VAX LISP/VMS
System Access Programming Guide

Order Number: AA-GH75A-TE

May 1986

This document contains information required by a LISP language programmer to make use of routines and other facilities offered by the VMS operating system.

Operating System and Version:	VAX/VMS Version 4.2
Software Version:	VAX LISP/VMS Version 2.0

digital equipment corporation
maynard, massachusetts

First Printing, May 1986

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Printed in U.S.A.

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	UNIBUS	PDP
DECUS	VAX	VMS
MicroVAX	MicroVAX II	MicroVMS
VAXstation	VAXstation II	AI VAXstation
DECnet	ULTRIX	ULTRIX-11
ULTRIX-32		
ULTRIX-32m	digital™	

CONTENTS

FIGURES

TABLES

PREFACE

Manual Objectives

The *VAX LISP/VMS System Access Programming 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 manual is intended for programmers with a good knowledge of both LISP and the programming interface to the VMS operating system.

Structure of This Document

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

PART I: GUIDE TO SYSTEM ACCESS PROGRAMMING

Part I consists of five chapters, which explain how to use the VAX LISP interface to operating system routines.

- Chapter 1 provides an overview of the VAX LISP system access facilities.
- Chapter 2 shows how to define an external (system) routine and how to call it from LISP.
- Chapter 3 explains alien structures, which allow you to exchange data between LISP and routines written in other languages.
- Chapter 4 describes interrupt functions, which you can use to handle asynchronous events in the operating system.

- Chapter 5 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.

PART II: OBJECT DESCRIPTIONS

Part II contains full descriptions of the functions, macros, variables, and constants involved with system access. Each function or macro description explains the function's or macro's use and shows its format, applicable arguments, return value, and examples of use. Each variable or constant description explains the variable's or constant's use and provides examples of its use.

Associated Documents

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

- *VAX LISP/VMS User's Guide*
- *COMMON LISP: The Language*
- *VAX/VMS Linker Reference Manual*
- *Introduction to VAX/VMS System Routines*
- *VAX/VMS System Services Reference Manual*
- *VAX/VMS Run-Time Library Routines Reference Manual*
- *VAX/VMS Record Management Services Reference Manual*
- *VAX/VMS Utility Routines Reference Manual*
- *VAX Architecture Handbook*

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

Conventions Used in This Document

The following conventions are used in this manual:

Convention	Meaning
( )	Parentheses used in examples of LISP code indicate the beginning and end of a LISP form. For example: `(SETQ NAME LISP)`

Convention	Meaning
[ ]	Square brackets enclose elements that are optional. For example: [*doc-string*] Square brackets do not indicate optional elements when they are used in the syntax of a directory name in a VAX/VMS file specification. Here, the square bracket characters must be included in the syntax.
...	A horizontal ellipsis means that the element preceding the ellipsis can be repeated. For example: *function-name* ...
{ }	In function and macro format specifications, braces enclose elements that are considered to be one unit of code. For example: {*keyword value*}
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered to be 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 after it are defined to be optional. For example: PPRINT *object* &OPTIONAL *package* 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 the function or macro whose definition includes &REST.

Convention	Meaning
&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 ... Do not specify &KEY when you invoke the function or macro whose definition includes &KEY.
UPPERCASE characters	DCL commands and qualifiers, and defined LISP functions, macros, variables, and constants are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters.
lowercase italics	Lowercase italics in function and macro descriptions and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters.
<RET>	A symbol with a 1- to 3-character abbreviation indicates that you press a key on the terminal. For example: <RET> or <ESC> In examples, carriage returns are implied at the end of each line. However, the <RET> symbol is used in some examples to emphasize carriage returns.
CTRL/x	CTRL/x indicates a control key sequence where you hold down the CTRL key while you simultaneously press another key. For example: CTRL/C or CTRL/Y The system echoes control key sequences as ^x; therefore, in examples of output, CTRL/x is shown as ^x. For example: ^C or ^Y
. . .	A vertical ellipsis indicates that all the information that the system would display in response to the particular function call is not shown; or, that all the information a user is to enter is not shown.

Convention	Meaning

Black print	In examples, output lines and prompting characters that the system displays are in black print. For example:
	
	`$ LISP/COMPILE`
	`$_File(s): MYPROG.LSP`

Red print	In examples, user input is shown in red print. For example:
	
	`$ LISP/COMPILE`
	`$_File(s): MYPROG.LSP`

PART I

GUIDE TO SYSTEM ACCESS PROGRAMMING

CHAPTER 1

OVERVIEW OF SYSTEM ACCESS FACILITIES

The VAX LISP system is layered on top of the VMS operating system. If you restrict your use of LISP to COMMON LISP functions and use the utilities provided with VAX LISP, you may hardly notice the operating system. VAX LISP, however, provides various means of access to the facilities of the operating system. This chapter provides a broad view of those means of access. The remainder of this manual 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 Calling Standard can be called by, and can return values to, routines written in other languages. For example, a LISP program can call a numeric analysis routine written in FORTRAN.

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 THE CALL-OUT FACILITY

As a VAX LISP programmer, your primary means of access to routines external to LISP is the call-out facility. To use the call-out 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 Volume 5 of the VAX/VMS document set, or the *MicroVMS FORTRAN Programming Support Manual* of the MicroVMS document set. 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 routine's arguments. The VAX data types and passing mechanisms of those arguments are especially important.

Once you have identified or written an external routine, you must define it, using the DEFINE-EXTERNAL-ROUTINE macro. This macro makes known to LISP 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.

1.2 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 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 also provides a constructor function, field accessor functions, a type predicate, and so on.

You pass an instance of an alien structure to an external routine using CALL-OUT. Since DEFINE-ALIEN-STRUCTURE provides precise control over the memory layout of the structure, you can set up the alien structure so that the external routine can properly map its own data types into it. 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.

1.3 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.

In the operating system, 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 workstation user moves the pointing device or presses a pointer button.

A number of system routines initiate operating system 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 in the operating system. 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.

1.4 CONTROLLING INTERRUPTIONS AND SYNCHRONIZING EXECUTION

VAX LISP allows you to 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 parts of code -- for example, those that modify 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

CALLING EXTERNAL ROUTINES

VAX LISP has a facility that lets you call routines written in other languages from within a VAX LISP program. Using this facility, VAX LISP programs can call 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

To call an external routine, the routine must follow the VAX Procedure Calling Standard. In addition, if you call a routine that is not in the RTL or that is not a system service, the routine must be linked into a position-independent shareable image.

The call-out facility cannot call external routines that require an extensive, nonstandard software environment. 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 Part II).

Programs written in other VAX languages cannot call VAX LISP functions. The reason is that most functions written in LISP depend on an entire LISP environment being present at run time. As an example of this dependency, take garbage collection. If the LISP function that was called from another language, for example, FORTRAN, ran out of dynamic memory, if would normally cause a garbage collection in the LISP environment. However, since the whole LISP environment is not present when the LISP function is called, the FORTRAN program would have to deal with the memory management tasks normally performed by the garbage collector. This would require the FORTRAN program to have knowledge of the internals of the LISP system.

The *Introduction to VAX/VMS System Routines* contains detailed information about calling external routines and passing arguments. You should be familiar with these subjects before you use the VAX LISP call-out facility.

A routine that can be called is termed a "procedure" in the previously mentioned manual. This chapter, however, uses the expression "external routine" to maintain consistency with the VAX LISP language terminology.

This chapter covers the following:

- Lists the steps to take in calling an external routine.
- Describes the standard VAX calling conventions.
- Explains and gives examples of how to define and call external routines, including system services.
- Shows how data types are converted from LISP objects to VAX objects and vice versa.
- Explains the errors that can occur while executing an external routine.
- Shows how a LISP system containing external routine definitions is suspended.

2.1 STEPS TO TAKE IN CALLING AN EXTERNAL ROUTINE

For a LISP program to call an external routine, you must:

1. Write the external routine.
2. Compile it.
3. Debug it.
4. Link it into a VMS shareable image.
5. Define it in LISP.
6. Call it from LISP.

Figure 2-1 illustrates these steps. Note that VAX LISP currently has no way to debug external routines.

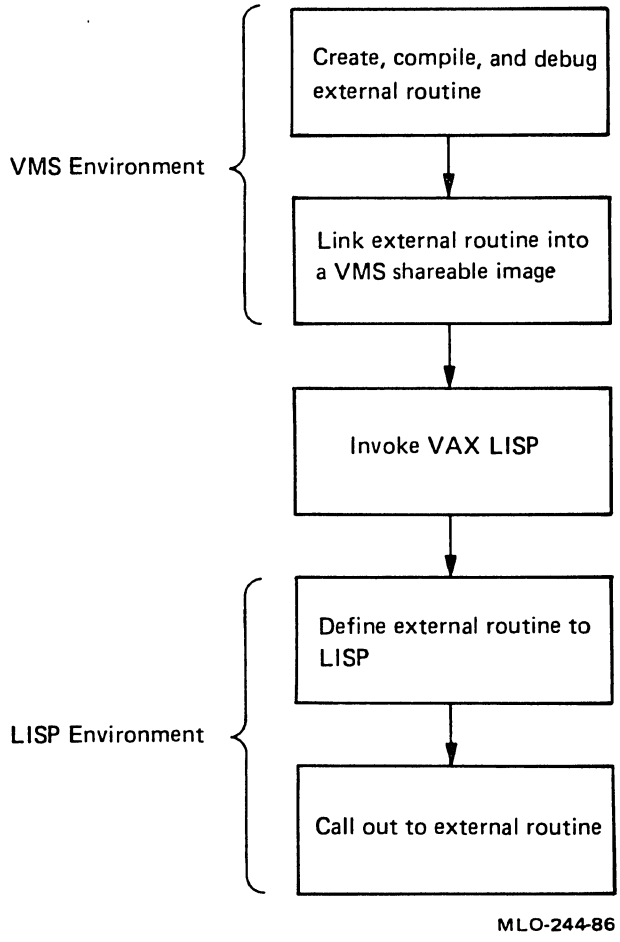


Figure 2-1: Calling External Routines

2.2 STANDARD VAX CALLING CONVENTIONS

The VAX Procedure Calling Standard defines a uniform method for routines to call one another -- see the *Introduction to VAX/VMS System Services*. This standard prescribes how routines receive and return control, how arguments are passed, and how function values are returned. By means of the standard call conventions, most languages used with the VAX/VMS operating system can call library routines and routines written in other VAX native-mode languages. Interpreted and compiled VAX LISP programs cannot conform to the standard because of the nature of the LISP language. For this reason, VAX LISP provides a facility that lets you call routines written in other VAX languages that do conform to the standard. The next four sections briefly summarize how VAX LISP calls routines conforming to the VAX Procedure Calling Standard.

2.2.1 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.

2.2.2 Argument Lists

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 in the list 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.

