual address space size.
`:WORKING-SET-AUTHORIZED-EXTENT`	An integer that specifies the maximum authorized working set extent.
`:WORKING-SET-AUTHORIZED-QUOTA`	An integer that specifies the authorized working set quota.
`:WORKING-SET-COUNT`	An integer that specifies the number of process pages in the working set.
`:WORKING-SET-DEFAULT`	An integer that specifies the default working set size.
`:WORKING-SET-EXTENT`	An integer that specifies the current working set size extent.
`:WORKING-SET-PEAK`	An integer that specifies the peak working set size.
`:WORKING-SET-QUOTA`	An integer that specifies the current working set quota.
`:WORKING-SET-SIZE`	An integer that specifies the current working set size.

The keywords and their values are returned as a property list in the following format:

({:*keyword value*}*)

The function preserves the order of the keyword–value pairs in the argument list. If you do not specify any keywords, the function returns a list of all the keyword–value pairs. If the process does not exist, the function returns NIL.

This function is similar to the VMS system service $GETJPI. For more information on the $GETJPI system service, see the *VMS System Services Reference Manual*.

3.6 Controlling Terminal Characteristics

VAX LISP provides functions for getting information about, and for changing, the terminal characteristics of the device associated with the *TERMINAL-IO* variable. The GET-TERMINAL-MODES and SET-TERMINAL-MODES functions are similar to the DCL commands SHOW TERMINAL and SET TERMINAL, respectively. (See the *VMS DCL Dictionary* for more information on the DCL commands.)

3.6.1 Using the GET-TERMINAL-MODES Function

The GET-TERMINAL-MODES function returns information about the terminal characteristics of the device associated with the *TERMINAL-IO* variable when you invoke the LISP system. If the stream bound to this variable is not connected to a terminal, the LISP system signals an error.

The format of the GET-TERMINAL-MODES function is:

GET-TERMINAL-MODES &REST {*keyword*}*

The keywords you specify determine the terminal characteristics about which the function returns information. Table 3–4 lists the keywords and the information they return. Do not specify values with the keywords when you call GET-TERMINAL-MODES.

Table 3–4: GET-TERMINAL-MODES Keywords

Keyword	Return Value
:BROADCAST	Either T or NIL. The function returns T if your terminal can receive broadcast messages, such as MAIL notifications and REPLY messages; otherwise, returns NIL.
:ECHO	Either T or NIL. The function returns T if the terminal displays the input character that it receives; otherwise, returns NIL. If the function returns NIL, the terminal displays only data output from the system or a user application program.
:ESCAPE	Either T or NIL. The function returns T if ANSI standard escape sequences transmitted from the terminal are handled as a single multicharacter terminator; otherwise, returns NIL. The terminal driver checks the escape sequences for syntax before passing them to the program. For more information on escape sequences, see the *VMS I/O User's Reference Manual: Part I*.
:HALF-DUPLEX	Either T or NIL. The function returns T if the terminal's operating mode is half-duplex, and the function returns NIL if the operating mode is full-duplex. For a description of terminal operating modes, see the *VMS I/O User's Reference Manual: Part I*.

(continued on next page)

Table 3–4 (Cont.): GET-TERMINAL-MODES Keywords

Keyword	Return Value
`:PASS-ALL`	Either `T` or `NIL`. The function returns `T` if the system does not expand tab characters to blanks, fill carriage return or linefeed characters, recognize control characters, and receive broadcast messages. The function returns `NIL` if the system passes all data to an application program as binary data.
`:PASS-THROUGH`	Either `T` or `NIL`. This mode is the same as the `:PASS-ALL` mode, except that "TTSYNC" protocol (Ctrl/S and Ctrl/Q) is still used.
`:TYPE-AHEAD`	Either `T` or `NIL`. The function returns `T` if the terminal accepts input that is typed when there is no outstanding read, and the function returns `NIL` if the terminal driver is dedicated and accepts input only when a program or the system issues a read.
`:WRAP`	Either `T` or `NIL`. The function returns `T` if the terminal generates a carriage return and a line feed when the end of a line is reached. Otherwise, the function returns `NIL`. The end of the line is determined by the terminal-width setting.

The keywords and their values are returned as a list in the following format:

(*:keyword-1 value-1 :keyword-2 value-2 . . .*)

This is the same format as an argument to the `SET-TERMINAL-MODES` function.

The function preserves the order of the keyword–value pairs in your argument list. If you do not specify keywords, the function returns a list of all eight keyword–value pairs. For example:

```
Lisp> (get-terminal-modes)
(:BROADCAST T :ECHO T :ESCAPE NIL :HALF-DUPLEX NIL :PASS-ALL NIL
:PASS-THROUGH NIL :TYPE-AHEAD T :WRAP T)
```

3.6.2 Using the SET-TERMINAL-MODES Function

The `SET-TERMINAL-MODES` function changes certain terminal characteristics of the stream bound to the `*TERMINAL-IO*` variable when you invoke the LISP system.

NOTE

Changing terminal modes affects *all* the streams that are open to the terminal. If you put one stream into pass-through mode, for example, every stream open to the terminal is put into pass-through mode.

The `SET-TERMINAL-MODES` function has the following syntax:

```
SET-TERMINAL-MODES &KEY :BROADCAST :ECHO :ESCAPE :HALF-DUPLEX
                        :PASS-ALL :PASS-THROUGH :TYPE-AHEAD :WRAP
```

See Table 3–4 for a description of the keywords you can use with `SET-TERMINAL-MODES`.

3.6.3 Handling Nonstandard Terminal States

You can create an error handler to prevent your terminal from being placed in a nonstandard state. For example:

```
Lisp> (defvar *old-terminal-state*)
*OLD-TERMINAL-STATE*
Lisp> (defun pass-through-handler (function error &rest args)
        (let ((current-settings (get-terminal-modes)))
          (apply #'set-terminal-modes *old-terminal-state*)
          (apply #'universal-error-handler function error args)
          (apply #'set-terminal-modes current-settings)))
PASS-THROUGH-HANDLER
Lisp> (defun unusual-input nil
        (let ((*old-terminal-state* (get-terminal-modes))
              (*universal-error-handler* #'pass-through-handler))
          (unwind-protect (progn
                             (set-terminal-modes :pass-through t
                                                 :echo nil)
                            (get-input))
                          (apply #'set-terminal-modes
                                 *old-terminal-state*))))
UNUSUAL-INPUT
```

This example illustrates the following points:

- The call to the DEFVAR macro informs the LISP system that *OLD-TERMINAL-STATE* is a special variable.
- The first call to the DEFUN macro defines an error handler named PASS-THROUGH-HANDLER, which is used when the terminal is placed in an unusual state. The handler assumes that the normal terminal modes are stored as the value of the *OLD-TERMINAL-STATE* variable.
- The second call to the DEFUN macro defines a function named UNUSUAL-INPUT, which causes the function PASS-THROUGH-HANDLER to be the error handler while the function GET-INPUT is being executed. The GET-INPUT function is inside a call to the UNWIND-PROTECT function so an error or throw puts the terminal back in its original state.

See Chapter 6 of the *VAX LISP Implementation and Extensions to Common LISP* for more information on creating error handlers.

3.7 Getting System Messages

The GET-VMS-MESSAGE function returns the system message associated with the VMS status given as its required argument. An optional argument lets you specify which parts of the message are returned. The format of GET-VMS-MESSAGE is:

GET-VMS-MESSAGE *status* &OPTIONAL *flags*

The *status* argument is a fixnum that specifies the VMS status code of the message that will be returned. See the *VMS System Messages and Recovery Procedures Reference Manual* for information on VMS message status codes. The function returns NIL if you specify a status code that does not exist.

The *flags* argument is a bit vector of length four that specifies the content of the VMS message. The information that is included in the message when each of the four bits is set is:

Bit	Information
0	Text
1	Message ID
2	Severity
3	Facility

The default value is `#*0000`, which indicates that the process default message flags are used. For example,

```
Lisp> (get-vms-message 25)
"%SYSTEM-S-EXQUOTA, exceeded quota"
Lisp> (get-vms-message 25 #*1010)
"%S, exceeded quota"
```

3.8 Using Logical Names

A logical name is a symbolic name for any or all portions of a file specification. Logical names can provide a shorthand method for referring to commonly used files. Logical names can be defined for individual processes or for all members of a group or system. Each logical name is placed in a process, group, or system logical name table.

The `TRANSLATE-LOGICAL-NAME` function searches a logical name table for a logical name, translates it, and returns the translation(s) as a list of strings. If the logical name has no translation, the function returns `NIL`. The format of the `TRANSLATE-LOGICAL-NAME` function is:

TRANSLATE-LOGICAL-NAME *string* &KEY :TABLE :CASE-SENSITIVE

The *string* argument specifies the logical name for which the function will search. The `:TABLE` argument specifies which logical name table the function will search. The values you can specify with the `:TABLE` keyword are listed in Table 3–5.

Table 3–5: Logical Name Table Keywords

Keyword	Description
`:PROCESS`	Process name table (LNM$PROCESS_TABLE)
`:GROUP`	Group name table (LNM$GROUP)
`:SYSTEM`	System name table (LNM$SYSTEM_TABLE)
`:DECW`	DECwindows name table (DECW$LOGICAL_NAMES)
`:ALL`	All tables (LNM$DCL_LOGICAL) This is the default.

If you do not specify a table name, the process, group, system, and DECwindows tables are searched in that order.

The value of the `:CASE-SENSITIVE` keyword may be `T` (for a case-sensitive search) or `NIL` (for a case-insensitive search). The default is `NIL`. Use a value of `T` if you have multiple logical names that differ only in case.

The TRANSLATE-LOGICAL-NAME function is similar to the DCL command SHOW LOGICAL. However, while the command performs iterative translations, the function performs only one level of logical-name translation. For example:

```
Lisp> (translate-logical-name "sys$disk")
("DBA1:")
Lisp> (translate-logical-name "dba1:")
("$1$DUA1:")
```

If you use the SHOW LOGICAL DCL command to determine the translation of a logical name, it would show the translation found in all logical name tables. For example, SYS$DISK may be defined in both the process and system tables, as follows:

```
$ show logical sys$disk
   "SYS$DISK" = "LISPW$:" (LNM$PROCESS_TABLE)
   "SYS$DISK" = "$1$DUS0:" (LNM$SYSTEM_TABLE)
```

For additional information about the SHOW LOGICAL command, logical name tables, or logical names, see the *VMS DCL Dictionary*.

Chapter 4

Interacting with External Routines

The VAX LISP callout facility lets you call routines written in other languages from a LISP program. Using the callout facility, LISP programs can call (but are not linked to) routines written in languages that adhere to the VAX Procedure Calling Standard, such as:

- C and FORTRAN
- VMS and RMS system services
- Run-time library (RTL) routines

The VAX LISP callback facility makes it possible for a LISP function to be called from an external routine. To call back to LISP, the external routine must first be called from LISP. In addition, if you call out to a routine that is not a system service or in the RTL, the routine must be linked into a position-independent shareable image.

Execution of functions written in LISP depends on an entire LISP environment being present at run time. For example, garbage collection depends on this environment. If a LISP function were called from another language and ran out of dynamic memory before the LISP environment was established, it would not be able to perform the garbage collection.

Furthermore, the callout facility cannot call external routines that require an extensive, nonstandard software environment of their own. Routines written in APL and interpreted BASIC are examples of such routines. You can use VMS subprocess and mailbox facilities to communicate with such routines. VAX LISP provides functions for subprocess operations (see *VAX LISP/VMS Function, Macro and Variable Descriptions*).

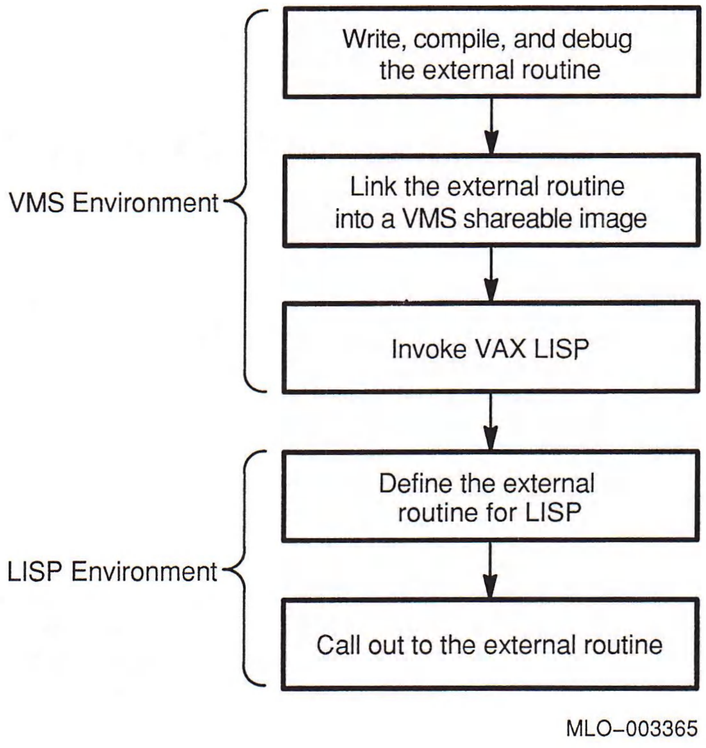
VAX LISP does not call external routines the same way other VAX languages call each other. Other VAX languages specify information about external routines by compiling code into object modules that are linked by the VMS linker. Because VAX LISP does not create linkable object modules, it must specify information about an external routine in another way. It does this by evaluating and saving the information specified by the user when defining the external routine.

To call an external routine from LISP, you must:

1. Write, compile, and debug the external routine.
2. Link it into a VMS shareable image.
3. Define its arguments and calling conventions for LISP.
4. Call it from LISP.

Figure 4–1 illustrates these steps.

Figure 4–1: Calling External Routines

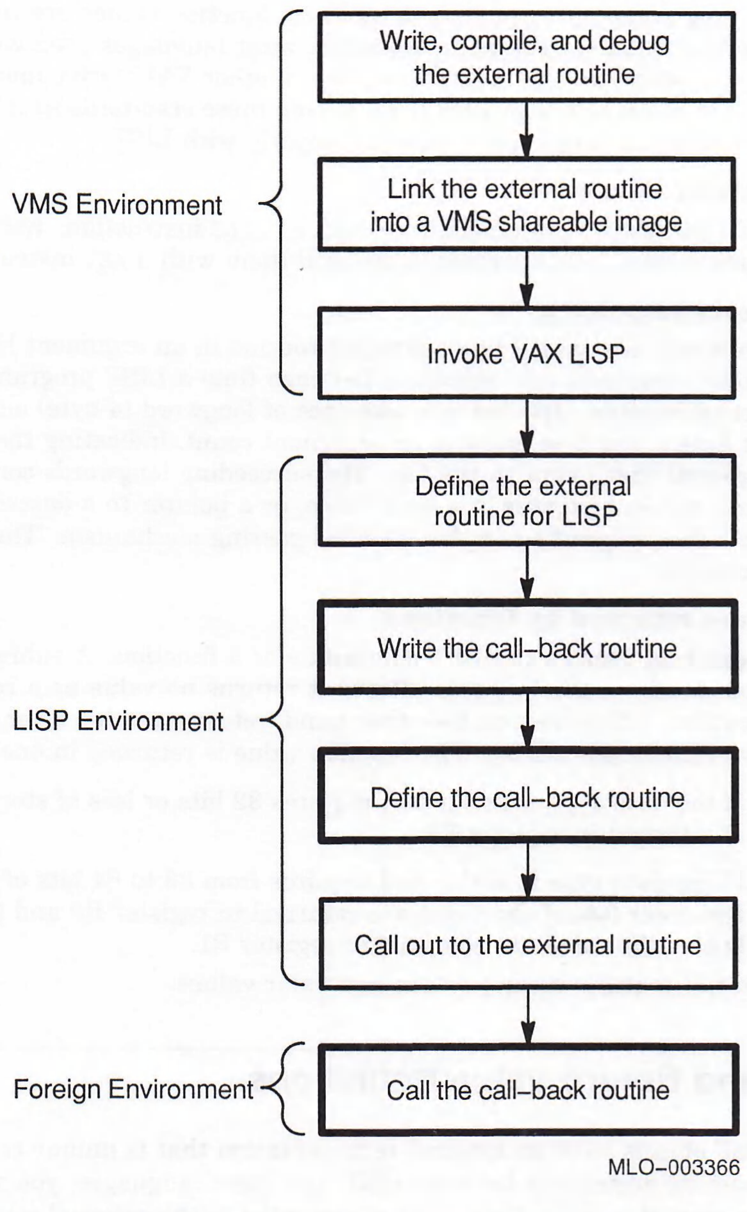

VMS Environment
Write, compile, and debug the external routine
Link the external routine into a VMS shareable image
Invoke VAX LISP
LISP Environment
Define the external routine for LISP
Call out to the external routine
MLO–003365

To call back from an external routine to LISP, you must perform these additional steps:

- Define the callback routine in LISP.
- Define and call out to an external routine.
- Call the callback routine from the external routine.