2.2.3 Mechanisms for Passing Arguments

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

- By immediate value -- The argument list contains the value.
- By reference -- The argument list contains the address of the value.
- By descriptor -- The argument list contains the address of a descriptor of the value.

Section 2.4.7.3 describes how to specify an argument's passing mechanism.

2.2.4 Values Returned by Functions

An external routine can be a subroutine or a function. A subroutine is invoked only to produce side effects, and returns no value as a result of its execution. A function, on the other hand, returns a value after execution and might 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.

2.3 LINKING A SHAREABLE IMAGE

Before a LISP program can call external routines, you must link the required object modules into one or more position-independent VMS shareable images. The following example links the object modules TEST.OBJ and FUN.OBJ into a shareable image called MYIMAGE.EXE. The linker option UNIVERSAL is used to list the entry points that are available to LISP through the call-out facility. As the values of the UNIVERSAL option in the following example are taken from SYS$INPUT, <CTRL/Z> is used to specify the end of file.

```
$ LINK/SHAREABLE=MYIMAGE TEST,FUN,SYS$INPUT:/OPTIONS
UNIVERSAL=ENTRY_1,ENTRY_2,ENTRY_3
<CTRL/Z>
```

The number of individual shareable images that can be mapped into VAX LISP depends on VMS shareable image restrictions and the available address space.

If you specify a base address as an option in a command that invokes the VMS linker or if the linker issues a warning message that informs you that the shareable image is based, you cannot call external routines in that image.

You can call external routines in shareable images that contain writable sections. Routines that are written in VAX-11 FORTRAN, which use COMMON blocks, are examples of routines that 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, shareable to writable, not shareable.

For example, suppose you have two routines which access a named COMMON block called SHARED_SPACE. These routines exist in different source modules, FOR1.FOR and FOR2.FOR. After compiling both source modules, 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 using the VMS INSTALL utility. 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 the *VAX/VMS Linker Reference Manual*.

2.4 DEFINING AN EXTERNAL ROUTINE

Programs written in VAX LISP cannot call external routines the same way as programs written in other VMS languages. When a program calls an external routine, the program must specify information about the routine. Other VMS languages specify the information by compiling code into object modules that are linked by the VMS linker. Since VAX LISP does not create object modules that can be linked, it must specify information about an external routine another way.

After you link an external routine into a shareable image, enter the VAX LISP environment and define the routine, using the VAX LISP DEFINE-EXTERNAL-ROUTINE macro. The definition provides the VAX LISP system with the information needed to create an argument list and to locate and call the external routine. A description of the DEFINE-EXTERNAL-ROUTINE macro is provided in Part II.

The format for defining an external routine is:

```
(DEFINE-EXTERNAL-ROUTINE (routine-name keyword-1 value-1
                                       keyword-2 value-2
                                       ...)
  [doc-string]
  (argument-name keyword-1 value-1
                 keyword-2 value-2
                 ...)
  (argument-name ...) ...)
```

The following example illustrates an external routine definition. The keywords used in this example are explained in the next sections. An illustration of calling out to this external routine is given in Section 2.5.1.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE (LIB$CREATE_DIR
                                :FILE "LIBRTL"
                                :CHECK-STATUS-RETURN T)
          (DEV-DIR-SPEC :LISP-TYPE STRING))
LIB$CREATE_DIR
```

This external routine will create a new disk directory given its device and directory specification.

2.4.1 External Routine Name 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.

2.4.2 External Routine Name

The external routine name is a symbol that uniquely identifies that routine among all external routines being defined. The name also serves as the entry-point name unless a different entry-point name is specified with the :ENTRY-POINT option (see Section 2.4.3.2).

2.4.3 External Routine Options

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.

Specify external routine options in a list whose first element is the name of the routine the options characterize. The format in which to specify the name and options is:

(name keyword-1 value-1 keyword-2 value-2 ...)

Option values are not evaluated. Table 2-1 alphabetically lists the option keywords you can use. The next sections explain each option in detail. For examples of how to use these options, see Section 2.10.

Table 2-1: Keywords Specifying External Routine Options

Keyword	Purpose
:CHECK-STATUS-RETURN	To check the return status
:ENTRY-POINT	To name the entry point
:FILE	To specify the VMS file name of the external routine's shareable image
:RESULT	To define the data type of the result
:TYPE-CHECK	To check the data types of the arguments

NOTE

You must specify the :FILE option unless you are calling a system service.

2.4.3.1 **Checking the Return Status** - The :CHECK-STATUS-RETURN keyword specifies whether the call-out facility is to examine the contents of register R0 on return from the external routine. The default is NIL, which means that no checking is done. If you specify a T, the R0 register is assumed to contain a status code. If the severity of the status is warning, error, or severe-error, a continuable error is signaled. The presence of this option implies that the external routine returns an integer; thus, you should not specify the :RESULT option with this option.

2.4.3.2 **Naming the Entry Point** - The :ENTRY-POINT keyword specifies the entry-point name of an external routine. You must specify this keyword with a string that represents the name of the entry point that is to be called if that string is different from the name you specify for the external routine. (The default entry point is the print name of the external routine.)

2.4.3.3 **Specifying the Shareable Image** - The :FILE keyword specifies a string that represents the specification of the external routine's shareable image. The specification must be in upper case and must be either a logical name that refers to the shareable image, or the file name of a shareable image in the SYS$SHARE directory. The specification cannot be an arbitrary file specification. You must include this keyword in a routine's definition unless you are calling a system service.

Note that this keyword is named :IMAGE-NAME in Version 1 of VAX LISP. For the sake of compatibility, either keyword (:FILE or :IMAGE-NAME) is allowed. This change was made for compatibility with the ULTRIX operating system.

NOTE

If the specified entry point is already available in LISP, modified versions of the routine are not used.

2.4.3.4 **Specifying the Result Data Type** - 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.

If the routine does return a value, then the :RESULT keyword can specify the LISP (or both a LISP and a VAX) data type that the external routine is to return to the LISP system. Specify the value with :RESULT as a LISP data type. If the VAX type of the returned value does not correspond with the LISP data type, use a list of the format (:LISP-TYPE lisp-type :VAX-TYPE vax-type). See Table 2-4 for valid result types. Do not specify both the :CHECK-STATUS-RETURN keyword and the :RESULT keyword.

2.4.3.5 **Checking the Argument Data Types** - The :TYPE-CHECK keyword specifies that the data types of the arguments passed to an external routine be checked for compatibility with the argument descriptions.

You can specify the keyword with either T or NIL. If you specify T, the LISP system generates code that checks the type of actual LISP objects when you call the CALL-OUT macro. 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

Type checking adds considerable overhead to the call-out process.

2.4.4 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 in the definition after the name and options list.

2.4.5 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 a routine.

If the default characteristics are adequate, then an argument description is nothing more than the name of the argument. Otherwise, the argument description is a list whose first element is the name and whose remaining elements specify the characteristics.

2.4.6 Argument Name

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

2.4.7 Argument Options

You can define the characteristics of an external routine argument by specifying options in the argument description. Each option consists of a keyword-value pair. Specify options in a list whose first element is the name of the argument they characterize. The format is:

```
(argument-name keyword-1 value-1 keyword-2 value-2 ...)
```

Option values are not evaluated. Table 2-2 is an alphabetical list of the argument-option keywords with the values they define:

Table 2-2: Keywords Specifying External Routine Argument Options

Keyword	Value Defined	Options
:ACCESS	Access capability	:IN (default) :IN-OUT
:LISP-TYPE	LISP type	see Table 2-4 INTEGER (default)

Table 2-2 (cont.)

Keyword	Value Defined	Options
:MECHANISM	Passing mechanism	:VALUE :REFERENCE (default except for :VAX-TYPE :TEXT) :DESCRIPTOR (default for :VAX-TYPE :TEXT)
:VAX-TYPE	VAX data type	see Table 2-5 (default depends on LISP type)

2.4.7.1 **Access Capability** - The :ACCESS keyword specifies the access capability for an argument. The possible values are :IN for input access and :IN-OUT for both input and output access. The default is :IN. Since 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 both input and output access, the external routine can obtain the argument's value and optionally modify it. The argument must be specified as 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. See Section 2.6.2 for more details.

2.4.7.2 **LISP Data Type** - The :LISP-TYPE keyword defines the LISP data type of an argument. Specify this keyword with the types shown in Table 2-4. The LISP type defaults to INTEGER.

If the values you specify for the LISP data type and the VAX data type are incompatible, an error is signaled.

2.4.7.3 **Passing Mechanism** - The :MECHANISM keyword defines the mechanism by which an argument is to be passed to an external routine. With the :MECHANISM keyword, you can specify one of the three values in Table 2-3. These values correspond to the three defined mechanisms described in Section 2.2.3.

Table 2-3: Values of the :MECHANISM Keyword

Value Name	Corresponding VAX Mechanism	Description
:VALUE	Immediate Value	The immediate value mechanism 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	The reference mechanism passes the address of the argument in the argument list.
:DESCRIPTOR	Descriptor	The descriptor mechanism passes the address of an argument descriptor in the argument list. The descriptor is a data structure that contains the address of the argument, as well as its data type and size.

Note that these keyword value names have been changed. In Version 1 of VAX LISP, :VALUE was :IMMED, :REFERENCE was :REF, and :DESCRIPTOR was :DESCR. For the sake of compatibility, both the old and the new names can be used.

You cannot specify specific VMS descriptor classes in external routine definitions. The DEFINE-EXTERNAL-ROUTINE macro assigns an appropriate descriptor class to a routine when the LISP system evaluates it. The values the macro assigns are DSC$K_CLASS_S or DSC$K_CLASS_A. To pass an argument using a user-specified descriptor, define the descriptor and the argument to be alien structures and pass the alien-structure descriptor with the reference mechanism. For information on defining alien structures, see Chapter 3.

2.4.7.4 **VAX Data Type** - The :VAX-TYPE keyword defines the VAX data type of the argument. Specify this keyword with the types in Table 2-4. The default depends on the LISP type, also in Table 2-4.

2.5 CALLING AN EXTERNAL ROUTINE

This section describes how to call an external routine, what the CALL-OUT macro does, and how the CALL-OUT macro uses internal data structures.

2.5.1 How to Call an External Routine

You call an external routine by using the VAX LISP CALL-OUT macro with:

- The defined name of the external routine.
- Arguments to be passed to the external routine. These must be compatible with the arguments defined in the call to the DEFINE-EXTERNAL-ROUTINE macro.

The format for calling an external routine is:

```
(CALL-OUT routine-name arg1 arg2 ...)
```

The following example calls out to the external routine LIB$CREATE_DIR defined in Section 2.4. This call will create the new directory LISPW$:[MYNAME.WORK]:

```
Lisp> (CALL-OUT LIB$CREATE_DIR "LISPW$:[MYNAME.WORK]")
561
```

If you specify fewer arguments to the CALL-OUT macro than those defined, the remaining defined 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 those defined, an error is signaled.

If an argument evaluates to NIL, a zero is placed in the corresponding argument list longword. The zero is normally used to mean that an optional argument is not desired.

2.5.2 What the CALL-OUT Macro Does

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.

4. Transfers control to the external routine.

5. Returns any specified result from the external routine, or no values if there is no result.

2.5.3 How the CALL-OUT Macro Uses Internal Data Structures

When you define an external routine, an internal data structure is created and associated with the symbol naming that routine. This data structure is then used by both the CALL-OUT macro and the resulting LISP code. Therefore, you must ensure that an external routine is defined before it is called.

In particular, when running LISP functions, make sure the file containing the definition is loaded before calling out to that external routine.

2.6 DATA TYPE CONVERSIONS

The internal representation of LISP objects differs from the standard VAX format for the corresponding data types. The call-out facility converts the LISP argument to a VAX data type before passing the argument to an external routine. Likewise, after the external routine returns, the call-out facility converts the resulting VAX data to a LISP object before the it can return the data to the LISP system.

2.6.1 Converting LISP Objects to VAX Data Types

The call-out facility must convert the arguments for an external routine from a LISP object to a VAX data type. This conversion is controlled by the :LISP-TYPE and :VAX-TYPE options in an argument definition. Table 2-4 shows the valid combinations of LISP data types and VAX data types. For each LISP type, the default VAX type is marked with an asterisk.

Table 2-4 also shows the passing mechanisms (V = Value, R = Reference, and D = Descriptor) that are valid for each combination. In addition, the table specifies the descriptor class and data type that will be included in the argument descriptor when passing by descriptor. The descriptor formats, descriptor class, and data type codes are described in the *Introduction to VAX/VMS System Routines*.

Table 2-4: Conversion Table from LISP Type to VAX Type

LISP Type	Default	VAX Type	Mechanisms Allowed	Descriptor Class/ Data Type
CHARACTER	*	:UNSIGNED-BYTE	V,R,D	Scalar/BU
INTEGER		:BIT	V,R,D	Scalar/LU
INTEGER		:BYTE	V,R,D	Scalar/B
INTEGER		:UNSIGNED-BYTE	V,R,D	Scalar/BU
INTEGER		:WORD	V,R,D	Scalar/W
INTEGER		:UNSIGNED-WORD	V,R,D	Scalar/WU
INTEGER	*	:LONGWORD	V,R,D	Scalar/L
INTEGER		:UNSIGNED-LONGWORD	V,R,D	Scalar/LU
INTEGER		:QUADWORD	R,D	Scalar/Q
INTEGER		:UNSIGNED-QUADWORD	R,D	Scalar/QU
SINGLE-FLOAT	*	:F-FLOATING	V,R,D	Scalar/F
DOUBLE-FLOAT	*	:G-FLOATING	R,D	Scalar/G
DOUBLE-FLOAT		:D-FLOATING	R,D	Scalar/D
LONG-FLOAT	*	:H-FLOATING	R,D	Scalar/H
STRING	*	:TEXT	R,D	Scalar/T
STRING		:ASCIZ	R	
SIMPLE-BIT-VECTOR	*	:BIT	R,D	Scalar/V
SIMPLE-BIT-VECTOR		:UNSIGNED-LONGWORD	V,R,D	Scalar/LU
ALIEN-STRUCTURE	*	:UNSPECIFIED	R,D	Scalar/Z
(ARRAY CHARACTER)	*	:UNSIGNED-BYTE	R,D	Array/BU
(SIMPLE-ARRAY BIT)	*	:BIT	R,D	Array/V
(ARRAY (UNSIGNED-BYTE 8))	*	:UNSIGNED-BYTE	R,D	Array/BU

Table 2-4 (cont.)

LISP Type	Default	VAX Type	Mechanisms Allowed	Descriptor Class/ Data Type
(ARRAY (UNSIGNED-BYTE 16))	*	:UNSIGNED-WORD	R,D	Array/WU
(ARRAY (SIGNED-BYTE 32))	*	:LONGWORD	R,D	Array/L
(ARRAY SINGLE-FLOAT)	*	:F-FLOATING	R,D	Array/F
(ARRAY DOUBLE-FLOAT)	*	:G-FLOATING	R,D	Array/G
(ARRAY LONG-FLOAT)	*	:H-FLOATING	R,D	Array/H

2.6.2 Arguments with :IN-OUT Access

Arguments with both input and output access can be modified by the external routine. If the argument is a character or a number, the modified value will be made into a new LISP object that is distinct from the original argument. This action ensures that you can pass constants or shared data objects and they will not be modified.

If the argument is not a character or a number, then the argument will be directly modified by the external routine, and no copy is made. This means that all array arguments are modified in place.

2.6.3 :ASCIZ 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 will not notice this and shorten the string. You must take care of this situation yourself.

2.6.4 Converting VAX Data Types to LISP Objects

The call-out facility must convert the VAX data resulting from the execution of an external routine to a LISP object before the facility

can return the data to the LISP system. Table 2-5 shows the valid combinations of LISP data types and VAX data types. It also specifies the location of the result on return from the external routine. The default cases, marked with an asterisk, require that you specify only the LISP type with the :RESULT keyword (see Section 2.4.3). All other cases require that you specify both the LISP and the VAX types. Since you must always specify the LISP type, that type is in the first column of the table.

Table 2-5: Conversion Table from VAX Type to LISP Type

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

2.7 CALLING SYSTEM SERVICES

The call-out facility provides a mechanism for LISP programs to call standard VMS and RMS system services. Sections 2.7.1 and 2.7.2 list how to define and call system services. Section 2.10 provides examples of calling system services. For a listing of VMS services

and information on those services, see the *VAX/VMS System Services Reference Manual*. For a listing of RMS services and information on those services, see the *VAX Record Management Services Reference Manual*.

2.7.1 Defining System Services

Defining VMS and RMS system services is similar to defining other external routines with a few restrictions. You must be familiar with the explanation of defining an external routine in Section 2.4 to understand the following restrictions:

- You must omit the file name argument from the DEFINE-EXTERNAL-ROUTINE macro specification. Omission of this argument causes the macro to assume that the function being defined is a system service. If you use the name of a system service but supply a file name (not NIL), the LISP system assumes that you want an entry point in an ordinary shareable image of that name rather than the VMS system service given as the external routine name.

- The entry-point name in the DEFINE-EXTERNAL-ROUTINE macro specification must be one of the system service entry points.

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

- If a system service resides in a shareable image (for example, SYS$MOUNT in MOUNTSHR), then the system service is defined as any other external routine in a shareable image. For purposes of this discussion, the system service is is not considered a system service.

See Section 2.10 for examples of defining external routines that call out to system services.

2.7.2 Calling out to System Services

Calling VMS and RMS system services is similar to calling other external routines with a few restrictions. You must be familiar with the explanation of calling out to an external routine in Section 2.5 to understand the following restrictions:

- You must always call system services with a complete argument list, even if you omit the last several arguments. 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 IOSB for SYS$QIO). These data structures must be created as alien structures and statically allocated so that they will not be moved by VAX LISP (See Chapter 3).

- 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.

2.8 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 might occur during these operations include VMS errors that occur while you are 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

The VAX LISP error handler regards signaled conditions as fatal errors (including conditions that have a success status).

Status codes returned by an external routine, however, do not always represent uncorrectable errors. The operation that the call-out facility performs when a routine returns a status code is determined by the value that is specified with the :CHECK-STATUS-RETURN keyword (see Section 2.4.3) in the routine's definition. If the value is T, the facility 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 :CHECK-STATUS-RETURN keyword is specified with 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 was not specified in the UNIVERSAL option to the VMS linker when the shared image was created.

2.9 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 be aware of certain restrictions to ensure correct operation of the resumed system. 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, and they cannot be automatically remapped during a resume operation that follows a suspend 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 might create undesirable results. Undesirable results can occur from the following:

- Memory acquired with LIB$GET_VM
- Data initialization
- Open files
- Undefined logical names

2.9.1 Acquiring Memory 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 and is not remapped by a resume operation. This prevents you from storing 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.

2.9.2 Initializing Data

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, which 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.

2.9.3 Using 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.

2.9.4 Using Logical Names

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

2.10 EXAMPLES OF USING THE CALL-OUT FACILITY

The following examples show both how to define external routines and how to call out to them.

1.
```
Lisp> (DEFINE-EXTERNAL-ROUTINE (MTH$ACOSD
                                :FILE "MTHRTL"
                                :RESULT (:LISP-TYPE
                                            SINGLE-FLOAT
                                         :VAX-TYPE
                                            :F-FLOATING))
        "This routine returns the arc cosine
         of an angle in degrees."
        (X :LISP-TYPE SINGLE-FLOAT
           :VAX-TYPE :F-FLOATING))
MTH$ACOSD
```

   Defines an RTL routine, called MTH$ACOSD, which returns the arc cosine of an angle in degrees. The routine takes one read-only argument, which is a F_floating number, and returns the result as a F_floating number.

```
Lisp> (CALL-OUT MTH$ACOSD 0.5)
60.0
```

   Calls the RTL routine MTH$ACOSD, and returns the routine's value.

2.
```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SMG$CREATE_PASTEBOARD
                                  :FILE "SMGSHR"
                                  :RESULT INTEGER)
        (NEW_PASTEBOARD-ID :LISP-TYPE INTEGER
                           :VAX-TYPE :UNSIGNED-LONGWORD
                           :ACCESS :IN-OUT)
        (OUTPUT-DEVICE :LISP-TYPE STRING)
        (PB-ROWS :LISP-TYPE INTEGER :ACCESS :IN-OUT)
        (PB-COLUMNS :LISP-TYPE INTEGER :ACCESS :IN-OUT)
        (PRESERVE-SCREEN-FLAG :LISP-TYPE INTEGER
                              :VAX-TYPE :UNSIGNED-LONGWORD))

SMG$CREATE_PASTEBOARD
```

Defines the SMG screen management routine called SMG$CREATE_PASTEBOARD.

```
Lisp> (DEFVAR *PASTEBOARD-ID* -1)
*PASTEBOARD-ID*
```

Defines a special variable that will contain the pasteboard ID returned by the external routine.

```
Lisp> (CALL-OUT SMG$CREATE_PASTEBOARD *PASTEBOARD-ID*
                                      NIL NIL NIL 1)
1
```

Calls the external routine SMG$CREATE_PASTEBOARD, specifying the special variable to receive the pasteboard ID. Three arguments are omitted, and a preserve-screen-flag of 1 is given. The result status is returned.

3. This example shows you how to call out to an external routine that is written in FORTRAN:

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

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

Defines a function written in FORTRAN, called NUMBERS, which manipulates two integers and returns an integer.

```
$ FORTRAN NUMBERS
```

Compiles the FORTRAN function NUMBERS.