Figure 4–2 highlights these additional steps with thick-ruled boxes.

Figure 4–2: Calling Back to LISP


VMS Environment
Write, compile, and debug the external routine
Link the external routine into a VMS shareable image
Invoke VAX LISP
LISP Environment
Define the external routine for LISP
Write the call–back routine
Define the call–back routine
Call out to the external routine
Foreign Environment
Call the call–back routine
MLO–003366

The *Introduction to VMS System Routines* contains detailed information about the VAX standard for calling and passing arguments to external routines. You should be familiar with these subjects before you use the VAX LISP callout or callback facilities. The *Introduction to VMS System Routines* uses the term procedure when referring to a routine that can be called. This chapter uses the expression external routine in place of procedure to maintain consistency with VAX LISP terminology.

4.1 VAX Procedure Calling Conventions

The VAX Procedure Calling Standard defines a uniform method for routines to call one another. This standard prescribes how routines receive and return control, how arguments are passed, and how function values are returned. By means of the standard calling conventions, most languages used with the VMS operating system can call routines written in other VAX native-mode languages. You need to know how VAX LISP implements these standards so that the routines you write in other languages can work properly with LISP.

- **Transfer of control**

  VAX LISP calls external routines with a `CALLG` instruction. External routines return control to the programs that call them with a `RET` instruction.

- **Argument passing**

  Arguments are passed to an external routine in an argument list. The LISP system constructs this argument list each time a LISP program calls an external routine. The list is a sequence of longword (4-byte) entries. The first byte of the first entry is an argument count, indicating the number of longwords that follow in the list. The succeeding longwords contain either a data value, a pointer to a data value, or a pointer to a descriptor of a data value, depending on the specified passing mechanism. The limit is 254 arguments.

- **Values returned by functions**

  An external routine can be a subroutine or a function. A subroutine is invoked only to produce side effects; it returns no value as a result of execution. A function, on the other hand, returns a value after executing and may produce side effects. The function value is returned in one of two ways:

  - If the data type is scalar and requires 32 bits or less of storage, the value is returned in register R0.
  - If the data type is scalar and requires from 33 to 64 bits of storage, the low-order bits of the value are returned in register R0 and the high-order bits of the value are returned in register R1.

  External routines cannot return nonscalar values.

4.2 Argument and Return Value Definitions

VAX LISP objects have an internal representation that is unique to the language. When passing arguments between LISP and other languages, you must specify how you want the LISP object to be represented to the external routine, and vice versa.

The three characteristics of an argument that you must define before you call out to an external routine or call back to LISP are:

- **The access capability**

  An argument may have input or input/output access. Input access means that the external routine can access but not modify the argument. Input/output access means the external routine can access the argument value and may also modify it. Section 4.2.1 describes these types of access in detail.

- **The passing mechanism**

  The VAX Procedure Calling Standard defines three mechanisms by which arguments are passed to external routines:

  - By immediate value
  - By reference
  - By descriptor

  These passing mechanisms are described in Section 4.2.2.

- **The data type conversions required**

  You must specify how the LISP representation of the argument should be converted into its corresponding VAX type, and vice versa. See Section 4.2.3 for more information on data type conversions.

When defining result values, you specify only the data type conversions required for the value. The access capability and passing mechanism do not apply to results.

LISP defines default characteristics for arguments if you do not supply them. Therefore, if the default characteristics are adequate, an argument description is nothing more than the name of the argument, as in:

(*argument-name*)

An argument name is a symbol. It must either be unique within the routine's definition or `NIL` if no name is desired. Unique names make some error messages easier to understand.

When the argument has nondefault characteristics, the argument description is written as a list of options, each of which is a keyword–value pair:

```
(argument-name keyword value
            ...)
...
```

Option values are not evaluated when the argument is defined. Rather, they are evaluated when the routine is called.

For example, the `multiplicand` argument below is an input-only integer argument to be passed by reference. It can be written like this:

```
(multiplicand :access :in
              :mechanism :reference
              :lisp-type integer)
```

However, you need not include the `:ACCESS`, `:MECHANISM`, and `:LISP-TYPE` keywords in this definition because each characteristic uses the default value. Therefore, you can define the `multiplicand` argument as follows:

```
(multiplicand)
```

4.2.1 Access Capability

The `:ACCESS` keyword specifies the access capability for an argument. The possible values are `:IN` for input access (the default) and `:IN-OUT` for both input and output access. Because external routines cannot allocate LISP objects, `:OUT` is not a possible value.

If an argument has input access, it is assumed to be read only and the external routine may not modify it. If it is modified, the results are unpredictable. If an argument has `:IN-OUT` access, the external routine can obtain the argument's value and optionally modify it.

The argument must be specified in a form acceptable to `SETF`. The `CALL-OUT` macro passes the argument to the external routine and uses `SETF` to reassign the new value after the routine returns.

Arguments defined with `:IN-OUT` access must be initialized before they are used in a call to an external routine that modifies them. This is true even if the external routine never uses the value of the argument. The value used is unimportant; it is the act of initializing the argument that prepares the argument to be modified by the external routine. Failure to initialize an `:IN-OUT` argument results in an access violation when the external routine attempts to modify the argument. Once initialized, arguments can be used many times; they need not be initialized before each use with `CALL-OUT`.

For example, you could define an argument called `*flagnumber*` as follows:

```
(*flagnumber* :access :in-out)
```

Because it is an input-output variable, you must initialize it before you call out to the external routine. You can initialize the variable in any of the following ways:

```
(defvar *flag-number* -1)

(setf *flag-number* -1)

(let ((*flagnumber* -1) ...))
```

If an input-output argument is a character or a number, the modified value is made into a new LISP object that is distinct from the original argument. This ensures that when you pass constants or shared data objects, they will not be modified. If the argument is not a character or a number, the argument is directly modified by the external routine; no copy is made and no conversion is performed. This means that all array arguments are modified in place.

4.2.2 Passing Mechanism

The `:MECHANISM` keyword defines the way an argument is passed to or from an external routine. With the `:MECHANISM` keyword, you can specify one of the three values in Table 4–1.

Table 4–1: Values of the :MECHANISM Keyword

Name	VAX Mechanism	Description
`:VALUE`	Immediate Value	Passes a copy of the argument in the argument list. You can use this mechanism only for arguments that have input access and that have data types requiring no more than a longword of storage.
`:REFERENCE`	Reference	Passes the address of the argument in the argument list.
`:DESCRIPTOR`	Descriptor	Passes the address of an argument descriptor in the argument list.

You cannot specify VMS descriptor classes for arguments to external routines. The `DEFINE-EXTERNAL-ROUTINE` macro assigns an appropriate class when the LISP system evaluates the argument. The values the macro assigns are `DSC$K_CLASS_S` (scalar) or `DSC$K_CLASS_A` (array).

To pass an argument using a user-specified descriptor, you must define the descriptor and the argument as alien structures and pass the alien structure descriptor by reference. For information on defining alien structures, see Chapter 5.

4.2.3 Data Type Conversions

When calling out to an external routine, VAX LISP converts LISP data types to VAX types. Results must be converted from VAX types to LISP types when the routine returns a value to the LISP function. Similar conversions are required for callback routines.

You define the LISP data type of an argument or return value with the `:LISP-TYPE` keyword. You define the VAX data type of an argument or return value with the `:VAX-TYPE` keyword.

Table 4–2 shows the types of conversions that are available for callout and callback arguments and for callback return values. For all arguments and return values, the LISP type defaults to `INTEGER`.

The table shows one or more of the compatible VAX data types for each LISP type. When no VAX type is specified in the argument definition, LISP converts the LISP data type to the default VAX type. If the values you specify for the LISP data type and the VAX data type are incompatible, an error is signaled.

Table 4–2 also shows the valid passing mechanisms (V = Value, R = Reference, and D = Descriptor) for each data type. The default mechanism for all types except `:VAX-TYPE :TEXT` is by reference. The default for text is by descriptor. Table 4–2 also specifies the descriptor class and data type that are included in the argument descriptor when passing by descriptor. The descriptor formats, descriptor class, and data type codes are described in the *Introduction to VMS System Routines*.

Table 4–2: Data Type Conversion

LISP Type	Compatible VAX Types	Mechanisms Allowed	Descriptor Class/Data Type
CHARACTER	:UNSIGNED-BYTE†	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_BU
INTEGER	:BIT	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_LU
	:BYTE	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_B
	:UNSIGNED-BYTE	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_BU
	:WORD	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_W
	:UNSIGNED-WORD	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_WU
	:LONGWORD†	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_L

† Default VAX type

(continued on next page)

Table 4–2 (Cont.): Data Type Conversion

LISP Type	Compatible VAX Types	Mechanisms Allowed	Descriptor Class/Data Type
	:UNSIGNED-LONGWORD‡	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_LU
	:QUADWORD	R, D	DSC$K_CLASS_S DSC$K_DTYPE_Q
	:UNSIGNED-QUADWORD	R, D	DSC$K_CLASS_S DSC$K_DTYPE_QU
SHORT-FLOAT	:F-FLOATING	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_F
SINGLE-FLOAT	:F-FLOATING†	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_F
DOUBLE-FLOAT	:G-FLOATING†	R, D	DSC$K_CLASS_S DSC$K_DTYPE_G
	:D-FLOATING	R, D	DSC$K_CLASS_S DSC$K_DTYPE_D
LONG-FLOAT	:H-FLOATING†	R, D	DSC$K_CLASS_S DSC$K_DTYPE_H
STRING	:TEXT†	R, D	DSC$K_CLASS_S DSC$K_DTYPE_T
	:ASCIZ	R	
SIMPLE-BIT-VECTOR	:BIT	R, D	DSC$K_CLASS_S DSC$K_DTYPE_V
	:UNSIGNED-LONGWORD	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_LU
ALIEN-STRUCTURE	:UNSPECIFIED†	R, D	DSC$K_CLASS_S DSC$K_DTYPE_Z
(COMPLEX SINGLE-FLOAT)	:F-FLOATING-COMPLEX†	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_FC
(COMPLEX SHORT-FLOAT)	:F-FLOATING-COMPLEX†	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_FC
(COMPLEX DOUBLE-FLOAT)	:G-FLOATING-COMPLEX†	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_GC
	:D-FLOATING-COMPLEX	V, R, D	DSC$K_CLASS_S DSC$K_DTYPE_DC
(COMPLEX LONG-FLOAT)	:H-FLOATING-COMPLEX†	R, D	DSC$K_CLASS_S DSC$K_DTYPE_HC
(ARRAY CHARACTER)	:UNSIGNED-BYTE†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_BU
(SIMPLE-ARRAY BIT)	:BIT†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_V
(ARRAY (UNSIGNED-BYTE 2))	:UNSIGNED-BYTE	R, D	DSC$K_CLASS-A DSC$K_DTYPE_BU
(ARRAY (UNSIGNED-BYTE 4))	:UNSIGNED-BYTE	R, D	DSC$K_CLASS_A DSC$K_DTYPE_BU

† Default VAX type
‡ Default VAX type for callback return values

(continued on next page)

Table 4–2 (Cont.): Data Type Conversion

LISP Type	Compatible VAX Types	Mechanisms Allowed	Descriptor Class/Data Type
(ARRAY (SIGNED-BYTE 8))	:SIGNED-BYTE	R, D	DSC$K_CLASS_A DSC$K_DTYPE_B
(ARRAY (UNSIGNED-BYTE 8))	:UNSIGNED-BYTE†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_BU
(ARRAY (UNSIGNED-BYTE 12))	:UNSIGNED-BYTE	R, D	DSC$K_CLASS_A DSC$K_DTYPE_BU
(ARRAY (SIGNED-BYTE 16))	:SIGNED-WORD	R, D	DSC$K_CLASS_A DSC$K_DTYPE_W
(ARRAY (UNSIGNED-BYTE 16))	:UNSIGNED-WORD†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_WU
(ARRAY (SIGNED-BYTE 32))	:LONGWORD†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_L
(ARRAY (UNSIGNED-BYTE 32))	:UNSIGNED-LONGWORD	R, D	DSC$K_CLASS_A DSC$K_DTYPE_LU
(ARRAY (SIGNED-BYTE 64))	:QUADWORD	R, D	DSC$K_CLASS_A DSC$K_DTYPE_Q
(ARRAY (UNSIGNED-BYTE 64))	:UNSIGNED-QUADWORD	R, D	DSC$K_CLASS_A DSC$K_DTYPE_QU
(ARRAY SHORT-FLOAT)	:F-FLOATING†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_F
(ARRAY SINGLE-FLOAT)	:F-FLOATING†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_F
(ARRAY DOUBLE-FLOAT)	:G-FLOATING†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_G
(ARRAY LONG-FLOAT)	:H-FLOATING†	R, D	DSC$K_CLASS_A DSC$K_DTYPE_H

† Default VAX type

Every simple string is guaranteed to have a zero byte at the end, following the last actual character. Thus, you do not have to be concerned about adding the zero byte when passing simple strings as ASCIZ arguments.

If an ASCIZ argument has :IN-OUT access and is modified by placing a zero byte somewhere in the middle of the string, VAX LISP truncates the string at the first zero byte. You must take care of this situation yourself.

The callout facility restricts the types of values that an external routine can return. These types are shown in Table 4–3. It also lists the location of the result on return from the external routine. The default cases require that you specify only the LISP type with the :RESULT keyword. All other cases require that you specify both the LISP and the VAX types.

Table 4–3: Conversion of Callout Return Values

LISP Type	VAX Type	Location of Result
CHARACTER	:UNSIGNED-BYTE†	Low-order byte of R0
INTEGER	:BIT	R0, unsigned
	:BYTE	R0, signed
	:UNSIGNED-BYTE	R0, unsigned
	:WORD	R0, signed
	:UNSIGNED-WORD	R0, unsigned
	:LONGWORD†	R0, signed
	:UNSIGNED-LONGWORD	R0, unsigned
	:QUADWORD	R0/R1, signed
	:UNSIGNED-QUADWORD	R0/R1, unsigned
SHORT-FLOAT	:F-FLOATING†	R0
SINGLE-FLOAT	:F-FLOATING†	R0
DOUBLE-FLOAT	:G-FLOATING†	R0/R1
	:D-FLOATING	R0/R1
SIMPLE-BIT-VECTOR	:UNSIGNED-LONGWORD†	R0, unsigned

† Default VAX type

4.3 Writing and Linking an External Routine

You can call out to any function written in another language. For example, the following function written in FORTRAN, called NUMBERS, manipulates two integers and returns an integer.

```
FUNCTION NUMBERS(X, Y)
IMPLICIT INTEGER*4 (A-Z)

NUMBERS=Y * (X + Y ** X) / X
RETURN
END
```

Before you can call a function from LISP, you must compile and link it, as follows:

1. The function is compiled in the usual way:

   ```
   $ fortran numbers
   ```

2. You must link the object modules into one or more position-independent VMS shareable images, using an options file. An options file contains special instructions to the linker. When making a shareable image to be called from LISP, you must include a UNIVERSAL declaration in the options file. This declaration makes it possible to call the routine from outside the object module in which the routine is defined. The number of individual shareable images that can be mapped into VAX LISP depends on VMS shareable image restrictions and the available address space.

   To link the NUMBERS function into a shareable image, you specify the name NUMBERS as an entry point that is UNIVERSAL:

   ```
   $ link/shareable=dba2:[smith]example numbers,sys$input:/options
   universal=numbers
   CtrL/Z
   ```

In this example, SYS$INPUT is given as the name of the options file. The linker waits to read the linker options from the terminal. Ctrl/Z marks the end of the options file.

NOTE

> You cannot call external routines in a based image. A based image is one that has been assigned a base address with the BASE option. Do not use the BASE option in your linker options file.

3. If the resulting shareable image is not in SYS$SHARE, you must define a logical name that points to it. For example, the FORTRAN example is not in SYS$SHARE; it must be defined as a logical name:

   ```
   $ define example dba2:[smith]example
   ```

You can call external routines in shareable images that contain writable sections. For example, routines that are written in VAX FORTRAN, which use COMMON blocks, produce such code. A shareable image contains a writable section if the external routine contains a program section (PSECT) that has the write (WRT) and the share (SHR) attributes. To determine whether a program section in a shareable image has these attributes, examine the image's map file.

Before you can call an external routine in a shareable image that contains writable sections, you must either install the shareable image with the VMS INSTALL utility or do the following:

1. Link all routines that refer to the writable, shareable PSECT into the same shareable image in a single invocation of the VMS linker.
2. Supply an additional option to the linker that changes the attributes of the PSECT in question from writable and shareable (WRT, SHR) to writable, not shareable (WRT, NOSHR).

For example, suppose you have two routines that access a COMMON block named SHARED_SPACE. These routines exist in different source files, named FOR1.FOR and FOR2.FOR. After compiling both source files, you would use the following VMS linker command to create a shareable image:

```
$ link/share=myexe for1,for2,sys$input:/options
universal=routine1,routine2
psect_attr=shared_space,wrt,noshr
Ctrl/Z
```

The resulting shareable image may be called from VAX LISP without having to install it. However, this procedure will not work if routines that access the same writable, shareable PSECT exist in different shareable images.

The procedure for linking shareable images is explained in detail in the *VMS Linker Utility Reference Manual*.

4.4 Defining an External Routine

Because VAX LISP does not create linkable object modules, it must specify information about an external routine in another way. The DEFINE-EXTERNAL-ROUTINE macro does this by storing information about the routine in the LISP environment. The definition gives the VAX LISP system the information needed to create an argument list and locate the external routine when it is called.

When you define an external routine, LISP creates an internal data structure and associates the structure with the symbolic name of the routine. The data structure is used in the resulting LISP code and when the routine is called. Therefore, you must ensure that an external routine is defined before it is called.

The format for defining an external routine is:

```
DEFINE-EXTERNAL-ROUTINE name-and-options
  [doc-string]
  {argument-specifier}*
```

The arguments to `DEFINE-EXTERNAL-ROUTINE` describe the routine name and options, an optional documentation string, and the arguments to the function. For example, to define the external FORTRAN routine called NUMBERS, you must specify these three things:

- Where the shareable image is located. The name of the shareable image file is defined with the `:FILE` option. In Section 4.3, the shareable image's location is given the logical name EXAMPLE.
- The data type of the function's return value (an integer in the example). This is defined with the `:RESULT` option.
- The arguments to the routine. In the example, the arguments to NUMBERS are longword integers passed by reference. Because they have the default data type, they can be defined simply by giving their names.

Thus, the NUMBERS function is defined as follows:

```
Lisp> (define-external-routine
        (numbers :file "EXAMPLE"
                 :result integer)
          x y)
NUMBERS
```

Run-Time Library routines are provided by VMS in a library called LIBRTL. You can call these routines in LISP if you first define them. For example, the LIB$CREATE_DIR Run-Time Library routine creates a directory. It is defined as follows:

```
Lisp> (define-external-routine (lib$create_dir
                                    :file "librtl"
                                    :check-status-return t)
          "Call out to the lib$create_dir RTL routine"
          (dev-dir-spec :lisp-type string))
LIB$CREATE_DIR
```

The definition gives the name of the routine, the library in which it can be found, and states that LISP should check the status value returned by the routine. It also supplies a documentation string and defines the data type of the argument to the routine.

Note that if you call an external routine and then redefine it, you must exit and reenter LISP before you can use the new definition. This is because a loaded shareable image cannot be reloaded in the same process.

4.4.1 External Routine Names and Options

When you define an external routine, you must specify a name for it. In addition, you can specify options that provide the LISP system with information about how to call the external routine.

The external routine name is a symbol that uniquely identifies the routine. The name should not also be the name of a LISP function. The name serves as the entry-point name unless a different name is specified with the :ENTRY-POINT option.

You can assign specific characteristics to an external routine by specifying options in the routine's definition. Each option consists of a keyword–value pair. The option keywords are as follows:

- **The :CHECK-STATUS-RETURN Keyword**

  The :CHECK-STATUS-RETURN keyword states whether the callout facility should examine the contents of register R0 on return from the external routine. You can specify this option in the following ways:

  ```
  :check-status-return nil
  :check-status-return t
  :check-status-return n
  ```

  The default is NIL, which means that no checking is done. If you specify T, the R0 register is assumed to contain a status code. When the severity of the status is warning, error, or severe-error, a continuable error is signaled. If you specify an integer (*n*), an error is signaled when the return value is equal to *n*.

  The presence of this option implies that the external routine returns an integer; thus, you should not specify the :RESULT option when you specify this option.

- **The :ENTRY-POINT Keyword**

  The *routine-name* gives the name of the external routine as it is called from LISP. By default, LISP assumes that this is also the name of the entry point. When the names are different, use :ENTRY-POINT to supply the entry-point name of the external routine.

  For example, if you want to call the routine CREATE-DIR from LISP, you can define LIB$CREATE_DIR as follows:

  ```
  Lisp> (define-external-routine (create-dir
                                   :file "librtl"
                                   :check-status-return t
                                   :entry-point "lib$create_dir")
            "Call out to the lib$create_dir RTL routine"
            (dev-dir-spec :lisp-type string))
  CREATE-DIR
  ```

- **The :FILE Keyword**

  The :FILE keyword gives the file specification of the external routine's shareable image. By default, LISP assumes that the routine is a system service, and no :FILE specification is required. If the routine is not a system service, the :FILE specification must be either a logical name that refers to the shareable image or the name of a shareable image in the SYS$SHARE directory. The specification cannot be an arbitrary file specification.

  For example, because the NUMBERS routine is not a system service and is not located in SYS$SHARE, you must define its location as a logical name, such as:

  ```
  $ DEFINE EXAMPLE DBA2:[SMITH]EXAMPLE
  ```

- **The :RESULT Keyword**

  The `:RESULT` keyword specifies the type of value returned by the external routine. The default is `NIL`, which means that the routine is a subroutine and returns no value. Do not specify both `:CHECK-STATUS-RETURN T` and `:RESULT`.

  If the routine does return a value, then the `:RESULT` keyword can specify the LISP data type that the external routine will return. If the VAX type of the returned value does not correspond to the LISP data type, specify both the LISP and the VAX type as a list of the form:

  (:LISP-TYPE *lisp-type* :VAX-TYPE *vax-type*)

  See Table 4–3 for valid result data types.

- **The :TYPE-CHECK Keyword**

  The `:TYPE-CHECK` keyword directs LISP to check that the data types of the arguments passed to an external routine are compatible with the argument descriptions. You can specify the keyword in either of the following ways:

  ```
  :type-check nil
  :type-check t
  ```

  If you specify `T`, the code generated by the `CALL-OUT` macro checks the type of actual LISP objects at the time of the callout. If the types of the routine's defined and actual arguments are incompatible, an error is signaled. If you specify `NIL` (the default value), the system does not generate type-checking code.

  Note that type checking adds considerable overhead to the callout process.

Option values are not evaluated by `DEFINE-EXTERNAL-ROUTINE`. They are evaluated when the external routine is called.

4.4.2 Documentation String

You can include a documentation string for an external routine. The string is optional and is attached to the symbol as a documentation string of type `EXTERNAL-ROUTINE`. Place the string after the routine name and options list.

For example, the definition of LIB$CREATE_DIR contains a documentation string. When this external routine has been defined, you can access the documentation string with either of the following LISP forms:

```
Lisp> (describe 'lib$create_dir)
Lisp> (documentation 'lib$create_dir 'external-routine)
```

4.4.3 Argument Descriptions

External routines usually accept one or more arguments. The argument descriptions determine the number, order, and characteristics of the arguments that you can pass to the routine. See Section 4.2 for information on defining the arguments to an external routine.

4.5 Calling an External Routine

You call an external routine with the VAX LISP CALL-OUT macro, supplying the name of the external routine and its arguments. These must be compatible with the arguments defined in the call to the DEFINE-EXTERNAL-ROUTINE macro. Make sure the definition is evaluated before calling out to that external routine.

The format for calling an external routine is:

CALL-OUT *routine-name arg1 arg2 . . .*

The CALL-OUT macro produces code that performs the following operations:

1. Checks all arguments, if the :TYPE-CHECK option is specified.
2. On the first call to an external routine, reads the routine into memory.
3. Creates an argument list, using the arguments provided.

   If you specify fewer arguments to the CALL-OUT macro than are defined for the callout routine, the remaining arguments are not included in the argument list. The count in the first longword of the list reflects this situation. If you specify more arguments than are defined, an error is signaled.

   If an argument evaluates to NIL, a zero is placed in the corresponding argument list. Zero normally means that an optional argument is not desired.
4. Transfers control to the external routine with the CALLG instruction.
5. Returns any specified result from the external routine, or no values if there is no result.

For example, you can call out to the NUMBERS external routine with the CALL-OUT macro, as follows:

```
Lisp> (call-out numbers 5 7)
23536
```

The following LISP form calls out to the external routine LIB$CREATE_DIR defined in Section 4.4. This call creates the new directory LISPW$:[MYNAME.WORK]:

```
Lisp> (call-out lib$create_dir "lispw$:[myname.work]")
561
```

4.6 Calling System Services

The callout facility provides a mechanism for LISP programs to call standard VMS and RMS system services. For information about VMS system services, see the *VMS System Services Reference Manual*. For information about RMS services, see the *VMS Record Management Services Manual*.

4.6.1 Defining System Services

Defining VMS and RMS system services is similar to defining other external routines, with a few restrictions:

- You must omit the `:FILE` option from the `DEFINE-EXTERNAL-ROUTINE` specification. When you omit this option, LISP assumes that the external routine is a system service. Otherwise, LISP assumes you are referring to an entry point in an ordinary shareable image of that name.

  The exception to this rule is any system service that resides in a shareable image, such as SYS$MOUNT, which resides in MOUNTSHR. In this case, the system service is defined as any other external routine in a shareable image.

- The `:ENTRY-POINT` name in the `DEFINE-EXTERNAL-ROUTINE` specification must be a valid system service entry point.

- The order and number of arguments in the `DEFINE-EXTERNAL-ROUTINE` specification must correspond to the order and number specified by the system service's definition.

For example, the SYS$DALLOC system service deallocates a device. It returns a longword condition value. It takes two arguments. The *devnam* argument is the address of a string; *acmode* is a longword passed by value. You would define SYS$DALLOC as follows:

```
Lisp> (define-external-routine (sys$dalloc :result integer)
        (devnam :lisp-type string)
        (acmode :lisp-type integer :mechanism :value))
SYS$DALLOC
```

The following example defines the SYS$RENAME RMS system service, which changes the name, type, or version of a file, or moves the file to another directory. Assume the LISP variables `OLD` and `NEW` are bound to statically allocated alien structures that are the file attribute blocks to be used in a rename operation. The SYS$RENAME system service is defined as follows:

```
Lisp> (define-external-routine (sys$rename :result integer)
        (old-fab :lisp-type alien-structure)
         nil     ;error and success routines
         nil     ;will not be used
        (new-fab :lisp-type alien-structure))
SYS$RENAME
```

4.6.2 Calling System Services

Calling VMS and RMS system services is similar to calling other external routines with a few restrictions:

- You must always call a system service with a complete argument list. Put `NIL` in place of the omitted arguments—including omitted trailing arguments.

- Many system services require data structures that are filled in or modified at a later time. For example, the I/O status block argument (iosb) to the SYS$QIO system service is filled in by the system service with the completion status of the QIO request. Data structures such as this must be created as alien structures and statically allocated so that they will not be moved by VAX LISP (see Chapter 5).

- You cannot refer to VMS symbolic constants (such as return status values or field offsets) by their symbolic names. LISP has no knowledge of these constants. Definitions for many of these constants can be obtained from the definition files in the LISP$EXAMPLES directory. You probably want to include only those definitions that you require. The *VMS System Services Reference Manual* describes the necessary definitions for each System Service call.

For example, to deallocate the device named _tth7, you can call SYS$DALLOC as shown below. NIL is specified to account for the omitted argument.

```
Lisp> (call-out sys$dalloc "_tth7:" nil)
2312
```

To rename a file with SYS$RENAME, you can call the RMS system service as shown below. NIL is specified to account for the omitted arguments. This ensures that the correct number of arguments are specified. NIL is specified for the error and status routines because ASTADR arguments must be omitted. The call returns 1, indicating success.

```
Lisp> (call-out sys$rename old nil nil new)
1
```

Definitions for most system services and many other library routines can be found in LISP$EXAMPLES.

4.7 Creating a Callback Function

A callback function is a LISP function that can be called from an external routine. A callback function must be compiled. It executes in the full LISP environment. All the usual LISP facilities are available while the routine is executing, including the error handling system and the VAX LISP Debugger.

The LISP functions THROW and CATCH are available for transferring control between callback and non-callback LISP code. For example, a mainline LISP function can establish a CATCH, then call out to an external routine. This external routine can then invoke a callback routine, which issues a THROW to the CATCH tag established in the mainline code. Note that this implies cooperation on the part of the external code. Because a THROW must transfer control to the CATCH code, the external code must be prepared to signal and unwind when it encounters any signal that it does not define and handle itself.

You define a callback routine with MAKE-CALL-BACK-ROUTINE. You pass the alien structure returned by this call as an argument to an external routine, which can then invoke the callback function, using VAX standard calling conventions. The callback routine is valid across garbage collections, but you must ensure that any arguments you pass in the callout or callback are either in LISP static space or in foreign data space.

The MAKE-CALL-BACK-ROUTINE function has the following syntax:

MAKE-CALL-BACK-ROUTINE *function*
&KEY :ARGUMENTS *argument-specifier*
:RESULT *result-specifier*

The arguments to MAKE-CALL-BACK-ROUTINE describe the function name, the arguments to be passed to the function, and the data type of the result passed back by the function.

4.7.1 Callback Function Names

The name of a callback function is defined with the *function* argument to MAKE-CALL-BACK-ROUTINE. It must be a symbol or a function object. Symbols are useful if the specified function is later redefined or if you have not defined the function before the call to MAKE-CALL-BACK-ROUTINE.

4.7.2 Callback Arguments

The :ARGUMENTS keyword defines the argument passing mechanism, access capability, and data type conversions for each argument to the callback function. The value of the :ARGUMENTS keyword can be one of the following:

- NIL indicates that the callback routine takes no arguments.
- :AP indicates that the actual VAX argument list is passed to the callback routine as the only parameter. All arguments in the list must be accessed by the callback routine, using alien-structures. This is the default method for passing arguments.
- A list of argument definitions that follows a syntax similar to that for DEFINE-EXTERNAL-ROUTINE. See Section 4.2 for information on defining arguments.