```
$ LINK/SHAREABLE=DBA2:[SMITH]EXAMPLE NUMBERS,SYS$INPUT:/OPTIONS
UNIVERSAL=NUMBERS
<CRTL/Z>
$ DEFINE EXAMPLE DBA2:[SMITH]EXAMPLE
```

Links the FORTRAN function NUMBERS into a shareable image. The name NUMBERS is specified as an entry point that is globally available. A logical name is defined to refer to the new shareable image.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
        (NUMBERS :FILE "EXAMPLE"
                 :RESULT INTEGER)
           X Y)
NUMBERS
```

Defines an external routine, called NUMBERS, which manipulates two integers and returns an integer. The logical name is specified for the shareable image because the name is not in SYS$SHARE. The arguments do not have options because, by default, the arguments are assumed to be longword integers that are passed by reference.

```
Lisp> (CALL-OUT NUMBERS 5 7)
23536
```

Calls the external routine NUMBERS, which returns the function value.

4. This example illustrates a more complex use of the call-out facility. Assume that an external routine named COMPLEX exists outside the LISP system.

```
Lisp> (DEFINE-ALIEN-STRUCTURE COMPLEX-NUMBER
        (REAL      :G-FLOATING 0 8)
        (IMAGINARY :G-FLOATING 8 16))
COMPLEX-NUMBER
```

Defines a complex number (double precision). The alien structure facility is used to define complex numbers because the data type cannot be represented directly in VAX LISP. Chapter 3 describes the alien structure facility.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
        (COMPLEX_EXP :FILE "COMPLEX")
        (OUTPUT :LISP-TYPE ALIEN-STRUCTURE
                :ACCESS :IN-OUT)
        (INPUT  :LISP-TYPE ALIEN-STRUCTURE))

COMPLEX_EXP
```

Defines the external routine, called COMPLEX_EXP, which uses complex numbers.

```
Lisp> (SETQ C1 (MAKE-COMPLEX-NUMBER :REAL 5.0d1
                                    :IMAGINARY 6.123456d-4))
#<Alien Structure COMPLEX-NUMBER #x5010324>
Lisp> (SETQ C2 (MAKE-COMPLEX-NUMBER :REAL 0.0d0
                                    :IMAGINARY 0.0d0))
#<Alien Structure COMPLEX-NUMBER #x5010348>
```

These expressions create two complex numbers, C1 and C2.

```
Lisp> (CALL-OUT COMPLEX_EXP C2 C1)
```

Calls the external routine COMPLEX_EXP. The routine changes the values in the alien structure C2. A value is not returned from the function call because the :RESULT option was not specified.

5.
```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SYS$DALLOC :RESULT INTEGER)
        (DEVNAM :LISP-TYPE STRING)
        (ACMODE :LISP-TYPE INTEGER :MECHANISM :IMMED))
SYS$DALLOC
```

Defines the VMS system service SYS$DALLOC.

```
Lisp> (CALL-OUT SYS$DALLOC "TTH7:" NIL)
2312
```

Calls the VMS system service SYS$DALLOC. NIL is specified to account for the omitted argument; this ensures that the correct number of arguments are specified.

6. Suppose that the LISP variables OLD and NEW are bound to statically allocated alien structures (see Chapter 3), which are the file attribute blocks to be used in a rename operation.

```
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
```

Defines the RMS system service SYS$RENAME.

```
Lisp> (CALL-OUT SYS$RENAME OLD NIL NIL NEW)
1
```

Calls the RMS system service SYS$RENAME. 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.

CHAPTER 3

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 typedef 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 utilizing VAX data structures that LISP code cannot ordinarily access. Like a COMMON LISP structure, the definition of an alien structure causes the definition of a number of functions for the creation of alien structures, the accessing of fields or slots, and so on. The "alien" in the name "alien structure" refers to the structure's double purpose:

- To access data coded in a language foreign to LISP
- To make data coded in LISP available to a different language

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 call-out facility; they are used to create argument values for external routines that have arguments or control blocks too complicated for the call-out facility to convert (see Chapter 2).

This chapter describes:

- How to define an alien structure
- What the DEFINE-ALIEN-STRUCTURE macro does
- Components of an alien structure definition

- Examples of how to define alien structures
- How to create alien structures

The chapter also lists functions and macros you can use with the DEFINE-ALIEN-STRUCTURE macro. See Part II for a summary description of the DEFINE-ALIEN-STRUCTURE macro.

3.1 DEFINING AN ALIEN STRUCTURE DATA TYPE

Before you can create an alien structure, you must define that structure. You define the structure with the VAX LISP DEFINE-ALIEN-STRUCTURE macro. This macro is similar to the DEFSTRUCT macro described in *COMMON LISP: The Language*.

The DEFINE-ALIEN-STRUCTURE macro does not create a structure; rather this macro creates a definition of a structure. The LISP system treats this definition as a data type that you can then use to create individual structures of that type. This is different from the DEFUN (define function) macro that creates the function it defines.

The DEFINE-ALIEN-STRUCTURE macro is similar to the DEFSTRUCT macro in that both create a new compound data type (the data type contains more than one named component) and access, constructor, copier, predicate, and print functions. The DEFSTRUCT macro is different from the DEFINE-ALIEN-STRUCTURE macro in the kind of objects their defined data types contain. The DEFSTRUCT macro defines a type containing LISP objects while the DEFINE-ALIEN-STRUCTURE macro defines a type containing non-LISP objects.

The format of an alien structure definition is:

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

The following is an example alien structure definition:

```
(DEFINE-ALIEN-STRUCTURE SPACE
  "An example alien structure definition"
  (AREA-1 :SIGNED-INTEGER 0 4)
  (AREA-2 :SIGNED-INTEGER 4 8))
```

The preceding definition defines an alien structure named SPACE. This new data type is defined as an object consisting of two fields, AREA-1 and AREA-2, which are stored internally as VAX 32-bit integers. The numbers in the definition specify the structure's field lengths in bytes. See Sections 3.3 and 3.4 for a description of the components of an alien structure definition.

3.2 WHAT THE DEFINE-ALIEN-STRUCTURE MACRO DOES

When the LISP system evaluates the definition of an alien structure, the DEFINE-ALIEN-STRUCTURE macro automatically creates:

- **New data type**

  The name you give to the alien structure becomes a LISP data type. For example, the preceding definition creates the data type SPACE, which is a subtype of ALIEN-STRUCTURE.

- **Access functions**

  Access functions are created that can access the data in each data field of the defined alien structure. There are as many access functions as there are data fields in the alien structure. The DEFINE-ALIEN-STRUCTURE macro by default names each access function by prefixing each data field name with the name of the alien structure and a hyphen (-).

  In the preceding example, the access 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, access functions can have one or two arguments, depending on the complexity of the field the functions access.

  These access functions are acceptable access forms in a call to the SETF macro (unless :READ-ONLY T was specified as a field option -- see Section 3.4.4).

- **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 alien structures after you define them. 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 3.6 for information on keyword arguments the constructor function accepts.

- **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 a created alien structure. This copy is not a copy of a structure's definition, but a copy of a specific 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 occurrence 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 print 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 two 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. The number #x5036E8 is the memory address of that structure.

  If you want the print function to show the data in an alien structure, you must specify your own print function. See Section 3.3.2.5 on specifying a print function.

3.3 ALIEN STRUCTURE NAME, OPTIONS, AND DOCUMENTATION STRING

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.

3.3.1 Alien Structure Name

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
  ...)
```

If you specify options, specify the alien structure's name as the first element of a list whose other elements are separate lists for each option. For example:

```
(DEFINE-ALIEN-STRUCTURE (SPACE (option-1) (option-2) ...)
  ...)
```

NOTE

To use the same symbol both as the name of an alien structure data type and also as the name of a structure (DEFSTRUCT) data type is an error.

3.3.2 Options

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

- Change the default names of the access functions
- Change the default name of the constructor function
- 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 access, constructor, 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:

(alien-struc-name (keyword-1 value-1) (keyword-2 value-2) ...)

You can use the following keywords. The next sections explain each keyword in detail.

- :CONC-NAME -- to name access functions
- :CONSTRUCTOR -- to name the constructor function
- :COPIER -- to name the copier function
- :PREDICATE -- to name the predicate function
- :PRINT-FUNCTION -- to specify your own print function

3.3.2.1 **Naming Access Functions** - By default, the DEFINE-ALIEN-STRUCTURE macro produces names for an alien structure's access functions by prefixing each field name with the name of the alien structure and a hyphen (-). For example, the default names of the access functions created by the preceding definition are SPACE-AREA-1 and SPACE-AREA-2.

If you want to change the default names of an alien structure's access functions, specify the :CONC-NAME (concatenated name) keyword with a string (the prefix you want the names to have) 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 produces access 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.

The access functions can be used with SETF to change the value of a field.

3.3.2.2 **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.

If you want to change the default name of a constructor function, specify the :CONSTRUCTOR keyword with a string (the name 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 hyphenate your new name with the name of the structure, though it is appropriate for you to do that in the new name you create. 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 contructor functions do not take an argument list, although DEFSTRUCT constructor functions do take an argument list.

3.3.2.3 **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.

If you want to change the name of the copier function, specify the :COPIER keyword with a string (the name 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 produces 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.

3.3.2.4 **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.

If you want to change the name of the predicate function, specify the :PREDICATE keyword with a string (the name you want) 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 produces 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 access function of the P field. For example, with an alien struture of type SPACE, both the predicate function and the access function of the P field would have the same name, SPACE-P.

3.3.2.5 **Specifying a Print Function** - You can use the :PRINT-FUNCTION keyword option to specify the function that is to print an alien structure. You might want to do this since the default print function prints only the memory address of a structure; it does not print the contents of the structure's data fields. To alter the print representation of an alien structure, specify a print function in that alien structure's definition. The following example is of an alien structure definition specifying 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 have three arguments:

- A NAME indicating the alien structure to be printed
- A STREAM indicating the stream to print to
- 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 more useful with a structure than an alien structure. Consequently, that argument is often ignored with alien structures as in the following example of an alien structure print-function definition:

```
(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 the preceding example, the three arguments are ALIEN, STREAM, and DEPTH. The ALIEN argument refers to the individual alien structure to be printed. The STREAM argument is the stream to which to print.

The DEPTH argument is ignored here by using the DECLARE special form. The DEPTH argument can be compared with the value of *PRINT-LEVEL*, allowing you to control how deep the printer will print. This argument is useful with structures since you may wish to restrict the printer from printing all the information in a complex structure. However, this argument is ignored in this example because the fields of the alien structure are immediate objects, and so it is unnecessary to abbreviate the data fields printed.

If you want to use the DEPTH argument, see the the *PRINT-LEVEL* variable description in *COMMON LISP: The Language*.

The following example is the output as printed by the previous, user-defined print function:

```
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>
```

In the preceding example, the MAKE-SPACE function creates an individual structure of the previously defined type SPACE. In addition, the preceding, user-defined print function displays the contents of the new alien structure's data fields.

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

3.3.3 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 3.1.

3.4 ALIEN STRUCTURE FIELD DESCRIPTIONS

Alien structures are composed of data fields, each of which has a description in the alien structure definition. A data-field description contains:

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

When you define an alien structure, specify a field description as a list of the preceding elements whose first element is the field's name. Use this format:

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

For example:

```
(FIELD-1 :TEXT 0 9 :OCCURS 10 :OFFSET 15)
```

The following sections describe the elements in a field description.

3.4.1 Field Name

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

3.4.2 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.

3.4.2.1 **Given Field Types** - Table 3-1 lists the field types defined by VAX LISP. See Chapter 2 for more information on these types.

Table 3-1: Alien Structure Field Types

Type	Internal Storage Representation
:ASCIW	VAX character string; the first 16-bit word of the data vector contains a count of the number of characters in the string. You must allocate two bytes in addition to the maximum length of the string to hold this count.
:VARYING-STRING	A synonym for :ASCIW.
:ASCIZ	VAX character string terminated with the NULL character (0's in the last byte(s)). You must 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	A 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

NOTE

When you access a VAX :D-FLOATING type, the accessor converts it into a LISP DOUBLE-FLOAT, which is equilvalent to a VAX :G-FLOATING type.

:H-FLOATING	H_floating data

Table 3-1 (cont.)

Type	Internal Storage Representation
:POINTER	(See below)
:SELECTION	(See below)

The :POINTER and the :SELECTION types have the following explanations:

:POINTER

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 pointing 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 statically allocate alien structures, see Section 3.6.2.

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 start 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 is of a data field with the type :POINTER.