You cannot specify both the :AP keyword and a list of argument descriptions.

When a callback argument has :IN-OUT access, the callback function returns multiple values. The LISP code for the callback routine should implement multiple return values, using the VALUES function. The first value in the list must be the result. Each successive return value is mapped to the :IN-OUT arguments in the order in which they are defined. On return from the callback function, if there are more values returned than :IN-OUT arguments, the extra values are discarded. If there are fewer values than arguments, no values are assigned to the extra arguments. Note that :IN-OUT arguments must be initialized before calling out to an external routine. Therefore, extra arguments retain their initial values.

When you use the :AP mechanism, the argument can be modified in place, using alien structures; any return values beyond the first are ignored.

4.7.3 Callback Return Values

The :RESULT keyword to the MAKE-CALL-BACK-ROUTINE function specifies the data type of the return value and data type conversion mechanism from LISP to VAX data types. If the LISP function returns multiple values, the result must be the first value in the VALUES list.

The *result-specifier* defines only the :VAX-TYPE and :LISP-TYPE of the result, as described in Section 4.2. :ACCESS and :MECHANISM keywords do not apply to result specifiers. The default value is NIL, meaning that the callback routine returns no value.

4.7.4 Writing a Callback Routine

Any external routine called from LISP can call back to LISP. This section describes a C routine that calls back to a LISP routine. The C routine accepts two arguments: a CALL-BACK-ROUTINE alien structure and an argument to pass to the callback routine. The C function calls the callback routine, passing the argument from LISP and some C data. The callback routine modifies the LISP argument. The modified value is passed back to the C routine, which in turn passes it back to the mainline LISP code. The returned value is two times the input value because it is multiplied by two in the callback routine.

The C code must be compiled and linked into a shareable image, as follows:

1. Assuming the C source code is contained in the file called CTEST.C, compile the C source code with the command:

   ```
   $ cc ctest
   ```

2. Link the C source code as a shareable image called TEST.EXE:

   ```
   $ link/share=test ctest,sys$input:/option
   universal=int_test
   sys$library:vaxcrtl.exe/share
   Ctrl/Z
   ```

3. Define the shareable image file as a logical name:

   ```
   $ define test usr:[your.directory]test
   ```

The C routine is shown in Example 4–1.

Example 4–1: Calling an External Routine from C

```
#include <stdio.h>

int_test (lisp_func, lisp_arg)
  int (*lisp_func)();      /* An alien structure passed by reference */
  int *lisp_arg;           /* An integer passed by reference */
{
  int call_back_result;
  int foreign_arg;

  printf ("\nint_test args:  func= %x  arg= %d\n",
                             lisp_func, *lisp_arg);

  foreign_arg = 99;
  call_back_result = (*lisp_func)(foreign_arg, lisp_arg);

  printf ("\nint_test values after call_back: result=%d  arg=%d\n",
                             call_back_result, *lisp_arg);

  return (call_back_result);
}
```

Before you can call the C routine from LISP, you must define it with the DEFINE-EXTERNAL-ROUTINE macro. Note that the second argument is given the :IN-OUT access because it is modified by the callback routine:

```
(define-external-routine (int_test  :file "TEST"    :result integer)
  (func :access :in
        :mechanism :reference
        :lisp-type alien-structure)
  (arg  :access :in-out
        :mechanism :reference
        :lisp-type integer))
```

You can pass arguments to a callback routine as argument specifiers or as an argument list. The following sections show both methods.

4.7.4.1 Passing Arguments to a Callback Routine

A callback function named INTEGER-CALL-BACK is shown in Example 4–2. It receives two integer arguments. The first is the integer passed from C. The second is the :IN-OUT argument passed to C from LISP. All callback routines with :IN-OUT arguments must return the modified arguments as multiple values in a VALUES form. The first value is interpreted as the result of the callback routine invocation. The second is associated with the :IN-OUT parameter. Any number of return values may be passed, corresponding to IN-OUT arguments in the external routine. This callback routine multiplies the second argument by two and passes the result back as the second parameter to VALUES.

Note the EVAL-WHEN form; all callback routines must be compiled.

Example 4–2: Accessing Callback Arguments

```
(defconstant return-value 17)

(defun integer-call-back (arg1 arg2)
  (format t "~%~%call-back args:  arg1= ~d  arg2= ~d~%" arg1 arg2)
  (setf arg2 (* 2 arg2))
  (values return-value arg2))

(eval-when (eval) (compile 'integer-call-back))
```

In Example 4–3, the MAKE-CALL-BACK-ROUTINE macro defines the name of the callback function (INTEGER-CALL-BACK) and the order of the arguments, as well as information about type and access characteristics. As mentioned above, the second argument is defined to have :IN-OUT access.

Example 4–3: Defining Arguments to a Callback Routine

```
(defvar my-call-back (make-call-back-routine
              #'integer-call-back
              :arguments
                  '((arg1 :lisp-type integer
                      :access :in
                      :mechanism :value
                      :vax-type :unsigned-longword)
                    (arg2 :lisp-type integer
                      :access :in-out
                      :mechanism :reference))
              :result
                  '(:lisp-type integer)))
```

The external routine must be invoked from LISP. In Example 4–4, a function called TRY-CALL-BACK performs the callout.

Example 4–4: Calling Out and Back, Passing Arguments

```
(defun try-call-back (val)
  (let ((result nil))
    (setf result (call-out int_test my-call-back val))
    (format t "~%~%callout result= ~d new value= ~d~%"
              result val)
    result))
```

TRY-CALL-BACK can be invoked with an integer argument. The argument is passed through C code, into the callback routine, out again through C, and back into the mainline LISP code. Given an input value of 7, TRY-CALL-BACK produces the following output:

```
Lisp> (try-call-back 7)

int_test args:  func= 851828  arg= 7

call-back args:  arg1= 99  arg2= 7

int_test values after call_back: result=17  arg=14

callout result= 17 new value= 14
17
Lisp>
```

4.7.4.2 Passing an Argument List to a Callback Routine

Example 4–5 uses the same C function to call back to LISP. The callback routine described here takes a single argument: a VAX argument list. To access the individual arguments, the program must use alien structures to access the argument list (the AP alien structure) and dereference it (the DEREF alien structure).

Example 4–5: Dereferencing an Argument List

```
(define-alien-structure (ap (:copier nil) (:predicate nil))
      (numargs :unsigned-integer 0 1)
      (arg :unsigned-integer 4 8 :offset 4 :occurs 256))

(define-alien-structure (deref (:copier nil) (:predicate nil))
      (val :unsigned-integer 0 4))
```

The INTEGER-BY-AP function shown in Example 4–6 is the callback routine. The call to the MAKE-AP alien structure constructor function uses the :DATA option. This indicates that the alien structure will directly access the argument list in non-LISP space. The constructor function MAKE-DEREF also uses the :DATA option to access the first argument in the list. If the second argument were passed by value instead of reference, this extra indirection through the DEREF alien structure would not be necessary. It could be accessed either by (ALIEN-FIELD AP :UNSIGNED-LONGWORD 4 8) or (AP-ARG AP 1).

INTEGER-BY-AP then sets the value of that argument, prints the results to the terminal, and returns the value of the constant RETURN-VALUE to the external routine.

Example 4–6: Accessing an Argument List

```
(defconstant return-value 17)

(defun integer-by-ap (arg-pointer)
      (let* ((ap (make-ap :data arg-pointer))
             (arg2 (make-deref :data (ap-arg ap 1))))
        (setf (deref-val arg2) (* 2 (deref-val arg2)))
        (format t "~%    new arg2= ~d~%~%" (deref-val arg2)))
      (values return-value))

(eval-when (eval) (compile 'integer-by-ap))
```

The TEST function shown in Example 4–7 defines the callback routine, calls out to the C routine, and prints the return values from the C routine. This function does the same thing as TRY-CALL-BACK, defined in Example 4–4.

Example 4–7: Calling a Callback Routine with an Argument List

```
(defun test (val)
  (let* ((lisp-call-back-routine
                    (make-call-back-routine
                        'integer-by-ap
                        :arguments :ap
                        :result '(:lisp-type integer)))
         (temp val)
         (res nil))
    (setf res (call-out int_test lisp-call-back-routine temp))
    (format t "~%~%lisp:   result= ~d~
              ~% modified value= ~d~
              ~% expected value= ~d~%"
              res temp (* 2 val))))
```

You can invoke TEST as follows to see how LISP passes data between callout and callback routines:

```
Lisp> (test 7)
int_test args:  func: 827f90  arg = 7
    new arg2 = 14

int_test values after call_back: result = 17 arg = 14

lisp:    result = 17
  modified value = 14
  expected value = 14
NIL
```

4.7.5 Restrictions on Callback

The restrictions on using the callback facility are:

- **Unsafe Routines**

  You should keep LISP data in static space if it is to be used by external routines and callback routines. LISP considers all foreign data to be static. When external routines keep pointers to LISP data in dynamic memory, there is no adequate way to update these pointers in the event of a garbage collection.

- **LISP Return Value**

  If LISP returns the address of a descriptor that points to a string and if you need that string through further invocations of LISP, then you should copy it to your own non-LISP area (static space). Objects returned by reference or by descriptor have a lifetime that expires at the next garbage collection. Once a garbage collection occurs, both the descriptor and the string may disappear. The exception to this rule is when an object is returned by reference and the LISP object pointed to resides in LISP static space.

- **AST Level**

  Do not invoke callback functions at AST level.

- **Alien Structures**

  If you give an alien structure as the `:LISP-TYPE` in an argument specifier, you must already have defined the alien structure.

- **Suspend and Resume**

  Do not use a suspend during a callback function. The machine state at suspend time will not necessarily be present at resume time. For example, logical names may no longer be present, shareable images may be loaded at different addresses, externally allocated memory will no longer be present, and opened files will be closed. (See Section 4.10 for details on suspending a LISP system that contains external routine definitions.)

4.8 Errors During External Routine Execution

Errors that occur during the activation or the execution of an external routine are trapped by the VAX LISP error handler. The types of errors that may occur during these operations include VMS errors that occur while accessing a shareable image, and error conditions that the external routine signals by way of the VMS error-signaling mechanism. You cannot correct these errors.

Note that the VAX LISP error handler regards signaled operating system conditions as fatal errors. This includes conditions that have a success status. However, status codes returned by an external routine do not always represent uncorrectable errors. Values specified with the `:CHECK-STATUS-RETURN` keyword determine which operation the callout facility performs when a routine returns a status code. These values are as follows:

- If the value is `T`, the LISP system examines the contents of register R0 and interprets the routine's return value as a VMS status code or a user status code.
- If the severity of the return value is warning, error, or severe-error, the LISP system signals a continuable error.
- If the value is `NIL`, all status codes are ignored.
- If the value is an integer, an error is signaled if the return value is equal to that value.

The error message "Key not found in tree" may occur during execution of a call to an external routine. This means that the `CALL-OUT` macro was unable to locate the entry point specified with the `DEFINE-EXTERNAL-ROUTINE` macro. The macro specification may be incorrect or the entry point may not have been specified in the `UNIVERSAL` option to the VMS linker when the shared image was created.

4.9 Debugging an External Routine

You can use the VMS symbolic debugger to debug external routines that you write. For example, reconsider the FORTRAN NUMBERS function definition:

```
FUNCTION NUMBERS(X,Y)
IMPLICIT INTEGER*4 (A-Z)

NUMBERS=Y * (X + Y ** X) / X
RETURN
END
```

To use the symbolic debugger with this program, you may follow these steps:

1. Compile and link the file containing the external routine with the /DEBUG qualifier. For example:

   ```
   $ fortran/debug numbers
   $ link/debug/shareable=example numbers,sys$input:/options
   universal=numbers
   Ctrl/Z
   ```

2. If the resulting shareable image file is not in SYS$SHARE, define a logical name that points to the shareable image file, as in:

   ```
   $ define example dba2:[smith]example
   ```

3. Invoke LISP.

4. Define the external routine, using the logical name you defined in step 2 as the value of the :FILE option:

   ```
   Lisp> (define-external-routine
           (numbers :file "example"
                    :result integer)
             x y)
   NUMBERS
   ```

5. Use the VMS-DEBUG function to activate the shareable image, invoke the symbolic debugger, and set the debugger to that image.

   If you call VMS-DEBUG without arguments at this point, the symbolic debugger starts, but you cannot access symbols in your shareable image because it has not yet been activated. The image is activated either when a routine in the image is first called with CALL-OUT or when a call to VMS-DEBUG specifies a routine in the image as the value of the :EXTERNAL-ROUTINE keyword. Once the routine is activated, a breakpoint can be set at routines in that image. Therefore, use the VMS-DEBUG function to invoke the symbolic debugger and set a breakpoint at the external routine. Use the :EXTERNAL-ROUTINE argument to VMS-DEBUG to specify the external routine to be debugged.

   ```
   Lisp> (vms-debug :external-routine 'numbers)

                   VAX DEBUG Version V4.4-4

   %DEBUG-I-NOGLOBALS, some or all global symbols not accessible
   %DEBUG-I-INITIAL, language is BLISS, module set to 'ENVIRVMS'
   ```

6. Use symbolic debugger commands to set the language in which the debugger operates and to establish breakpoints and tracepoints in your external routine:

   ```
   DBG> set language fortran
   DBG> set break numbers
   ```

7. Use the GO command to return control from the symbolic debugger to VAX LISP.

   ```
   DBG> go
   NIL
   Lisp>
   ```

8. Call out to your external routine. If you have established a breakpoint, execution will stop at that point and you will be able to enter debugger commands. Similarly, the symbolic debugger will report execution of tracepoints and modification of watchpoints.

   Instead of entering symbolic debugger commands interactively, you can supply them with the :COMMAND-LINE argument to VMS-DEBUG:

   ```
   Lisp> (vms-debug :external-routine 'numbers :command-line
           "set language fortran; set break numbers; go")

                   VAX DEBUG Version V4.4-4

   %DEBUG-I-NOGLOBALS, some or all global symbols not accessible
   %DEBUG-I-INITIAL, language is BLISS, module set to 'ENVIRVMS'
   Lisp>
   ```

   Now, when you call out to the external routine NUMBERS, the symbolic debugger will gain control at the breakpoint you have set. For example:

   ```
   Lisp> (call-out numbers 2 3)
   %DEBUG-I-DYNMODSET, setting module NUMBERS
   break at routine NUMBERS
        1:         FUNCTION NUMBERS(X,Y)
   DBG> examine x
   NUMBERS\X:       2
   DBG> step
   stepped tc NUMBERS\%LINE 5
        5:         RETURN
   ```

9. When you have finished entering symbolic debugger commands, terminate the external routine by entering the symbolic debugger GO command. The external routine returns control to VAX LISP.

   ```
   DBG> go
   16
   Lisp>
   ```

10. Correct errors in the external routine, then return to step 1. *You must exit from LISP; otherwise, the new shareable image you create cannot be activated because the old one cannot be deactivated.*

To correct errors in an external routine, exit from LISP, edit the routine, compile it, and link it. If you do not exit from LISP, the old shareable image (the one with bugs) cannot be deactivated.

When you end a LISP session in which you have used VMS-DEBUG, control passes to the symbolic debugger instead of to DCL. Use the symbolic debugger EXIT command to return to DCL:

```
Lisp> (exit)
%DEBUG-I-EXITSTATUS, is '%LISP-S-SUCCESS, Standard successful
completion.'
DBG> exit
$
```

See the *VMS Debugger Manual* for a complete description of the VAX/VMS symbolic debugger.

4.10 Suspending a LISP System Containing External Routine Definitions

You can suspend an executing LISP system that contains external routine definitions or calls to external routines. When you suspend such a system, you must obey certain restrictions to ensure that the resumed system operates correctly. These restrictions exist because mapped images or memory acquired from outside the LISP environment (with LIB$GET_VM) are unmapped when the LISP system exits; they cannot be automatically remapped during a resume operation. Defined external routines are automatically remapped the next time the external routine is called. If you are not aware of the restrictions, other side effects may create undesirable results caused by the following factors:

- **Open files**

  When you exit the LISP system, open files are closed. A resume operation does not reopen files that were opened by external routines.

- **Undefined logical names**

  Logical names used in the call to the `DEFINE-EXTERNAL-ROUTINE` macro must still be defined when you resume a suspended system.

- **Memory acquired with LIB$GET_VM**

  Memory acquired with the VMS LIB$GET_VM function in an external routine is deleted when you exit the LISP system. This memory is not remapped by a resume operation. Therefore, you cannot store data in acquired memory between calls across a suspend/resume cycle. Many RTL routines, for example, use such memory, and you cannot resume the routines.

- **Data initialization**

  When an external routine contains code that sets flags for initialization and takes branches based on those flags, the flags are reset when the routine's image is remapped. As a result, the first time you call the routine after a resume operation, the routine executes as if it were executing for the first time.

  If you want to retain data across a suspend/resume cycle, do not write code that depends on a first-time flag. Use one of the following methods:

  - Retain data as individual LISP objects that are passed to external routines.
  - Store data in alien structures.

  Undesired side effects do not occur if external routines are defined in a series with the `DEFINE-EXTERNAL-ROUTINE` macro and the resulting system is suspended before a call to an external routine. The VAX LISP system retains the information the external routine definition provides.

- **Shareable Images**

  A shareable image cannot be guaranteed to be reloaded at the same address when a system is resumed. All references to shareable images must be position independent to allow proper use in a resumed LISP session.

Chapter 5

Defining and Creating Alien Structures

A structure in Common LISP is a collection of fields and field values. It is similar to a record in Pascal or a structure in C and is a useful data-management tool. See *Common LISP: The Language* for a full explanation of structures.

An alien structure is a VAX LISP data type used to exchange data between LISP programs and external routines that use VAX data structures, which LISP code cannot ordinarily access. Similar to a Common LISP structure, the definition of an alien structure defines a number of functions to create alien structures, access fields or slots, and so on.

The term "alien" in "alien structure" refers to the structure's double purpose:

- To access data that is foreign to LISP
- To make LISP data available outside of LISP

Typical alien structures are represented internally as byte-aligned collections of integers, floating-point numbers, strings, and bit vectors.

VAX LISP provides macros that let you define, create, and access alien structures. These macros are used primarily with the VAX LISP callout facility (see Chapter 4). They are used to create argument values for external routines whose arguments or control blocks are too complicated for the callout facility to convert.

This chapter describes:

- How to define an alien structure, using the `DEFINE-ALIEN-STRUCTURE` macro
- Components of an alien structure definition
- Alien structure field descriptions
- How to create alien structures

In addition to the `DEFINE-ALIEN-STRUCTURE` macro, VAX LISP provides the following alien structure macro and functions:

Macro/Function	**Description**
`DEFINE-ALIEN-FIELD-TYPE` macro	Defines alien structure field types.
`ALIEN-STRUCTURE-LENGTH` function	Returns the length in bytes of an alien structure.
`ALIEN-FIELD` function	Accesses arbitrary fields in an alien structure.
`ALIEN-DATA` function	Dereferences a pointer to an alien structure data vector.

See *VAX LISP/VMS Function, Macro and Variable Descriptions* for summaries of all the alien structure functions and macros.

5.1 Defining an Alien Structure

Before you can create an alien structure, you must define it with the VAX LISP DEFINE-ALIEN-STRUCTURE macro. This macro is similar to the DEFSTRUCT macro described in *Common LISP: The Language*. Both DEFINE-ALIEN-STRUCTURE and DEFSTRUCT create a new compound data type (the data type contains more than one named component), constructor, accessor, copier, predicate, and print functions. However, the DEFSTRUCT macro defines a type containing only LISP objects; the DEFINE-ALIEN-STRUCTURE macro defines a type containing both LISP and non-LISP objects. The LISP system treats the alien structure definition as a data type that you can then use to create individual structures (instances) of that type.

The format of an alien structure definition is:

DEFINE-ALIEN-STRUCTURE *name-and-options*
    [*doc-string*]
    {*field-description*}*

For example:

```
(define-alien-structure space
  "An example alien structure definition"
  (area-1 :signed-integer 0 4)
  (area-2 :signed-integer 4 8))
```

This example defines an alien structure named SPACE. It is an object consisting of two fields: AREA-1 and AREA-2. These fields are stored internally as VAX 32-bit integers (4 bytes). The numbers 0 4 in AREA-1 and 4 8 in AREA-2 specify the structure's field positions and lengths in bytes.

The name you give to the alien structure becomes a LISP data type. When the LISP system evaluates the definition of an alien structure, the DEFINE-ALIEN-STRUCTURE macro automatically defines the type. Alien structure instances consist of the length of the structure and a pointer to a data vector. A data vector is an array of unsigned 8-bit bytes that is long enough to hold all the fields in the structure. Formatting of the vector is done by the data type keywords in the field descriptions.

In addition to creating the new data type, DEFINE-ALIEN-STRUCTURE also creates the following functions specifically for that data type:

- **Constructor function**

  A constructor function, whose default name is the new data-type name with the prefix "MAKE-", is created. A constructor function is used to create instances of the alien structure. For example, the preceding definition automatically creates a constructor function named MAKE-SPACE. You would use this function to create structures of type SPACE. See Section 5.5 for information on keyword arguments the constructor function accepts.

- **Accessor functions**

  Accessor functions can access the data in each field of the defined alien structure. By default, the DEFINE-ALIEN-STRUCTURE macro names each accessor function by prefixing each data field name with the name of the alien structure and a hyphen.

In the preceding example, the accessor functions SPACE-AREA-1 and SPACE-AREA-2 are created automatically. These 1-argument functions return the LISP integers corresponding to the VAX integers stored in the fields AREA-1 and AREA-2. Although these functions have only one argument, an accessor function can have one or two arguments, depending on the complexity of the field. For example, it can take an index with which to access individual elements of a repeating field.

These accessor functions are acceptable access forms in a call to the SETF macro (unless :READ-ONLY T was specified as a field option—see Section 5.4.4).

- **Copier function**

  A copier function, whose default name is the new data-type name beginning with the prefix "COPY-", is created. A copier function is a 1-argument function that can make a copy of an alien structure instance. This copy is not a copy of a structure's definition, but a copy of an instance of an alien structure.

  For example, the preceding definition creates a copier function named COPY-SPACE. This function is a 1-argument function that returns a copy of its argument if the argument (the alien structure) is of type SPACE.

  It is sometimes useful to preserve a copy of an alien structure before passing it to a routine that modifies it destructively.

- **Predicate function**

  A predicate, whose default name is the new data-type name ending with the suffix "-P", is created. A predicate is a 1-argument function that determines whether its argument is an instance of the defined alien structure. For example, the preceding definition automatically creates a 1-argument predicate named SPACE-P. This function returns T if its argument is of type SPACE.

- **Print function**

  A print function is created. However, this function prints only the memory address of an individual structure. This print function does not print the contents of an alien structure's data fields. For example, the following line would be displayed on your output device as the value of an individual alien structure having the default print function:

  ```
  #<Alien Structure SPACE #x5036E8>
  ```

  The initial pound (#) character and the angle brackets (< >) are part of the standard Common LISP syntax used to print nonreadable objects. The name Alien Structure identifies the object as an alien structure. The word SPACE identifies the structure's user-defined data type.

  If you want the print function to show the data in an alien structure, you must specify your own print function. See Section 5.2.5 on specifying a print function.

5.2 Alien Structure Name and Options

When you define an alien structure, you must specify a name for the structure. In addition, you can specify options that apply to the structure as a whole and a documentation string.

When specifying the alien structure's name without options, specify it as a symbol as in the preceding definition of type SPACE. For example:

```
(define-alien-structure space
   . . . )
```

NOTE

> If you use the same symbol as the name of an alien structure (DEFINE-ALIEN-STRUCTURE) and as the name of an ordinary structure (DEFSTRUCT), LISP signals a warning message stating that you have redefined the type.

By specifying options in the name field of an alien structure's definition, you can perform the following operations:

- Change the default name of the constructor function
- Change the default names of the accessor functions
- Change the default name of the copier function
- Change the default name of the predicate function
- Specify your own print function

You can also request that the constructor, accessor, copier, and predicate functions not be generated at all.

Specify an option as a list that contains a keyword and a symbol value. You can specify more than one option at a time. The format is:

(*name* (*keyword value*) . . .)

The next sections explain each keyword in detail.

5.2.1 Naming the Constructor Function

By default, the DEFINE-ALIEN-STRUCTURE macro produces a name for an alien structure's constructor function by prefixing the string "MAKE-" to the alien structure's name. For example, the default name of the constructor function created by the preceding definition is MAKE-SPACE.

To change the default name of a constructor function, specify the :CONSTRUCTOR keyword with the symbol you want in your alien structure definition. For example:

```
Lisp> (define-alien-structure (space (:constructor create-space))
        (area-1 :unsigned-integer 0 4)
        (area-2 :unsigned-integer 4 8))
SPACE
```

The LISP system does not prefix your new name with the name of the structure. For example, when the LISP system evaluates the preceding definition, the macro names the constructor function CREATE-SPACE.

If you specify NIL with the :CONSTRUCTOR keyword, the DEFINE-ALIEN-STRUCTURE macro does not define a constructor function and you cannot create alien structures of that type.

NOTE

> Alien structure constructor functions do not take an argument list, although DEFSTRUCT constructor functions do take an argument list.

5.2.2 Naming Accessor Functions

The DEFINE-ALIEN-STRUCTURE macro produces default names for an alien structure's accessor functions by prefixing each field name with the name of the alien structure and a hyphen. For example, the default names of the accessor functions created by the preceding definition are SPACE-AREA-1 and SPACE-AREA-2.

To change the default names of an alien structure's accessor functions, specify the :CONC-NAME (concatenated name) keyword with a string or symbol for the prefix you want in your alien structure definition. For example:

```
Lisp> (define-alien-structure (space (:conc-name "galaxy-"))
        (area-1 :unsigned-integer 0 4)
        (area-2 :unsigned-integer 4 8))
SPACE
```

When the LISP system evaluates the preceding definition, the DEFINE-ALIEN-STRUCTURE macro generates accessor functions named GALAXY-AREA-1 and GALAXY-AREA-2. If you specify NIL with the :CONC-NAME keyword, the function names are the same as the field names, AREA-1 and AREA-2.

Use the accessor functions with SETF to change the value of a field.

5.2.3 Naming the Copier Function

By default, the DEFINE-ALIEN-STRUCTURE macro produces a name for an alien structure's copier function by prefixing the string "COPY-" to the alien structure's name. For example, the default copier function of the preceding definition is COPY-SPACE.

To change the name of the copier function, specify the :COPIER keyword with the symbol you want in your definition of an alien structure. For example:

```
Lisp> (define-alien-structure (space (:copier reproduce-space))
        (area-1 :unsigned-integer 0 4)
        (area-2 :unsigned-integer 4 8))
SPACE
```

When the LISP system evaluates the preceding definition, the DEFINE-ALIEN-STRUCTURE macro generates a copier function named REPRODUCE-SPACE.

If you specify NIL with the :COPIER keyword, the DEFINE-ALIEN-STRUCTURE macro does not define a copier function.

5.2.4 Naming the Predicate Function

By default, the DEFINE-ALIEN-STRUCTURE macro produces the name of the predicate function by attaching the string "-P" to the end of the alien structure's name. For example, the default name of the predicate function created by the preceding definition is SPACE-P.

To change the name of the predicate function, specify the :PREDICATE keyword with a symbol in your definition of an alien structure. For example:

```
Lisp> (define-alien-structure (space (:predicate check-space))
        (area-1 :unsigned-integer 0 4)
        (area-2 :unsigned-integer 4 8))
SPACE
```

When the LISP system evaluates the preceding definition, the DEFINE-ALIEN-STRUCTURE macro generates the predicate function CHECK-SPACE.

If you specify NIL with the :PREDICATE keyword, the DEFINE-ALIEN-STRUCTURE macro does not define a predicate function.

NOTE

Be aware that if you create a field with the name P, then there will be a name conflict between the default predicate function and the default accessor function of the P field. For example, with an alien struture of type SPACE, both the predicate function and the accessor function of the P field would have the same name, SPACE-P.

5.2.5 Specifying a Print Function

You can use the :PRINT-FUNCTION keyword option to specify the function to print an alien structure. The default print function prints only the name of the alien structure. You may want to print the contents of the structure's fields. The following example of an alien structure definition specifies a print function:

```
(define-alien-structure (space (:print-function space-print))
                        (area-1 :unsigned-integer 0 4)
                        (area-2 :unsigned-integer 4 8))
```

If you specify a print function in an alien structure definition, you also must have previously defined that print function. This print function can be defined to have an arbitrary action. However, the print function definition must take at least three arguments:

- A *name* indicating the alien structure to be printed
- A *stream* indicating the stream to which to print
- An *integer* indicating the current print depth

These three arguments are requirements of a structure's user-defined print function, as specified by Common LISP. However, the last argument, indicating the current print depth, is not as useful with alien structures as with other structures. With complex structures, for example, you may not want to print all the information in the structure. However, the fields of an alien structure are immediate objects, so it is usually desirable to print all the fields. Consequently, the *depth* argument is often ignored with alien structures, as in the following example:

```
(defun space-print (alien stream depth)
  (declare (ignore depth))
  (format stream "#<space: area-1 = ~d, area-2 = ~d>~%"
          (space-area-1 alien)
          (space-area-2 alien)))
```

In SPACE-PRINT function, the ALIEN argument refers to the specific alien structure to be printed. The STREAM argument is the stream to which to print. The DEPTH argument is ignored by using the DECLARE special form. If you want to use the DEPTH argument in your print function, see the *PRINT-LEVEL* variable description in *Common LISP: The Language*.

The following example shows how the PRINT-SPACE function prints a SPACE object called EXAMPLE-3:

```
Lisp> (setf example-3 (make-space :area-1 6 :area-2 5))
#<Space: area-1 = 6, area-2 = 5>
Lisp> example-3
#<Space: area-1 = 6, area-2 = 5>
```

For more information on creating print functions for structures and on formatting them, see *Common LISP: The Language*.

5.3 Alien Structure Documentation String

You can include a documentation string for an alien structure. The string is optional and is attached to the symbol as a documentation string of type `STRUCTURE`. Place the string in the definition after the name and options list as in the example in Section 5.1.

5.4 Alien Structure Field Descriptions

Alien structures are composed of data fields, each of which consists of:

- Field name
- Field type
- Start and end positions
- Options

When you define an alien structure, specify a field description as a list of the elements having the following format:

(*field-name type start-position end-position options*)

For example:

```
(field-1 :text 0 9 :occurs 10 :offset 15)
```

This describes a field named `FIELD-1`, a 9-byte text string. The field repeats 10 times, with a 15-byte offset from the beginning of one occurrence to the beginning of the next. This leaves a 6-byte gap between each string in the alien structure.

The following sections describe the elements of a field description.

5.4.1 Field Name

An alien structure's field name is a symbol naming that field. `FIELD-1` is a field name in the previous example. Accessor and constructor functions refer to field names to access and set the values of their respective fields.

5.4.2 Field Type

Alien structure field types specify a relationship between the VAX data in a field and a LISP data type. The LISP system converts alien structure data in both directions:

- When storing the data in a field, the system converts LISP objects into VAX data.
- When accessing the data in a field, the system converts VAX data into LISP objects.

In the previous example, `:TEXT` is a field type.

Table 5–1 lists the field types defined by VAX LISP. See Chapter 4 for more information on these types.