```
(AREA-1 (:POINTER SPACE) 0 4)
```

:SELECTION

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, take the following alien structure definition with one field of type :SELECTION.

```
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 give when creating an alien structure with the item in the selection list of the definition. 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 is used to access the field as an unsigned integer:

```
Lisp> (ALIEN-FIELD GEO :UNSIGNED-INTEGER 0 4)
0
```

Notice the actual value stored in the field is 0 since "MASSACHUSETTS" is the 0'th 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
```

3.4.2.2 **User-Defined Field Types** - In addition to the given field types, you can define your own field types with the DEFINE-ALIEN-FIELD-TYPE macro. See Part II for a description of this macro.

3.4.3 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 that determine the start and end positions of the field. 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)
```

3.4.3.1 **Start and End Positions** - The start position is inclusive and the end position is exclusive. That is, the first field in an alien structure's data area starts in position 0, and the last position in a field is the position preceding the field's end-position value. For example, if a field's start position is 0 and its end position is 4, the field occupies positions 0 to 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. However, because the units are 8-bit bytes, a start or end value with a denominator that does not divide 8 (for example, 1/3) causes an error when you call the DEFINE-ALIEN-STRUCTURE macro.

Some exceptions: all values that are strings or are of type :F-FLOATING, :G-FLOATING, :D-FLOATING, or :H-FLOATING 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.

3.4.3.2 **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 (see Section 3.4.4.4), you might produce gaps in an alien structure. See the second example in Section 3.5 for an illustration of gaps.

Even though gaps can exist between fields or at the beginning of a field -- if the first field does not start at 0, only the ALIEN-FIELD function (see Section 3.7) can access gaps. The LISP system does not generate forms that access or set fields that include gaps; that is, LISP-level code does not process gaps.

3.4.3.3 **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, the other overlapping 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 3.4.4.1 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
```

3.4.4 Field Options

By specifying options in the data-field descriptions of an alien structure's definition, you can define the following characteristics of that structure's data fields.

- Whether a field has an initial value
- Whether a field is read-only
- Whether a field repeats and how often
- The distance between similar fields

Specify a data-field option as a keyword and a value. Include that option in a list whose first element is the name of the field the option characterizes. You can specify more than one option at a time. The format is:

(field-name keyword-1 value-1 keyword-2 value-2 ...)

you can use the following keywords. The next sections explain each keyword in detail.

- :DEFAULT -- Gives an initial value to a field
- :READ-ONLY -- Tells if a field can be set
- :OCCURS -- Tells the number of times a field repeats
- :OFFSET -- Tells the distance between similar fields

3.4.4.1 **Initial Value** - To specify an initial value for a field, specify that value with 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

3.4.3.2 **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 (see Section 3.4.4.4), you might produce gaps in an alien structure. See the second example in Section 3.5 for an illustration of gaps.

Even though gaps can exist between fields or at the beginning of a field -- if the first field does not start at 0, only the ALIEN-FIELD function (see Section 3.7) can access gaps. The LISP system does not generate forms that access or set fields that include gaps; that is, LISP-level code does not process gaps.

3.4.3.3 **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, the other overlapping 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 3.4.4.1 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
```

3.4.4 Field Options

By specifying options in the data-field descriptions of an alien structure's definition, you can define the following characteristics of that structure's data fields.

- Whether a field has an initial value
- Whether a field is read-only
- Whether a field repeats and how often
- The distance between similar fields

Specify a data-field option as a keyword and a value. Include that option in a list whose first element is the name of the field the option characterizes. You can specify more than one option at a time. The format is:

(field-name keyword-1 value-1 keyword-2 value-2 ...)

you can use the following keywords. The next sections explain each keyword in detail.

- :DEFAULT -- Gives an initial value to a field
- :READ-ONLY -- Tells if a field can be set
- :OCCURS -- Tells the number of times a field repeats
- :OFFSET -- Tells the distance between similar fields

3.4.4.1 **Initial Value** - To specify an initial value for a field, specify that value with 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 initial contents of a field are unpredictable.

3.4.4.2 **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 access functions that are not acceptable access forms in a call to the SETF macro. That is, if you specify the keyword-value pair :READ-ONLY T in a data-field description, you cannot use the SETF macro on the accessor function for that field after you create an individual structure having such a field; you can only access the field.

On the other hand, if you specify NIL (the default), the DEFINE-ALIEN-STRUCTURE macro generates access functions that are acceptable place indicators in a call to the SETF macro. That is, if you specify the keyword-value pair :READ-ONLY NIL in a data field (or omit the keyword altogether), you can write data in that field with the SETF form.

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 from this definiton.

```
(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))
```

3.4.4.3 **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 access function takes the field name as its argument, and the field occurs once. If you specify this keyword, the access 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 the LISP system evaluates the previous definition, the access functions AREA-1 and AREA-2 have the following formats:

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

3.4.4.4 **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 the :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. The offset defaults to the length of the field.

3.5 EXAMPLES OF ALIEN STRUCTURE DEFINITIONS

This section provides two examples of how to define an alien structure.

1.
```
Lisp> (DEFINE-ALIEN-STRUCTURE MY-ALIEN (FIELD-1 :TEXT 0 9))
MY-ALIEN
```

   This form defines an alien structure named MY-ALIEN, which contains one field named FIELD-1. The structure is a string that begins on the first byte and is 10 characters long.

2. The following example shows a Pascal record structure definition:

```
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}
```

An equivalent LISP record structure definition:

```
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
```

This form defines an alien structure named FAMILY-REC that has 66 fields indicating the members of a family, their ages, and the sex of the children. The definition contains the :DEFAULT, :SELECTION, :OCCURS, and :OFFSET keywords.

The fields that repeat are the CHILD-NAME, the CHILD-AGE, and the CHILD-SEX fields since 20 children's names are possible in this 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. A gap comes between each occurrence of the CHILD-NAME field since the field contains 20 bytes but repeats itself every 25 bytes. This gap is filled by the CHILD-AGE and CHILD-SEX fields.

Figure 3-1 illustrates how storage is internally allocated for the preceding FAMILY-REC alien structure. Only the first part of the alien structure is shown since 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.

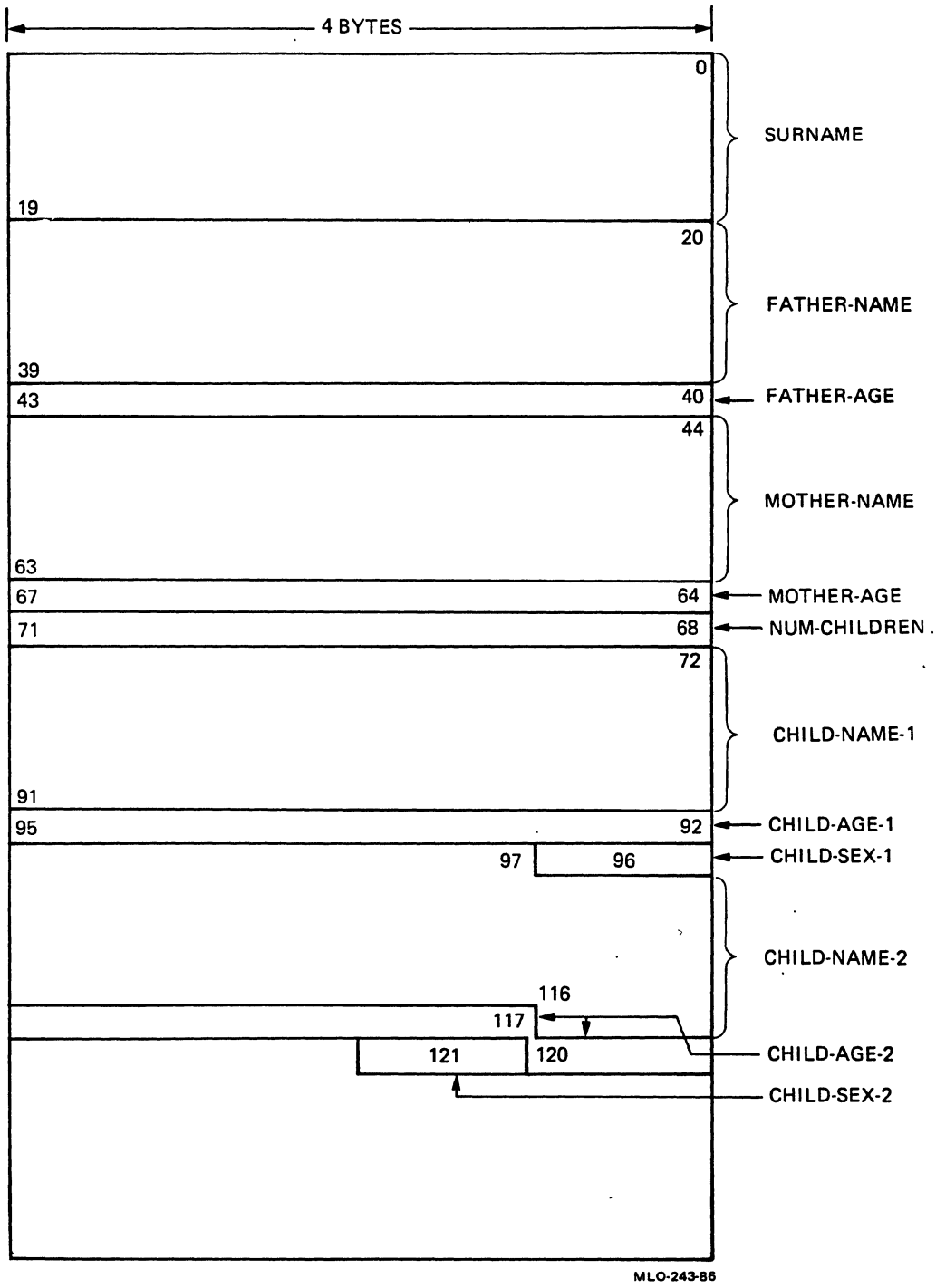


Figure 3-1: Internal Storage of FAMILY-REC

3.6 CREATING AN ALIEN STRUCTURE

Once you have defined an alien structure data type, you can create individual alien structures of that data type. To do so, specify a call to the constructor function of the data type you want (see Section 3.2). For example, in the following expression, the SETF macro gives the symbol EXAMPLE-1 a value of the alien structure of type SPACE, created when the LISP system evaluates the form (MAKE-SPACE).

```
(SETF EXAMPLE-1 (MAKE-SPACE))
```