Table 5–1: Alien Structure Field Types

Type	Internal Representation	Notes
`:ASCIW`	VAX character string	The first 16-bit word of the data vector contains a count of the number of characters in the string. Allocate two bytes in addition to the maximum length of the string to hold this count.
`:VARYING-STRING`	Synonym for `:ASCIW`	
`:ASCIZ`	VAX character string terminated with the NULL character (0s in the last byte(s)).	Allocate enough space for the terminating 0. On accessing this slot, the returned LISP string terminates at the first NULL character.
`:TEXT`	VAX nonvarying character string	Allocate one byte for every character in the string.
`:STRING`	Synonym for `:TEXT`	
`:SIGNED-INTEGER`	Signed two's complement integer	
`:UNSIGNED-INTEGER`	Unsigned integer	
`:BIT-VECTOR`	Unsigned integer	
`:F-FLOATING`	F_floating data	
`:G-FLOATING`	G_floating data	
`:D-FLOATING`	D_floating data	When you access a VAX `:D-FLOATING` type, the accessor converts it into a LISP `DOUBLE-FLOAT`, which is equivalent to a VAX `:G-FLOATING` type.
`:H-FLOATING`	H_floating data	
`:POINTER`		(See below)
`:SELECTION`		(See below)

In addition to these types, you can define your own field types with the `DEFINE-ALIEN-FIELD-TYPE` macro. See *VAX LISP/VMS Function, Macro and Variable Descriptions* for a description of this macro.

:POINTER Type

If you want your alien structure to contain the address of the data in another alien structure, specify the `:POINTER` field type in one of the data fields. This field type indicates that the field contains a VAX pointer to the start of the data area of another alien structure.

NOTE

The alien structure pointed to must not be dynamically allocated. Otherwise, after a garbage collection, the pointer will no longer point to the specified data field. For a description of how to allocate alien structures statically, see Section 5.5.3.

The format for using a `:POINTER` field type is:

(:POINTER [*name*] [:DISPLACED *value*])

The optional *name* argument is the type of alien structure pointed to. If you specify this argument, the field's update function checks that the new value of this field (the name you give it when you create an instance of the structure) points to a structure of the specified type.

The optional `:DISPLACED` keyword causes the stored VAX pointer to point to the beginning of the alien structure data area plus the number of bytes specified for the value. You can omit the parentheses if you do not specify the field name and the `:DISPLACED` keyword. The following example shows a data field with the type `:POINTER`:

```
(area-1 (:pointer space) 0 4)
```

or

```
(area-1 :pointer 0 4)
```

:SELECTION Type

The `:SELECTION` field type lets you enumerate all the possible data values of a field. The format for using a `:SELECTION` field type is:

(:SELECTION *s0 s1 s2* . . .)

If you specify the `:SELECTION` type, the `DEFINE-ALIEN-STRUCTURE` macro associates each element in the list (*sn*) with an unsigned integer corresponding to the element's position in the list. For example, consider the following alien structure definition with one `:SELECTION` field:

```
Lisp> (define-alien-structure map
          (state (:selection "massachusetts" "new york"
                             "california" "new hampshire")
                 0 4))
MAP
```

This defines a `MAP` structure whose `MAP-STATE` field can have one of the following values (`"MASSACHUSETTS" "NEW YORK" "CALIFORNIA" "NEW HAMPSHIRE"`). The field is internally stored as an unsigned integer indicating the position of the value in the selection list `("MASSACHUSETTS" "NEW YORK" "CALIFORNIA" "NEW HAMPSHIRE")`.

The `DEFINE-ALIEN-STRUCTURE` macro uses the `EQUALP` function to compare the LISP object you specify when creating an alien structure with the item in the definition's selection list. Next, an instance of a `MAP` structure is created, with its `MAP-STATE` field initialized to `"MASSACHUSETTS"`:

```
Lisp> (setf geo (make-map :state "massachusetts"))
#<Alien Structure MAP #x47D95C>
```

Then, the `ALIEN-FIELD` function can access the field as an unsigned integer:

```
Lisp> (alien-field geo :unsigned-integer 0 4)
0
```

Note that the actual value stored in the field is 0, because `"MASSACHUSETTS"` is the 0th element of the list. Next, the `MAP-STATE` accessor function accesses the field as an unsigned integer and uses that integer as an index into the selection list, returning the corresponding element:

```
Lisp> (map-state geo)
"MASSACHUSETTS"
```

Finally, the SETF form places "CALIFORNIA" in the field and the ALIEN-FIELD function verifies that "CALIFORNIA" is in position 2.

```
Lisp> (setf (map-state geo) "california")
"CALIFORNIA"
Lisp> (alien-field geo :unsigned-integer 0 4)
2
```

5.4.3 Field Positions

You position a field in an alien structure's data area by specifying start and end values in the field specification. These arguments are rational numbers. For example, in the following field description, the 0 and the 4 are the start and end positions of the field:

```
(area-1 :signed-integer 0 4)
```

5.4.3.1 Start and End Positions

The start position is inclusive and the end position is exclusive. For example, if a field's start position is 0 and its end position is 4, the field occupies positions 0, 1, 2, and 3.

Each field is measured in units of 8-bit bytes. The position value, therefore, can be a ratio; that is, you can specify fields within arbitrary bit boundaries. For example, a field with a start value of 1/2 starts on the fifth bit of the data area. Because the units are 8-bit bytes, a start or end value must go evenly into 8. For example, 1/3 would cause an error when you called the DEFINE-ALIEN-STRUCTURE macro.

Exceptions to this rule are all string values or :F-FLOATING, :G-FLOATING, :D-FLOATING, or :H-FLOATING values. These objects must begin and end on byte boundaries; that is, their start and end positions must be fixnums, not ratios.

The LISP system does not evaluate the start and end positions when it expands the DEFINE-ALIEN-STRUCTURE macro. It evaluates these positions when invoking the accessor functions.

5.4.3.2 Gaps Between Field Positions

A gap is memory space that you can allocate as part of an alien structure. For example, if you use the :OFFSET keyword you can produce gaps in an alien structure.

The following example shows a Pascal record structure definition that contains gaps:

```
TYPE

    FAMILY_REC = RECORD
    {A record structure definition.}
        SURNAME         : PACKED ARRAY[1..20] OF CHAR;
        FATHER          : RECORD
                NAME    : PACKED ARRAY[1..20] OF CHAR;
                AGE     : INTEGER;
        END; {of father record}
        MOTHER          : RECORD
                NAME    : PACKED ARRAY[1..20] OF CHAR;
                AGE     : INTEGER;
        END; {of mother record}
        NUM_CHILDREN    : INTEGER;
        CHILDREN        : ARRAY [0..20] of RECORD
                NAME    : PACKED ARRAY[1..20] OF CHAR;
                AGE     : INTEGER;
                SEX     : (FEMALE, MALE);
        END; {of children record}
    END; {of family record}
```

To refer to this structure from LISP, you have to define it as an alien structure, as follows:

```
Lisp> (define-alien-structure family-rec
      "A record structure definition."
        (surname :text 0 20)
        (father-name  :text 20 40)
        (father-age   :unsigned-integer 40 44)
        (mother-name  :text 44 64)
        (mother-age   :unsigned-integer 64 68)
        (num-children :unsigned-integer 68 72 :default 2)
        (child-name   :text 72 92 :occurs 20 :offset 25)
        (child-age    :unsigned-integer 92 96 :occurs 20
                                              :offset 25)
        (child-sex (:selection "female" "male") 96 97
                                                :occurs 20
                                                :offset 25))
FAMILY-REC
```

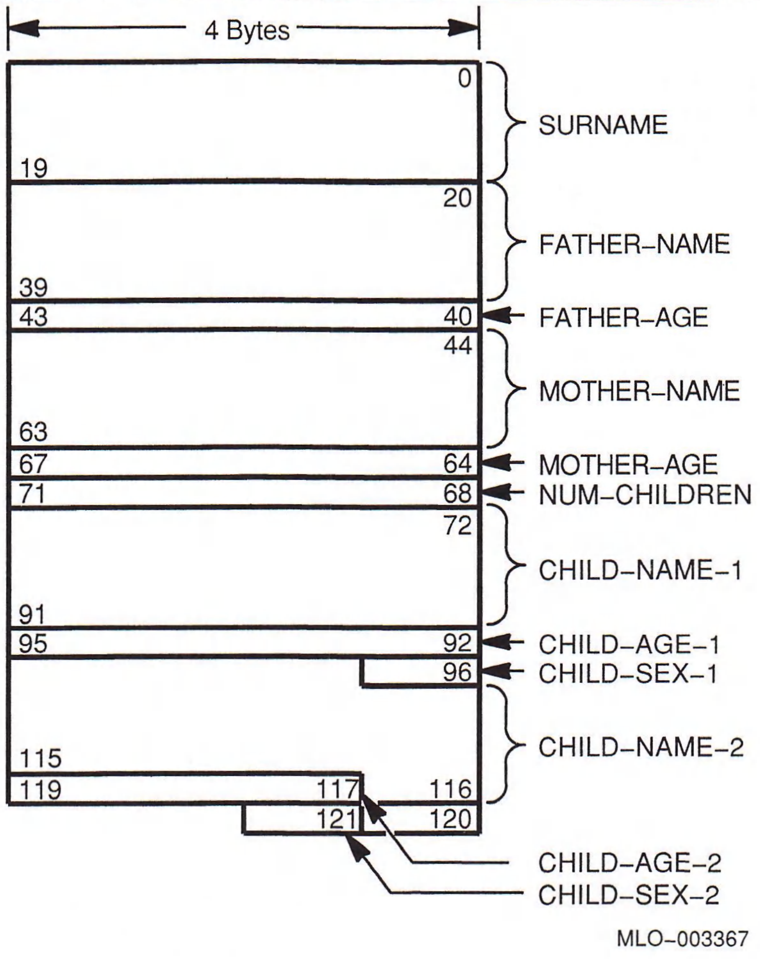
The alien structure named FAMILY-REC has 66 fields in which to store information about the members of a family, including the ages and sex of the children. The CHILD-NAME, CHILD-AGE, and the CHILD-SEX fields occur 20 times, allowing up to 20 children in a family record. The default number of children, however, is two.

The name fields are strings that can be up to 20 characters in length. The age fields are integers that are one longword in length. The sex fields can be either of the two indicated values that are internally represented by an unsigned integer, one byte in length.

The gap comes between each occurrence of the CHILD-NAME field. Note how the field is 20 bytes long but is offset by 25 bytes, leaving 5 bytes between each occurrence of the field. The gap is filled by the CHILD-AGE and CHILD-SEX fields.

Figure 5–1 illustrates how storage is internally allocated for the preceding FAMILY-REC alien structure. Only the first part of the alien structure is shown because the rest of the structure would be repeated in a similar way. The numbers indicate bytes; for example, the surname field occupies bytes 0 through 19. The names identify the fields.

Figure 5–1: Internal Storage of FAMILY-REC



Even though a gap can exist between fields or at the beginning of a field, it may not be accessible. The first field must begin at offset 0 to be processed by LISP-level code. If the first field does not begin at offset 0, only the `ALIEN-FIELD` function (see *VAX LISP Function, Macro and Variable Descriptions*) can access the gap.

5.4.3.3 Overlapping Fields

Alien structure fields can overlap, letting you access data from more than one field at a time or from one field in a number of ways. If you change the data in a field that overlaps other fields, these overlapped fields are also changed.

Overlapping fields are useful when you want data to be interpreted in more than one way. The following definition defines an alien structure that contains fields that overlap. The individual `BIT` fields overlap the `NUMBER` field, though they do not overlap one another:

```
Lisp> (define-alien-structure mask
          (number :unsigned-integer 0 4)
          (bit-0  :unsigned-integer 0 1/8)
          (bit-1  :unsigned-integer 1/8 2/8)
          (bit-2  :unsigned-integer 2/8 3/8)
          (bit-3  :unsigned-integer 3/8 4/8)
          (bit-4  :unsigned-integer 4/8 5/8))
MASK
```

If you specify different values for overlapping fields when you initialize them (see Section 5.4.4.1 on initializing fields), the field values that result are undefined. For example, consider an alien structure of the previously defined MASK type where the number field overlaps the bit fields. If you create an instance of MASK with the MAKE-MASK function, and you initialize the number and bit fields to conflicting values (for example, `(make-mask :number 0 :bit-2 1)`), the result is undefined.

The next example shows the creation of the alien structure NEWMASK of the previously defined type MASK:

```
Lisp> (setf newmask (make-mask))
#<Alien Structure MASK #x50C600>
```

The following are two ways to set bits 2 and 4 in NEWMASK and to clear all other bits:

```
Lisp> (setf (mask-number newmask) (+ 4 16))
20

Lisp> (setf (mask-number newmask) 0
            (mask-bit-2 newmask) 1
            (mask-bit-4 newmask) 1)
1
```

5.4.4 Field Options

By specifying options in the alien structure's data-field descriptions, you can define the characteristics of the fields. You specify a data-field option as a keyword–value pair. Include each option in a list whose first element is the name of the field. You can specify more than one option at a time in the list. The format for an options list is:

(*field-name keyword value* . . .)

The next sections explain each keyword in detail.

5.4.4.1 Initial Value

To specify an initial value for a field, use the :DEFAULT keyword in the alien structure's definition. Then, when you create an instance of a structure with initialized fields, you do not have to specify values for those fields. Instead, the LISP system automatically puts your initial values in the fields you create. For example, in the following data field specification of an alien structure definition, the value of the NUM-CHILDREN field is initialized to 2.

```
(num-children :unsigned-integer 68 72 :default 2)
```

You can override the default field value for an alien structure's field on creating the structure. To do so, place new values in the initialized fields when you create a specific instance of a defined structure. For example, in the following creation of an alien structure of type FAMILY-REC, the :NUM-CHILDREN field is initialized to 3.

```
(setf example-4 (make-family-rec :num-children 3))
```

The default field value can also be changed after creation of an alien structure by using the SETF macro with the accessor function of that field.

NOTE

By default, the uninitialized initial contents of a field are unpredictable.

5.4.4.2 Read-Only Value

The :READ-ONLY keyword lets you specify whether a field can be accessed or set. The value you specify with this keyword can be either T or NIL. NIL is the default.

If you specify T, the DEFINE-ALIEN-STRUCTURE macro generates accessor functions that are not acceptable in a call to the SETF macro. That is, after you create an individual structure containing the field, you can only access the field to read data from it. You cannot use the SETF macro on the accessor function for that field to write data to it. If you specify NIL (the default), the DEFINE-ALIEN-STRUCTURE macro generates accessor functions that are acceptable in a call to SETF.

For example, in the following definition, the default value of the AREA-2 field is 4. This value can be accessed but not changed after you create an individual structure from this definition. However, the value of the AREA-1 field, which defaults to 2, can be changed after you create an individual structure:

```
(define-alien-structure (space (:print-function #'space-print))
                        (area-1 :unsigned-integer 0 4 :default 2)
                        (area-2 :unsigned-integer 4 8 :default 4
                                :read-only t))
```

5.4.4.3 Repeated Field

A field can be repeated within an alien structure. By specifying a positive integer with the :OCCURS keyword, you determine the number of times the field is repeated. For example, the following line indicates that the NAME field occurs 20 times with its first occurrence between bytes 20 and 30.

```
(name :text 20 30 :occurs 20)
```

If you do not specify the :OCCURS keyword, the accessor function takes the field name as its argument, and the field occurs once. If you specify this keyword, the accessor function takes the field name and an index for arguments. The index is an integer that indicates the occurrence of the field. The first occurrence of the field has an index of 0. Consider the following definition:

```
Lisp> (define-alien-structure space
        (area-1 :unsigned-integer 0 4)
        (area-2 :unsigned-integer 4 8 :occurs 4))
SPACE
```

When LISP evaluates this definition, the accessor functions AREA-1 and AREA-2 have the following formats:

(SPACE-AREA-1 *field*)
(SPACE-AREA-2 *field index*)

5.4.4.4 Similar-Field Distances

You can specify how far apart similar fields are by using the :OFFSET keyword. This option makes sense only if used with the :OCCURS keyword.

A field offset is the distance in 8-bit bytes from the start of one occurrence of a field to the start of the next occurrence of that field. Specifying an offset lets you access data files that consist of repeated substructures.

You define an offset value by specifying a rational number with the :OFFSET keyword. For example, the following line indicates that 25 8-bit bytes come between each occurrence of the CHILD-NAME field:

```
(child-name :text 72 92 :occurs 20 :offset 25)
```

If you specify a value that is greater than the field length (as in the previous example), the DEFINE-ALIEN-STRUCTURE macro produces gaps in the alien structure. You can fill them by defining one or more other fields with the :OCCURS and :OFFSET keywords; that is, you can interleave different fields.

The LISP system does not evaluate the value you specify with the :OFFSET keyword when it expands the DEFINE-ALIEN-STRUCTURE macro. It does the evaluation when you invoke the structure's accessor functions. The offset defaults to the length of the field.

5.5 Creating an Alien Structure

After you have defined an alien structure data type, you can create an instance of that data type. To do so, specify a call to the constructor function of the data type you want. For example, in the following expression, the SETF macro gives the symbol EXAMPLE-1 the value of the alien structure SPACE:

```
(setf example-1 (make-space))
```

Constructor functions accept optional keywords that initialize data fields and affect memory allocation of alien structures.

5.5.1 Initializing and Changing Data Fields

The constructor function for an alien structure accepts keyword arguments to initialize data fields. Each keyword is the name of a data field prefixed by a colon. For example, when the LISP system evaluates the following definition, the MAKE-SPACE constructor function accepts two data-initialization keywords, :AREA-1 and :AREA-2.

```
Lisp> (define-alien-structure space
        (area-1 :unsigned-integer 0 4)
        (area-2 :unsigned-integer 4 8))
SPACE
```

When you create an individual alien structure, you can assign values to the structure's fields with the initialization keywords. For example:

```
Lisp> (setf example-1 (make-space :area-1 5 :area-2 10))
#<Alien Structure SPACE #x403B80>
```

You can also initialize the fields by specifying the :DEFAULT keyword (see Section 5.4.4.1) with a value when you define the structure. For example, the following AREA fields have default initial values of 6 and 12:

```
Lisp> (define-alien-structure space
        (area-1 :unsigned-integer 0 4 :default 6)
        (area-2 :unsigned-integer 4 8 :default 12))
SPACE
```

Initializing data with the constructor function overrides a default in the same field in the alien structure definition.

If you want to change a field value after you have created it, you can change it with the SETF macro if the field definition allows the change. (See Section 5.4.4.2). For example, the field AREA-1 is set to 28 in the following SETF form:

```
Lisp> (setf (space-area-1 example-1) 28)
28
```

5.5.2 Setting Allocation Size

You can override the amount of storage allocated by the constructor function using the :ALIEN-DATA-LENGTH keyword. This keyword lets you set the number of bytes of memory to be allocated for the alien structure's data vector. It has the following syntax:

:ALIEN-DATA-LENGTH *integer*

This keyword uses storage efficiently when you use alien structures as data buffers for variable size records. The default is a number large enough to store the defined alien structure. A length larger than the default allows a larger than normal alien structure to be allocated; the “extra” data can be accessed with the ALIEN-FIELD function. If an alien structure is constructed with a smaller size than the default, it is an error to access or set the omitted fields.

See the ALIEN-STRUCTURE-LENGTH function description in *VAX LISP Function, Macro and Variable Descriptions* for an example of default byte allocations.

5.5.3 Allocating Static or Dynamic Space

The :ALLOCATION keyword lets you set the type of allocation to be used for the alien structure, as follows:

:ALLOCATION *value*

Valid values are :DYNAMIC and :STATIC. :DYNAMIC is the default.

If :STATIC is specified, the alien structure is allocated in static space and its virtual address is not changed during garbage collection (see the *VAX LISP/VMS Program Development Guide*).

5.5.4 Setting the Pointer to the Data Vector

The `:DATA` keyword lets you override the data vector used by the alien structure. With this keyword, you can make the pointer to the data vector point to any object in either LISP or non-LISP space. Use this keyword when you want the data of the alien structure to be something other than the default. For example, you can set `:DATA` to a LISP simple string or other array of 8-bit bytes or to non-LISP memory. Using `:DATA` to point to other than the default data vector allows you to modify an object in place or use an existing object as the data of an alien structure.

The `:DATA` keyword has the following syntax:

:DATA *value*

Its *value* can be the address of either a simple string or an array of 8-bit bytes.

After you create an instance of an alien structure with this keyword, you can dereference the pointer by calling the `ALIEN-DATA` function. This function returns the data contained at the specified address.

Chapter 4 contains an example of a callback routine that accepts a VAX argument list as its only argument. The argument list is in non-LISP space. It is represented as an alien structure. The callback routine uses the `:DATA` keyword to make the alien structure point to the actual argument list rather than to a data vector in LISP space.

NOTE

Because `:DATA` is a keyword to the alien structure constructor function, do not use `DATA` for the name of any field in the alien structure. Otherwise, unpredictable results will occur.

Chapter 6

Interrupt Functions

VAX LISP provides support for handling asynchronous events. An asynchronous event is one that interrupts the normal flow of program execution, such as the completion of I/O, expiration of a timer, or movement of a workstation pointing device. This mechanism allows you to specify a function, called an interrupt function, to be executed when a particular asynchronous event occurs.

This chapter provides a guide to using interrupt functions. The chapter is organized as follows:

- Section 6.1 provides an overview of the use of interrupt functions.
- Section 6.2 defines an asynchronous event and shows how the VMS operating system, through the AST mechanism, lets you request notification of asynchronous events.
- Section 6.3 shows how to establish an interrupt function in LISP and specify that it be executed as the result of a particular asynchronous event.

Chapter 7 contains information on topics related to interrupt functions, including establishing priority levels at which interrupt functions operate, protecting sections of code against interruption, and synchronizing the execution of interrupt functions with the `WAIT` function.

6.1 Overview of Interrupt Functions

Interrupt functions allow your LISP program to respond to events that occur independently of normal program execution. To use an interrupt function, follow these steps:

1. Decide what asynchronous event should trigger the function, what the function should do, and what information the function requires. Section 6.2 provides information about various sources of asynchronous events.
2. Define the function as you would any LISP function. Try to localize data manipulation. A LISP closure may be useful for this. You can synchronize access to global state information with the facilities described below.
3. Use the `INSTATE-INTERRUPT-FUNCTION` function to make your function known to LISP as an interrupt function. `INSTATE-INTERRUPT-FUNCTION` returns an identification number, the *iif-id*, that you must retain for future use. Section 6.3 shows how to use `INSTATE-INTERRUPT-FUNCTION`.

4. Define and call the routine that causes an AST, using one of the following methods:

    - Use the callout facility to define and call a system routine or other external routine that causes an AST. Supply the *iif-id* of your interrupt function as the *astprm* argument to the system routine.
    - Call one of the VAX LISP functions that establishes a response to an asynchronous event. Give the *iif-id* as the *action* argument.

    Note that an external routine may post and even execute an AST while the external code is executing. However, the associated LISP interrupt function will not execute until the external code returns. In general, execution of LISP interrupt functions is only allowed while other LISP code is, or could be, executing; not during execution of foreign code.

    Section 6.3 provides examples of using both system routines and VAX LISP functions.

5. Your interrupt function executes every time the asynchronous event occurs.
6. To synchronize your program with the execution of an interrupt function, use the WAIT function. Chapter 7 describes this function.
7. When your interrupt function is no longer needed—that is, after the asynchronous event has occurred for the last time—use the UNINSTATE-INTERRUPT-FUNCTION function to remove the interrupt function from LISP's table of interrupt functions.

6.2 Asynchronous Events in VMS

An asynchronous event occurs at an unpredictable point during the execution of a program. (By contrast, a synchronous event happens at the same point in the program every time.) Some examples of asynchronous events are:

- A program queues a request for input, then continues execution. At some later point, unpredictable in advance, the input is completed. The input completion is an asynchronous event.
- A program sets a timer to go off in five seconds, then continues execution. When the interval is up, an asynchronous event occurs. This event is asynchronous because it is impossible to predict what code will be executing.
- The user of an application moves a workstation pointer and clicks one of the pointer buttons. The pointer movement and the button click are asynchronous events.

The rest of this section describes how the VMS operating system provides access to asynchronous events. For more information, see the *VMS System Services Reference Manual*.

6.2.1 Asynchronous System Traps (ASTs)

The VMS operating system provides an Asynchronous System Trap (AST) mechanism that lets you request notification of an asynchronous event. The AST mechanism lets you specify a routine to be executed when a specified asynchronous event occurs; such routines are called AST service routines.

Some system routines, by their nature, can cause only one AST for each call. For example, a routine that sets a timer can cause only one AST because the timer can expire only once. Other system routines can cause an unlimited number of ASTs from a single call. For example, a system routine that establishes an AST service routine for pointer button activation causes an AST each time a pointer button is pressed or released. The pointer button ASTs continue until they are halted by another call to the system routine.

6.2.2 Routines That Cause ASTs

All ASTs are caused by system routines, whether you call them directly through the callout facility or indirectly through functions supplied by VAX LISP.

6.2.2.1 System Routines

There are two ways to determine whether a VMS system routine causes an AST. First, the documentation notes that the routine completes asynchronously. Second, the routine has the following two arguments in its argument list:

- *astadr*—the address of the AST service routine. You supply this address when you call the routine that causes the AST.
- *astprm* (or some equivalent)—the AST parameter. This is an arbitrary value that you supply. VMS passes the AST parameter to the AST service routine. When multiple asynchronous events share a single AST service routine, the AST service routine can use the AST parameter to determine which asynchronous event caused it to be called.

To use system routines that declare ASTs, you must use the callout facility, just as with any other external routine. Section 6.3 describes how to call these system routines.

6.2.2.2 VAX LISP Routines

A number of VAX LISP functions establish actions to be taken when specific asynchronous events occur. Currently, most of these functions are part of VAX LISP's support of the VAXstation. They set up a response to pointer movement, pointer button activation, and viewport manipulation. These functions ultimately call system routines. Using these functions saves you the trouble of defining the system routine and using CALL-OUT. In other respects, the use of these functions is similar to the use of system routines.

6.2.2.3 Keyboard Functions

You can use the BIND-KEYBOARD-FUNCTION function to bind a control character on the keyboard to the execution of a LISP function. (See the *VAX LISP/VMS Function, Macro and Variable Descriptions* manual for a description of BIND-KEYBOARD-FUNCTION.) VAX LISP invokes all keyboard functions through a single interrupt function. The function that you specify with BIND-KEYBOARD-FUNCTION can interrupt the execution of LISP code. However, you do not have to instate or uninstate keyboard functions.

6.3 Establishing LISP Interrupt Functions

This section details the steps you take to use an interrupt function. The section is divided as follows:

- Any function that LISP is to execute asynchronously must be made known to LISP as an interrupt function with the INSTATE-INTERRUPT-FUNCTION function. Section 6.3.1 shows how to use INSTATE-INTERRUPT-FUNCTION.
- After a function has been made known as an interrupt function, you must associate the interrupt function with one or more asynchronous events. Section 6.3.2 describes two ways of doing so.
- When an interrupt function is no longer needed, remove it from LISP's table of interrupt functions to conserve system resources. See Section 6.3.3.
- If you suspend a LISP system, the interrupt functions you have instated become uninstated in the suspended system and are not automatically reinstated when the system is resumed. You must reinstate those functions yourself. (See Section 6.3.4.)

6.3.1 Defining an Interrupt Function

To define an interrupt function, you first define the function, which may take arguments. You then use INSTATE-INTERRUPT-FUNCTION to make your function known to LISP. The INSTATE-INTERRUPT-FUNCTION function takes a function as an argument and returns an identifying number, the *iif-id*. LISP adds your function to an internal list that identifies interrupt functions. The *iif-id* allows LISP to retrieve a particular interrupt function at a future time.

For example:

```
(let ((iif-id (instate-interrupt-function #'key-handler)))
     . . .   )
```

This example makes the function KEY-HANDLER known as an interrupt function. The value returned by INSTATE-INTERRUPT-FUNCTION is bound to the symbol IIF-ID, to be used later in the body of the LET. Section 6.3.2 shows how the *iif-id* associates an interrupt function with an asynchronous event.

You can use INSTATE-INTERRUPT-FUNCTION on a single function as many times as you like, thereby creating more than one interrupt function with the same function definition. This technique is useful when you need an interrupt function to perform essentially the same operation in response to slightly differing asynchronous events.

6.3.1.1 Passing Arguments to Interrupt Functions

Ordinarily, an interrupt function receives no arguments when it is invoked. You can, however, specify that one or more arguments be passed to an interrupt function. This technique is useful in the following cases:

- You use INSTATE-INTERRUPT-FUNCTION more than once on a single function, thereby making several interrupt functions with the same function definition. You can cause each interrupt function to be passed a different argument, allowing the function to take appropriate action depending on what asynchronous event invoked it.

- Your interrupt function needs to manipulate a data structure. The only safe way for an interrupt function to manipulate a data structure is to pass the data structure to the interrupt function as an argument. You should not store the data structure in a special variable, because the binding of special variables cannot be certain at the time the interrupt function executes.

The INSTATE-INTERRUPT-FUNCTION function takes a keyword argument, :ARGUMENTS, whose value is a list of the arguments to be passed to the interrupt function. For example:

```
(let ((iif-id (instate-interrupt-function #'key-handler
                 :arguments (list interrupting-key))))
  . . . )
```

This example makes the function KEY-HANDLER known as an interrupt function and requests that the data structure INTERRUPTING-KEY be passed to KEY-HANDLER when it is invoked. KEY-HANDLER can manipulate the contents of INTERRUPTING-KEY. Following execution of KEY-HANDLER, other LISP functions can use the modified contents of INTERRUPTING-KEY.

6.3.1.2 Specifying the Interrupt Level

Each interrupt function has an interrupt level. An interrupt function can only interrupt code that is executing at a lower interrupt level than its own. Chapter 7 contains more information about interrupt levels.

Use the :LEVEL keyword with INSTATE-INTERRUPT-FUNCTION to specify the interrupt level. The value for this keyword is an integer in the range 0 through 7. The default is 2.

6.3.1.3 Automatic Removal of Interrupt Functions

The INSTATE-INTERRUPT-FUNCTION function takes a keyword argument, :ONCE-ONLY-P, that lets you request that the interrupt function be uninstated after a single execution. If you include :ONCE-ONLY-P with a non-NIL argument, the interrupt function can execute only once; then it is automatically removed from LISP's table of interrupt functions. Use this keyword only if you know that the interrupt function will be needed only once. For interrupt functions that may execute more than once, remove them explicitly with the UNINSTATE-INTERRUPT-FUNCTION function after the last use. (Section 6.3.3 describes UNINSTATE-INTERRUPT-FUNCTION.)

6.3.2 Associating an Interrupt Function with an Asynchronous Event

To request invocation of an interrupt function, you must associate it with one or more asynchronous events. This section shows how to call out to system routines that cause ASTs and how to pass interrupt functions as arguments to VAX LISP functions that establish the response to an asynchronous event.

6.3.2.1 Calling Out to System Routines That Cause Asynchronous Events

System routines that can cause asynchronous events are characterized by the presence of two arguments, the *astadr* (AST service routine address) and the *astprm* (AST parameter). To call out to such a routine, you first define the routine, using DEFINE-EXTERNAL-ROUTINE.

- Always use :MECHANISM :VALUE to pass the *astadr* argument.

- Use either :MECHANISM :VALUE or :MECHANISM :REFERENCE to pass the *astprm* argument. Consult the documentation of the system routine to determine how the routine expects the *astprm* argument to be passed.