Constructor functions accept two types of optional keywords:

- Keywords for initializing data fields
- Keywords affecting allocation of alien structures

3.6.1 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 named :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 3.4.4.1) 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
```

A data initialization you make 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 3.4.4.2). For example, the field AREA-1 is set to 28 in the following SETF form:

```
Lisp> (SETF (SPACE-AREA-1 EXAMPLE-1) 28)
28
```

3.6.2 Allocating Memory

In addition to the keywords defined by the data fields, all constructor functions also accept two keywords that affect data allocation, :ALIEN-DATA-LENGTH and :ALLOCATION.

Table 3-2: Values Used with Memory-Space Keywords

Keyword	Value
:ALIEN-DATA-LENGTH *integer*	The number of bytes of memory to be allocated for the alien structure's data vector. This keyword allows efficient use of storage when you are using alien structures as data buffers for variable size records. The default is 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 description of the ALIEN-STRUCTURE-LENGTH function in Part II for an example of default byte allocations.

Table 3-2 (cont.)

Keyword	Value
:ALLOCATION *value*	The type of allocation to be used for the alien structure. 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 a garbage collection (see the *VAX LISP/VMS User's Guide*).

3.7 ADDITIONAL ALIEN STRUCTURE MACRO AND FUNCTIONS

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

- DEFINE-ALIEN-FIELD-TYPE macro -- Defines alien structure field types.
- ALIEN-STRUCTURE-LENGTH function -- Returns the length of an alien structure.
- ALIEN-FIELD function -- Can be used to access arbitrary fields in an alien structure.

Descriptions of the macro and functions are in Part II.

CHAPTER 4

INTERRUPT FUNCTIONS

VAX LISP provides a mechanism whereby events that occur asynchronously -- for example, the completion of I/O, the expiration of a timer, or movement of a workstation pointing device -- can be made to interrupt the normal flow of LISP program execution. This mechanism allows you to specify a function to be executed when a particular asynchronous event occurs. Such functions are called interrupt functions.

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

- Section 4.1 provides an overview of the use of interrupt functions.
- Section 4.2 defines an asynchronous event and shows how the VMS operating system, through the AST mechanism, lets you request notification of asynchronous events.
- Section 4.3 shows how you establish an interrupt function in LISP and specify that it be executed as the result of a particular asynchronous event.

Chapter 5 contains information on three topics that relate to interrupt functions:

- Establishing priority levels at which interrupt functions operate
- Protecting sections of code against interruption
- Synchronizing the execution of interrupt functions using the WAIT function

4.1 OVERVIEW OF INTERRUPT FUNCTIONS

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

- Decide what asynchronous event is to trigger the function, what the function is to do, and what information the function requires. Section 4.2 provides information about various sources of asynchronous events.

- Define the function as you would any LISP function.

- 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 4.3 shows how to use INSTATE-INTERRUPT-FUNCTION.

- Use the call-out facility to define and call a system routine that causes an AST; or call one of the VAX LISP-supplied functions that establishes a response to an asynchronous event such as pointer movement. When using CALL-OUT, supply the *iif-id* of your interrupt function as the *astprm* argument to the system routine. When calling one of the VAX LISP functions, give the *iif-id* as the *action* argument. Section 4.3 provides examples of using both system routines and VAX LISP functions.

- When the specified asynchronous event occurs, your interrupt function executes. If the asynchronous event can occur more than once, your interrupt function executes each time the asynchronous event occurs.

- If you need to synchronize your program with the execution of an interrupt function, use the WAIT function. Chapter 5 describes this function.

- 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.

4.2 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 follow:

- 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 to predict what code will be executing is impossible.

- The user of an application moves a workstation pointer and clicks one of the pointer buttons. The movement of the pointer and the button click are asynchronous events.

The rest of this section describes how the VMS operating system provides access to asynchronous events. As a LISP programmer, knowledge of this mechanism will provide a useful background. For more information, see the *VAX/VMS System Services Reference Manual*.

4.2.1 ASTs

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

A variety of system routines can cause an asynchronous event. These routines are primarily those related to time and to I/O, and, if you are using a workstation, to pointer events. When you use these routines, you can specify an AST service routine. When the activity started by the system routine completes, the system causes an AST, and the AST service routine you specified executes, interrupting the normal flow of program execution. When the AST service routine completes, program execution resumes where it was interrupted.

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; 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 will continue until they are halted by another call to the system routine.

4.2.2 Routines that Cause ASTs

Ultimately, all ASTs are caused by system routines. You can make direct use of these system routines. Also, a number of VAX LISP functions call system routines that cause ASTs.

4.2.2.1 **System Routines** - A number of VMS system services and other system routines can cause ASTs. The documentation for these routines notes when a routine completes asynchronously. You can also tell when a routine can cause an AST by the presence of the following two arguments in the routine's 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 make use of system routines that declare ASTs, you must use the call-out facility, just as with any other external routine. Section 4.3 describes how to call these system routines.

4.2.2.2 **VAX LISP Routines** - A number of VAX LISP functions establish actions to be taken when specified asynchronous events occur. Currently, most of these functions are part of VAX LISP's support of the VAXstation. Functions are provided that 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.

4.2.2.3 **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 User's Guide* 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.

4.3 ESTABLISHING LISP INTERRUPT FUNCTIONS

This section details the steps you must 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. The INSTATE-INTERRUPT-FUNCTION function informs LISP that a particular function may be invoked asynchronously at a future time. Section 4.3.1 shows how to use INSTATE-INTERRUPT-FUNCTION.

- Once a function has been made known as an interrupt function, you must associate the interrupt function with one or more asynchronous events. Section 4.3.2 describes two ways of doing so.

- When an interrupt function is no longer needed, you should remove it from LISP's table of interrupt functions to conserve system resources. See Section 4.3.3.

- 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. See Section 4.3.4.

4.3.1 Defining an Interrupt Function

To define an interrupt function, you first define the function in the normal LISP manner. The function may take arguments, since INSTATE-INTERRUPT-FUNCTION can specify arguments to pass to an interrupt function.

Next, you use the INSTATE-INTERRUPT-FUNCTION function to make your function known to LISP as an interrupt function. 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 4.3.2 shows how the *iif-id* is used to associate 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 having 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.

4.3.1.1 **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 caused it to be invoked.

- 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.

4.3.1.2 **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 5 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.

4.3.1.3 **Automatic Removal of Interrupt Functions** - The INSTATE-INTERRUPT-FUNCTION function takes a keyword argument, :ONCE-ONLY-P, that allows you to 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 only execute 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 only be needed once. For interrupt functions that may execute more than once, remove them explicitly with the UNINSTATE-INTERRUPT-FUNCTION function after the last use. (Section 4.3.3 describes UNINSTATE-INTERRUPT-FUNCTION.)

4.3.2 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.

4.3.2.1 **Calling Out to System Routines** - System routines that can cause asynchronous events are characterized by the presence of two arguments, the *astadr* (address of the AST service routine) 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-supplied 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.

4.3.2.2 **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.) 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 4.3.1, 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 onwards, 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 only specify that 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.

4.3.3 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, you should 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 4.3.1.3.

4.3.4 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 maintain knowledge of its interrupt functions, so that you can reinstate them when the system is resumed. 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 5

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 can use these facilities to control a system in which multiple interrupt functions and keyboard functions might interfere with each other or with code that must execute as a unit.

This chapter discusses the following subjects:

- Section 5.1 describes the system of interrupt levels. You can specify an interrupt level with the BIND-KEYBOARD-FUNCTION and INSTATE-INTERRUPT-FUNCTION functions.
- Section 5.2 describes critical sections, which you use to prevent a section of code from being interrupted during execution.
- Section 5.3 describes the WAIT function, which you can use to suspend execution of your program until a keyboard function or interrupt function executes.

5.1 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, you should 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 interefere 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 interruptable 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 wish to debug, but less than 6. For example, if your application includes an interrupt function that executes at level 3, you should specify level 4 or 5 with BIND-KEYBOARD-FUNCTION to be able 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 a THROW to some tag, such as CANCEL-CHARACTER-TAG. Typically, you should 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.)

5.2 CRITICAL SECTIONS

A critical section consists of forms contained 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 in situations where 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. For this reason, you must be careful not to allow any infinite loops in a critical section. Should an infinite loop occur, you have no recourse but to terminate the LISP image.

You should 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.

5.3 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. The first is a reason for the wait, typically a string. The second is 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.

The testing function you specify in a call to the WAIT function can be any 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 unknown intervals.

- The dynamic state of LISP is not guaranteed during execution of the testing function. Therefore, the testing function cannot reely 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 types 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. In these cases, 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.

PART II

OBJECT DESCRIPTIONS

ALIEN-FIELD Function

Accesses the value of a field of a specified type from an alien structure. The function ignores the alien structure's predefined fields.

You can modify alien structures if you use the ALIEN-FIELD function with the SETF macro. This function is most useful for debugging a program that uses alien structures. The function can also be used to write your own accessing functions, for example, to access unnamed gaps in an alien structure.

For more information about alien structures, see Chapter 3.

Format

ALIEN-FIELD *alien-structure field-type start end*

Arguments

alien-structure

The alien structure from which a field value is to be accessed.

field-type

The type of the field from which a value is to be accessed. This argument can be either a keyword that names a built-in alien structure field type, a symbol (for a user-defined field type), or a list whose first element names the field type.

start

A rational number that specifies the start position (in 8-bit bytes) of a field in the alien structure's data area. This value is inclusive and zero-based. Default: none.

end

A rational number that specifies the end position (in 8-bit bytes) of a field in the alien structure's data area. This value is exclusive. Default: none.

Return Value

The value of a field of the specified alien structure.

ALIEN-FIELD Function (cont.)

Example

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
        (AREA-1 :UNSIGNED-INTEGER 0 4 :DEFAULT 22)
        (AREA-2 :UNSIGNED-INTEGER 4 8 :DEFAULT 2764))
SPACE
Lisp> (SETF SPACE-RECORD (MAKE-SPACE))
#<Alien Structure SPACE #x45FA60>
Lisp> (SPACE-AREA-1 SPACE-RECORD)
22
Lisp> (SPACE-AREA-2 SPACE-RECORD)
2764
Lisp> (ALIEN-FIELD SPACE-RECORD :UNSIGNED-INTEGER 0 4)
22
Lisp> (ALIEN-FIELD SPACE-RECORD :UNSIGNED-INTEGER 4 8)
2764
Lisp> (ALIEN-FIELD SPACE-RECORD :UNSIGNED-INTEGER 0 8)
11871289606166
```

This example illustrates:

- If you specify the ALIEN-FIELD function with the same field types and positions that are in the definition of an alien structure, the data you access is the same as if you had accessed it with that structure's default accessor functions.
- If you specify the ALIEN-FIELD function with different field types and positions from those in a defined alien structure, the data you access could be different depending on the field type and field positions you specify.

ALIEN-STRUCTURE-LENGTH Function

Returns the length of an alien structure in bytes.

Format

ALIEN-STRUCTURE-LENGTH *alien-structure*

Argument

alien-structure

The alien structure whose length is to be returned.

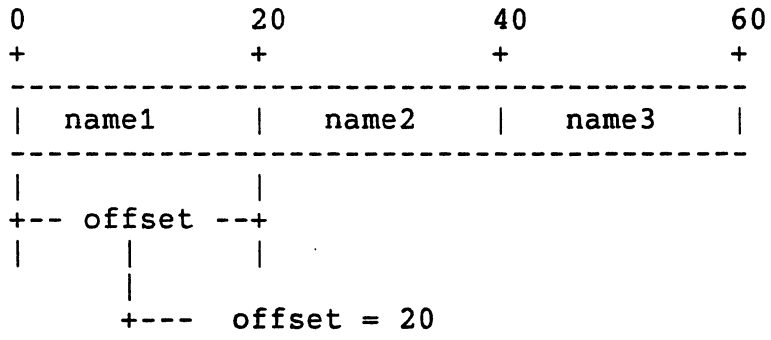
Return Value

The length of the alien structure in bytes

Example

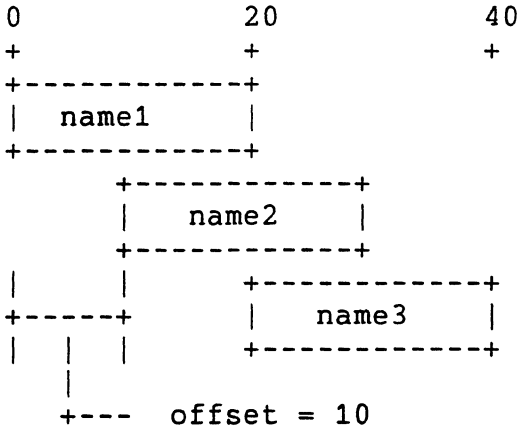
The following examples illustrate the use of the ALIEN-STRUCTURE-LENGTH macro. The diagram after each example illustrates why it returns a specific value.

```
1.  Lisp> (DEFINE-ALIEN-STRUCTURE EXAMPLE1
              (NAME :STRING 0 20 :OCCURS 3 :OFFSET 20))
    EXAMPLE1
    Lisp> (ALIEN-STRUCTURE-LENGTH (MAKE-EXAMPLE1))
    60
```



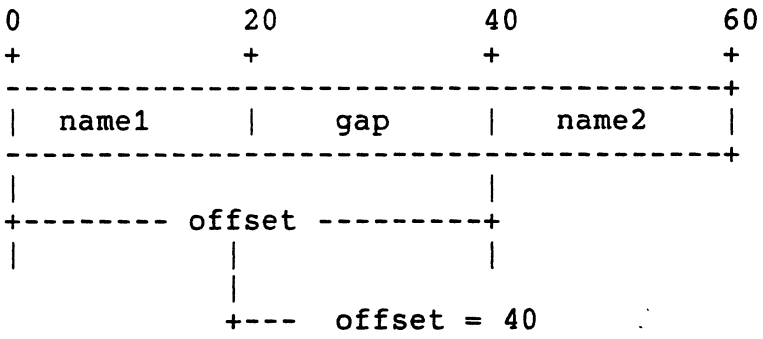
ALIEN-STRUCTURE-LENGTH Function (cont.)

2.
```
Lisp> (DEFINE-ALIEN-STRUCTURE EXAMPLE2
        (NAME :STRING 0 20 :OCCURS 3 :OFFSET 10))
EXAMPLE2
Lisp> (ALIEN-STRUCTURE-LENGTH (MAKE-EXAMPLE2))
40
```



In EXAMPLE2, the offset overlaps so that the last part of the information stored in NAME1 becomes the first part of the information stored in NAME2 and so on.

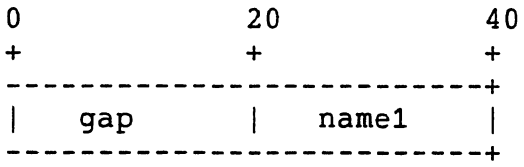
3.
```
Lisp> (DEFINE-ALIEN-STRUCTURE EXAMPLE3
        (NAME :STRING 0 20 :OCCURS 2 :OFFSET 40))
EXAMPLE3
Lisp> (ALIEN-STRUCTURE-LENGTH (MAKE-EXAMPLE3))
60
```



In EXAMPLE3 and EXAMPLE4, the gaps are counted as part of the length of the structure.

ALIEN-STRUCTURE-LENGTH Function (cont.)

4.
```
Lisp> (DEFINE-ALIEN-STRUCTURE EXAMPLE4 (NAME :STRING 20 40))
EXAMPLE4
Lisp> (ALIEN-STRUCTURE-LENGTH (MAKE-EXAMPLE4))
40
```



CALL-OUT Macro

Calls a defined external routine. If you specify an external routine that has not been defined with the DEFINE-EXTERNAL-ROUTINE macro, the LISP system signals an error.

For information about how to use the VAX LISP call-out facility, see Chapter 2.

Format

CALL-OUT *external-routine* &REST *arguments*

Arguments

external-routine

The name of a defined external routine.

arguments

Arguments to be passed to the external routine. The arguments correspond by position to the arguments defined for the routine. The LISP system evaluates the argument expressions before the external routine is called. You can omit an optional argument by specifying an expression whose value is NIL. The corresponding position in the argument list will contain a 0 to coincide with the VAX Procedure Calling Standard. If you specify fewer arguments than were specified in the definition, the argument list will contain only the number of arguments actually supplied. LISP signals an error if you supply more arguments than were specified in the definition.

Return Value

If the :RESULT option of the DEFINE-EXTERNAL-ROUTINE macro was specified, the external routine's result is returned. Otherwise, no value is returned.

CALL-OUT Macro (cont.)

Example

```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SMG$CREATE-PASTEBOARD
                                  :FILE "SMGSHR"
                                  :RESULT INTEGER)
        (NEW-PASTEBOARD-ID :LISP-TYPE INTEGER
                           :VAX-TYPE :UNSIGNED-LONGWORD
                           :ACCESS :IN-OUT)
        (OUTPUT-DEVICE :LISP-TYPE STRING)
        (PB-ROWS :LISP-TYPE INTEGER :ACCESS :IN-OUT)
        (PB-COLUMNS :LISP-TYPE INTEGER :ACCESS :IN-OUT)
        (PRESERVE-SCREEN-FLAG :LISP-TYPE INTEGER
                              :VAX-TYPE :UNSIGNED-LONGWORD))
SMG$CREATE-PASTEBOARD
```

Defines the SMG screen management routine called SMG$CREATE-PASTEBOARD.

```
Lisp> (DEFVAR *PASTEBOARD-ID* -1)
*PASTEBOARD-ID*
```

Defines a special variable which will contain the pasteboard ID returned by the external routine.

```
Lisp> (CALL-OUT SMG$CREATE-PASTEBOARD *PASTEBOARD-ID*
                                      NIL NIL NIL 1)
1
```

Calls the external routine SMG$CREATE-PASTEBOARD, specifying the special variable to receive the pasteboard ID. Three arguments are omitted, and a preserve-screen-flag of 1 is given. The result status is returned.

COMMON-AST-ADDRESS Parameter

Specifies the address of a VAX LISP/VMS-supplied routine that initially handles all
ASTs. This parameter must be given as the *astadr* argument to all external routines that can cause an AST. No other object can be passed as the *astadr* argument. Use the :VALUE mechanism to pass this parameter.

Format

COMMON-AST-ADDRESS

Example

See the description of INSTATE-INTERRUPT-FUNCTION for an example of the use of COMMON-AST-ADDRESS.

CRITICAL-SECTION Macro

Executes the forms in its body as a "critical section." During the execution of a critical section, all interrupt functions are blocked and queued for later execution. CTRL/C is also blocked, so a critical section must neither loop nor cause errors. It is an error to perform I/O or to call WAIT in a critical section.

If an error shoud occur during a critical section, VAX LISP invokes the debugger, and temporarily removes the restrictions on interrupts so 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.

Format

CRITICAL-SECTION {*form*}*

Argument

form

Form(s) to be executed as a critical section.

Return Value

The value(s) of the last form executed.

Example

```
Lisp> (DEFUN RESTORE-TO-FREE-LIST (CONS-CELL)
        (CRITICAL-SECTION
          (SETF (CDR CONS-CELL) *HEAD-OF-FREE-LIST*
                *HEAD-OF-FREE-LIST* CONS-CELL)))
RESTORE-TO-FREE-LIST
```

This example defines a function that restores a cons cell to the head of a list of free cells. During the call to SETF, the list is in an inconsistent state because the special variable *HEAD-OF-FREE-LIST* does not point to the head of the list. An interrupting function that used *HEAD-OF-FREE-LIST* to remove an element from the list would break the list. Therefore, RESTORE-TO-FREE-LIST uses the CRITICAL-SECTION macro to ensure that the SETF call completes without interruption.

DEFINE-ALIEN-FIELD-TYPE Macro

Defines alien-structure field types.

For information about alien structures, see Chapter 3.

Format

DEFINE-ALIEN-FIELD-TYPE *name lisp-type primitive-type access-function setf-function*

Arguments

name

The name of the alien-field type being defined.

lisp-type

A LISP data type indicating the type of LISP object to which the field is to be mapped.

primitive-type

Either one of the predefined alien-field types or a type that was previously defined with the DEFINE-ALIEN-FIELD-TYPE macro. A LISP object of type *primitive-type* is extracted from the alien structure's data when the field is accessed. The object is then passed to the specified access function. Predefined alien-field types are listed in Table 3-1.

access-function

A function of one argument (whose type is *primitive-type*) that returns an object of type *lisp-type*.

setf-function

A function of one argument (whose type is *lisp-type*) that returns an object whose type is the type of the default SETF form, as defined by the *primitive-type* argument. When the object is returned, it is packed into the alien structure's field data.