For example:

```
(define-external-routine (sys$setimr :check-status-return t)
  (efn :mechanism :value)
  (daytim :vax-type :quadword)
  (astadr :mechanism :value)
  (astprm :mechanism :value))   ; Called the REQIDT in VMS docs.
```

This example defines the external routine SYS$SETIMR, a system service that sets a timer and causes an asynchronous event to occur when the timer expires.

When you use CALL-OUT to call the system routine, you must pass appropriate values for both the *astadr* and the *astprm*:

- For the *astadr*, always pass the parameter COMMON-AST-ADDRESS. This is the address of a VAX LISP routine that initially handles all asynchronous events. You can pass no other object as the *astadr*.
- For the *astprm*, pass the *iif-id* of the interrupt function that is to service the asynchronous event.

For example:

```
(defun set-timer (delta-time)
  (let ((iif-id (instate-interrupt-function
                  #'timer-interrupt-handler
                  :once-only-p t)))
    (call-out sys$setimr nil delta-time
                         common-ast-address iif-id))
  t)
```

This example defines the function SET-TIMER, which in turn calls out to SYS$SETIMR. Each invocation of SET-TIMER causes the function TIMER-INTERRUPT-HANDLER to be instated as an interrupt function. The :ONCE-ONLY-P keyword causes TIMER-INTERRUPT-HANDLER to be removed (uninstated) after its first invocation.

6.3.2.2 Using VAX LISP Functions

Several VAX LISP functions establish an action that is to take place when a specified asynchronous event occurs. (Most of these functions support the use of the pointer on a VAXstation running UIS.) One of the actions you can request with these functions is the execution of an interrupt function. After you have defined and instated an interrupt function as described in Section 6.3.1, you can supply its *iif-id* as the *action* argument.

For example, the SET-POINTER-ACTION function establishes the response to pointer movement in a workstation viewport. It takes four required arguments: a virtual display, a window, and actions to perform when the pointer moves within the window and when it exits the window. The actions can be either NIL (do nothing) or an interrupt function to execute whenever the pointer moves within or out of the window. The following example shows the use of an interrupt function with SET-POINTER-ACTION:

```
(let ((iif-id (instate-interrupt-function #'draw-rubber-band)))
  (set-pointer-action *art-display* *art-window* iif-id nil)
     .
     .
  (set-pointer-action *art-display* *art-window* nil nil)
  (uninstate-interrupt-function iif-id))
```

In this example, a previously defined function, DRAW-RUBBER-BAND, is instated as an interrupt function. Its *iif-id* is then supplied as the first *action* argument to SET-POINTER-ACTION. From that point on, any pointer movement in the window *ART-WINDOW* causes the interrupt function DRAW-RUBBER-BAND to execute. The second call to SET-POINTER-ACTION requests that pointer movement not cause an asynchronous event, and the UNINSTATE-INTERRUPT-FUNCTION function removes the unneeded interrupt function from LISP's table of interrupt functions.

Some LISP functions that specify a response to asynchronous events cause arguments to be passed to the interrupt function that you specify. For example, the SET-BUTTON-ACTION function specifies the response when a workstation pointer button is pressed or released. If you specify an interrupt function with SET-BUTTON-ACTION, the interrupt function is automatically passed two arguments when it is invoked: the button involved and the direction of the transition (down or up). You can still use the :ARGUMENTS keyword with INSTATE-INTERRUPT-FUNCTION to specify arguments to be passed to the interrupt function. Arguments that you request are passed following any arguments passed automatically.

For example, assume that you want to pass a virtual display to your button-handling interrupt function, in addition to the two arguments that it receives automatically. You might define the interrupt function as follows:

```
(defun button-handler (button transition display)
      .
      . )
```

But when you use INSTATE-INTERRUPT-FUNCTION, you specify that only one argument be passed:

```
(let ((iif-id (instate-interrupt-function
                #'button-handler
                :arguments (list display))))

       . . .  )
```

When BUTTON-HANDLER executes as the result of a button being pressed or released, it will receive three arguments: the button and transition supplied by VAX LISP and the display that you supply.

6.3.3 Removing an Interrupt Function from LISP

When you use INSTATE-INTERRUPT-FUNCTION, VAX LISP adds the interrupt function to an internal table listing all such functions. Thus, to avoid unnecessary overhead, be sure to uninstate interrupt functions after their last use by using UNINSTATE-INTERRUPT-FUNCTION. UNINSTATE-INTERRUPT-FUNCTION takes a single argument, the *iif-id* of the interrupt function being uninstated.

You should uninstate an interrupt function only after you are sure that the asynchronous event causing it can no longer occur. Some asynchronous events occur only once for every use of the routine or function that causes them. Other asynchronous events, such as those caused by SET-BUTTON-ACTION, can occur repeatedly. In either case, it is your responsibility to be sure that the asynchronous event can no longer occur before uninstating the interrupt function the asynchronous event invokes. If an asynchronous event occurs and its associated interrupt function has been uninstated, LISP ignores the asynchronous event.

You can use the :ONCE-ONLY-P keyword with INSTATE-INTERRUPT-FUNCTION to cause an interrupt function to be uninstated automatically after one invocation. See Section 6.3.1.3.

6.3.4 Suspending Systems Containing Interrupt Functions

When you suspend a LISP system, the interrupt functions you have instated are uninstated in the suspended system. When you resume that system, these interrupt functions are not automatically reinstated. Therefore, if your system can be suspended, you must know what functions to reinstate when the system resumes. When you have reinstated an interrupt function, you must reassociate it with the asynchronous event that invokes it.

If you bind a control character to a function with the `BIND-KEYBOARD-FUNCTION` function, you do not need to rebind the function in a resumed system. VAX LISP automatically restores the bindings when the system is resumed.

Chapter 7

Interrupt Levels, Critical Sections, and Synchronization

This chapter discusses VAX LISP facilities that let you control the priorities of interrupt functions and keyboard functions. You use these facilities to control a system in which multiple interrupt functions and keyboard functions may interfere with each other or with code that must execute as a unit.

This chapter discusses the following subjects:

- Section 7.1 describes the system of interrupt levels. You can specify an interrupt level with the `BIND-KEYBOARD-FUNCTION` and `INSTATE-INTERRUPT-FUNCTION` functions.
- Section 7.2 describes critical sections, which prevent a section of code from being interrupted during execution.
- Section 7.3 describes the `WAIT` function, which suspends execution of a program until a keyboard function or interrupt function executes.

7.1 Using Interrupt Levels

You can use the `:LEVEL` keyword to assign an interrupt level either to an interrupt function or to a function you specify with `BIND-KEYBOARD-FUNCTION`. The interrupt level, which is an integer between `0` and `7`, controls when a function can execute. A function executes only if its interrupt level is greater than LISP's current interrupt level. For example, if you define two keyboard functions with `BIND-KEYBOARD-FUNCTION`, one at level 2 and one at level 3, the second function can interrupt the first but not the other way around.

When it is not executing a keyboard function or an interrupt function, VAX LISP can be interrupted by functions at any of the interrupt levels. Certain low-level LISP functions run at very high interrupt levels because they cannot be safely interrupted. Normally, however, a function at any interrupt level will interrupt LISP execution.

VAX LISP keyboard input operates at interrupt level 6, meaning that any function with an interrupt level less than 6 can perform input from the keyboard. Functions that operate at level 6 or 7 cannot obtain keyboard input.

When you use `BIND-KEYBOARD-FUNCTION` or `INSTATE-INTERRUPT-FUNCTION`, carefully consider which interrupt level to use. You must ensure that the function is able to interrupt other functions that it needs to interrupt and that the function can in turn be interrupted as necessary. Furthermore, if the function performs input from the keyboard, its level must be less than 6. Some guidelines are:

- In general, do not use interrupt levels 6 or 7. Use of these interrupt levels may interfere with VAX LISP's normal operation.

- If you bind a control character (such as Ctrl/E) to the ED function, use either level 1 (the default for BIND-KEYBOARD-FUNCTION) or 0. The Editor must be interruptible by keyboard input and by interrupt functions the Editor uses to handle pointer input.
- If you bind control characters to the DEBUG and BREAK functions, use an interrupt level high enough to interrupt functions you debug, but less than 6. For example, if your application includes an interrupt function that executes at level 3, specify level 4 or 5 with BIND-KEYBOARD-FUNCTION to invoke the debugger or break loop using the control character while that interrupt function is executing.
- In this framework, choose interrupt levels for your interrupt and keyboard functions that allow them to interrupt and to be interrupted as appropriate.

Functions that execute at interrupt level 7 can interrupt any LISP code not in a critical section, including low-level LISP code not normally interruptible. Functions that execute at level 7 may leave your program in an inconsistent state. Therefore, functions that execute at level 7 *must* terminate by executing ABORT. Do not use interrupt level 7 except to effect an emergency exit back to LISP's top level. (Ctrl/C is bound to a function that executes at level 7; therefore, you can always use Ctrl/C to get back to top level.)

7.2 Executing Critical Sections

A critical section consists of forms in the body of a CRITICAL-SECTION macro. The execution of forms in a critical section cannot be interrupted by any interrupt function or keyboard function, at any level. Use a critical section when the execution of code must not be interrupted. For example, a function that manipulates a data structure may temporarily leave the data structure in an inconsistent state during its execution. An interrupting function that tries to use the data structure can find it invalid. The manipulating function can use a critical section to make sure that it cannot be interrupted while the data structure is invalid.

Interrupts that occur during the execution of a critical section are queued. When the critical section ends, the interrupts are serviced.

Since a critical section cannot be interrupted, it cannot perform keyboard input. A critical section also cannot be stopped with Ctrl/C. So, be careful to avoid infinite loops in a critical section. Should an infinite loop occur, you have to terminate the LISP image.

Test your code thoroughly before you make it into a critical section. Critical sections should be short and error free. If an error does occur in a critical section, VAX LISP invokes the debugger and temporarily removes the restrictions on interrupts so that you can type to the debugger. If you continue from the debugger, LISP restores the restrictions on interrupts before continuing. However, LISP is open to interruptions while you are debugging the code.

7.3 Synchronizing Program Execution

Sometimes a program must stop execution until an event occurs or some piece of information becomes available. VAX LISP provides the WAIT function to allow such synchronization.

The WAIT function takes two required arguments: a reason for the wait, typically a string, and a testing function that LISP calls to determine if the wait condition has been satisfied. The WAIT function accepts any number of arguments following the second argument. These arguments are used as arguments to the testing function.

When the WAIT function is called, it causes normal program execution to halt. VAX LISP then repeatedly calls the testing function. When the testing function returns a non-NIL value, the WAIT function returns and execution continues.

You can specify any function as a testing function in a call to the WAIT function. However, remember the following points:

- The testing function should be short and error free. VAX LISP calls the testing function once before establishing the WAIT state. An error that occurs on this initial call can be debugged normally. However, if an error occurs in the testing function after the WAIT state has been established, the LISP system will be left in an inconsistent state and will have to be terminated.
- The testing function should not have side effects, since it is called at indeterminate intervals.
- The dynamic state of LISP is not guaranteed during execution of the testing function. Therefore, the testing function cannot rely on the values of special variables. You should pass it arguments instead.

One way to use WAIT is with an interrupt function or keyboard function that modifies a data structure accessed by the testing function. The data structure can be a cons cell, a structure, or an array. For the testing function, use an accessor function appropriate for that data structure. When the interrupt or keyboard function modifies the data structure, the testing function returns non-NIL, and execution continues.

For example, the following forms set up a variable called FLAG, which is then used in a WAIT function:

```
(setf flag (list nil))
(bind-keyboard-function
  #\^f
  #'(lambda () (setf (car flag) t)))
(wait "Wait for CTRL/F" #'car flag)
```

In this example, the value of FLAG is a list whose only element is NIL. BIND-KEYBOARD-FUNCTION binds Ctrl/F to a function that changes the element of FLAG to T. The WAIT function specifies CAR as its testing function, with FLAG given as the argument. As long as the testing function returns NIL, the WAIT function blocks further execution. When the user presses Ctrl/F, the first element of FLAG is set to T, the testing function returns T, the WAIT function returns, and normal execution continues.

To use the WAIT function to synchronize your program with an interrupt function, pass a data structure to both the interrupt function and the testing function named in the WAIT function. For example, consider the following interrupt function that handles the expiration of a timer:

```
(defun timer-interrupt-handler (flag)
  (setf (car flag) t))
```

This function could be used as follows:

```
(let* ((flag (list nil))
       (iif-id (instate-interrupt-function
                 #'timer-interrupt-handler
                 :once-only-p t
                 :arguments (list flag))))
  (call-out sys$setimr nil delta-time
                   common-ast-address iif-id)
    . . .
  (wait "Timer wait" #'car flag)
    . . . )
```

In this example, the program calls out to SYS$SETIMR, specifying that TIMER-INTERRUPT-HANDLER is to execute when the timer expires. TIMER-INTERRUPT-HANDLER is passed FLAG as an argument, a list whose only element is NIL. TIMER-INTERRUPT-HANDLER sets this element to T when the timer expires.

Meanwhile, after calling SYS$SETIMR, the program continues with code that can execute before the timer has expired. At some point, however, it calls WAIT to wait for the timer. Since both TIMER-INTERRUPT-HANDLER and the testing function CAR have been passed the same list, the WAIT will not return until TIMER-INTERRUPT-HANDLER sets the first element of the list to T.

Sometimes it is useful to pass a structure to an interrupt function. Then, you can include a slot in the structure for synchronization. Consider the following example:

```
(defstruct menu
    . . .
  (choice-made nil))

(defun click-in-menu (button transition menu)
    . . .
  (setf (menu-choice-made menu) t))

(defun post-menu (menu)
    . . .
  (let ((iif-id (instate-interrupt-function
                  #'click-in-menu
                  :arguments (list menu))))
    . . .
  (wait "Menu choice" #'menu-choice-made menu)
    . . .  ))
```

This example shows parts of a menu system. The menu is implemented as a structure, one of whose slots is called CHOICE-MADE. The initial value of CHOICE-MADE is NIL. The interrupt function CLICK-IN-MENU, which executes when a pointer button is pressed over a menu choice, is passed the menu structure as an argument. CLICK-IN-MENU sets the value of CHOICE-MADE to T. The function POST-MENU takes a menu structure as its argument, displays the menu, then waits for a choice to be made. POST-MENU uses the WAIT function and supplies MENU-CHOICE-MADE as the testing function. When CLICK-IN-MENU sets this slot to T, the WAIT function returns and execution continues.

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua NH 03061
Rest of U.S.A. and Puerto Rico[1]	800–DIGITAL	

[1]Prepaid orders from Puerto Rico, call Digital's local subsidiary (809–754–7575)

From	Call	Write
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order Desk
Internal orders (for software documentation)	—	Software Supply Business (SSB) Digital Equipment Corporation Westminster MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Services (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro MA 01532

Reader's Comments

VAX LISP/VMS System Access Guide
AA–GH75B–TE

Your comments and suggestions will help us improve the quality of our future documentation. Please note that this form is for comments on documentation only.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (product works as described)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What I like best about this manual: ______________________________

What I like least about this manual: ______________________________

I found the following errors in this manual:

Page	Description

My additional comments or suggestions for improving this manual:

____

Please indicate the type of user/reader that you most nearly represent:

☐ Administrative Support	☐ Scientist/Engineer
☐ Computer Operator	☐ Software Support
☐ Educator/Trainer	☐ System Manager
☐ Programmer/Analyst	☐ Other (please specify) ____________
☐ Sales	

Name/Title ______________________ Dept. ____________

Company ______________________ Date ____________

Mailing Address ______________________

______________________ Phone ____________

10/87

— — Do Not Tear — Fold Here and Tape — — — — — — — — — — — — — — — — — — —

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
PKO3-1/30D
129 PARKER STREET
MAYNARD, MA 01754-2198

— — Do Not Tear — Fold Here —

Cut Along Dotted Line