Return Value

The name of the alien-field type.

NOTE

Functions that access and set field values can take more than one argument; additional arguments

DEFINE-ALIEN-FIELD-TYPE Macro (cont.)

are optional. When the type argument in the DEFINE-ALIEN-STRUCTURE macro's field description is a list, the first element of the list is the field type, and the remaining elements are expressions the LISP system evaluates when it evaluates the access function. The resulting values are passed as additional arguments to the functions that access or set the field.

Examples

1.
```
Lisp> (DEFINE-ALIEN-FIELD-TYPE INTEGER-STRING-8
        'INTEGER
        :STRING
        #'(LAMBDA
        (X)
        (PARSE-INTEGER X :JUNK-ALLOWED T))
        #'(LAMBDA
        (X)
        (FORMAT NIL "~S" X)))
INTEGER-STRING-8
Lisp> (DEFINE-ALIEN-STRUCTURE TWO-ASCII-INTEGERS
        (INT-1 INTEGER-STRING-8 0 8)
        (INT-2 INTEGER-STRING-8 8 16))
TWO-ASCII-INTEGERS
```

- The call to the DEFINE-ALIEN-FIELD-TYPE macro defines a field type named INTEGER-STRING-8. The field type INTEGER-STRING-8 causes an alien structure to convert strings to integers.

- The call to the DEFINE-ALIEN-STRUCTURE macro defines an alien structure named TWO-ASCII-INTEGERS that has two fields, each of type INTEGER-STRING-8.

DEFINE-ALIEN-FIELD-TYPE Macro (cont.)

2.
```
Lisp> (DEFINE-ALIEN-FIELD-TYPE SELECTION
        T
        :UNSIGNED-INTEGER
        #'(LAMBDA
        (N)
        (NTH N '(MA RI NY)))
        #'(LAMBDA
        (X)
        (POSITION X '(MA RI NY))))
SELECTION
```

This is an example of how the :SELECTION type could be implemented. The example defines an alien-field type named SELECTION. This type defines a relationship between unsigned integers in an alien field and LISP data objects. In accessing the value of a field of this type, the access-function uses the integer stored in the alien field as an index into a list. In setting the value in this type of field, the position of a LISP object in that list is used to define the integer value stored in the alien structure.

DEFINE-ALIEN-STRUCTURE Macro

Defines alien structures. An alien structure is a collection of bytes containing VAX data types.

The syntax of the DEFINE-ALIEN-STRUCTURE macro is similar to the DEFSTRUCT macro described in *COMMON LISP: The Language*.

For an explanation of how to define an alien structure, see Chapter 3.

Format

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

Arguments

name-and-options

The *name-and-options* argument is the name and the options of a new LISP data type. The name argument must be a symbol. The options define the characteristics of the alien structure. If you do not specify options, you can specify the name-and-options argument as a symbol:

name

If you specify options, specify the name-and-options argument as a list whose first element is the name:

(*name* {(*keyword value*)}*)

Using the following format, specify options as a list of keyword-value pairs.

(*keyword value*)

Table 1 lists the keyword-value pairs that you can specify.

Table 1: DEFINE-ALIEN-STRUCTURE Options

Keyword-Value Pair	Description
:CONC-NAME *name*	Names the access functions. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes a prefix in the access function names. If you wish to include a hyphen (-) in the access function names,

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Table 1 (cont.)

Keyword-Value Pair	Description
	specify it as part of the prefix. If you specify NIL, the access function names are the same as the field names. By default, the prefix is the alien structure name followed by a hyphen.
:CONSTRUCTOR *name*	Names the constructor function. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes the name of the constructor function. If you specify NIL, the macro does not define a constructor function. If you do not specify this keyword, the constructor function's name is the prefix MAKE- attached to the alien structure name.
:COPIER *name*	Names the copier function. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes the name of the copier function. If you specify NIL, the macro does not create a copier function. If you do not specify this keyword, the copier function's name is the prefix COPY- attached to the alien structure name.
:PREDICATE *name*	Names the predicate function. The value can be a symbol or NIL. If you specify a symbol, the symbol becomes the name of the predicate function. If you specify NIL, the macro does not define a predicate function. If you do not specify this keyword, the macro names the predicate function by attaching the structure name to the characters -P.

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Table 1 (cont.)

Keyword-Value Pair	Description
:PRINT-FUNCTION *function-name*	Specifies the print function for the alien structure. The value must be a function. If you do not specify this keyword, the LISP system displays the alien structure in the following format: #<Alien Structure *name number*> In the preceding format, *name* is the name of the alien structure and *number* is a unique identification number, which distinguishes alien structures that have the same name.

doc-string

The documentation string to be attached to the symbol that names the alien structure. The documentation string is of type STRUCTURE. See *COMMON LISP: The Language* for information on the DOCUMENTATION function.

field-description

A field description for the alien structure. Specify a field description in the following format:

(*name type start end options*)

The *name* argument must be a symbol. It is used to name functions that access and set the value of the alien structure field.

DEFINE-ALIEN-STRUCTURE Macro (cont.)

The *type* argument determines the method by which the VAX data type stored in a field is converted to a LISP object and vice versa. Valid types are:

:STRING	:VARYING-STRING
:SIGNED-INTEGER	:UNSIGNED-INTEGER
:BIT-VECTOR	:F-FLOATING
:G-FLOATING	:D-FLOATING
:H-FLOATING	:POINTER
:SELECTION	Types defined with the VAX LISP DEFINE-ALIEN-FIELD-TYPE macro

See Chapter 3 for more information on field types.

As in COMMON LISP, the *start* and *end* arguments are zero-based with *start* being inclusive and *end* being exclusive.

The *start* argument must be a rational number or, in some cases, a fixnum (see Section 3.4.3.1) that specifies the 8-bit byte start position of the field in the alien structure's data area. Default: none. See Chapter 3 for more information on field start positions.

The *end* argument must be a rational number or, in some cases, a fixnum (see Section 3.4.3.1) that specifies the 8-bit byte end position of the field in the alien structure's data area. The last position a field occupies is the position that precedes the field's end position value. Default: none. See Chapter 3 for more information on field end positions.

The *options* define the characteristics for the field. Specify each option with a keyword-value pair:

keyword value

Table 2 lists the keyword-value pairs that you can specify.

Table 2: DEFINE-ALIEN-STRUCTURE Field Options

Keyword-Value Pair	Description
:DEFAULT *form*	Specifies the default initial value that is to occupy the field. If the field's initial value was not specified in a call to the alien structure's constructor function, the form is evaluated when the constructor function is called.

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Table 2 (cont.)

Keyword-Value Pair	Description
	The value that results from the evaluation is the field's default initial value. This value defaults to NIL.
:READ-ONLY *value*	Specifies whether the field can be accessed or set. The value can be T or NIL. If you specify T, the macro generates access functions that are not acceptable place indicators in a call to the SETF macro. If you specify NIL, the macro generates access functions that are acceptable place indicators in a call to the SETF macro. The default is NIL.
:OCCURS *integer*	Specifies the number of times the field is to be represented within the alien structure. The value must be an integer. The default value is 1 (which means no repeats).
:OFFSET *rational-number*	Specifies the distance in 8-bit bytes from the start of one occurrence of the field to the start of the next occurrence of the field. The value must be a rational number. If you specify a value that is greater than the field's length, the alien structure contains gaps. You can access the gaps with other field definitions.

Return Value

The name of the alien structure.

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Example

```
Lisp> (DEFINE-ALIEN-STRUCTURE ET
        (SPACE-SHIP     :STRING 0 10)
        (PHONE-NUMBER   :UNSIGNED-INTEGER 10 17)
        (HOME           :STRING 17 32))
ET
```

Defines an alien structure named ET, which contains three fields named SPACE-SHIP, PHONE-NUMBER, and HOME. The fields SPACE-SHIP and HOME are defined to be strings of length 10 and 15 respectively. The field PHONE-NUMBER is defined to be an unsigned integer seven bytes long.

More examples of how to define alien structures are provided in Chapter 3.

DEFINE-EXTERNAL-ROUTINE Macro

Defines an external routine that a LISP program is to call. You can call routines defined with this macro with the VAX LISP CALL-OUT macro. For information about how to use the VAX LISP call-out facility, see Chapter 2.

Format

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

Arguments

name-and-options

The name argument is the name of the external routine being defined. The name argument must be a symbol. The options define the characteristics of the name argument. If you do not specify options, you can specify the name-and-options argument as a symbol:

name

If you specify options, specify the name-and-options argument as a list whose first element is the name:

(*name* {*keyword value*}*)

Specify the options with keyword-value pairs:

keyword value

The option values are not evaluated.

Table 3 lists the keyword-value pairs that you can specify.

Table 3: DEFINE-EXTERNAL-ROUTINE Options

Keyword-Value Pair	Description
:CHECK-STATUS-RETURN *value*	Specifies whether the call-out facility is to check the severity of the value that an external routine returns in register R0. The value you specify can be T, an integer, or NIL. If you specify T, the call-out facility checks the severity of the return value.

DEFINE-EXTERNAL-ROUTINE Macro (cont.)

Table 3 (cont.)

Keyword-Value Pair	Description
	If the severity is warning, error, or severe, the LISP system signals a continuable error. If you specify an integer, an error is signaled if that value is returned by the routine. If you specify NIL, the call-out facility does not check the severity of the return value. NIL is the default value. If you specify this option, do not specify the :RESULT option.
:ENTRY-POINT *string*	Names the external routine's entry point. The value must be a string. The macro converts the name to uppercase characters. The default value is the print name of the external routine name.
:FILE *pathname*	Specifies the shareable image that was created for the external routine. This must be in upper case and must be a logical name, or the name of an executable image in the SYS$SHARE directory. The file specification is merged with the file SYS$SHARE:.EXE. You must specify this option unless you are calling a system service.
:RESULT *type*	Specifies the type of LISP object the external routine is to return. The value can be a LISP type, a type-spec-list, or NIL. A type-spec-list has the following format: :RESULT (:LISP-TYPE *LISP-type* :VAX-TYPE *VAX-type*) See Table 2-4 for a list of LISP/VAX types. NIL specifies

DEFINE-EXTERNAL-ROUTINE Macro (cont.)

Table 3 (cont.)

Keyword-Value Pair	Description
	that the routine returns no value. The default value is NIL. If you specify this option, do not specify the :CHECK-STATUS-RETURN option.
:TYPE-CHECK *value*	Specifies whether the call-out facility is to check the types of the arguments passed to the external routine for compatibility with the LISP types specified in the argument specification. The value can be T or NIL. If you specify T, the facility checks the types for compatibility; if you specify NIL, the facility does not check the argument types. The default value is NIL.

doc-string

The documentation string for the symbol that names the external routine. The documentation string is of type EXTERNAL-ROUTINE. See *COMMON LISP: The Language* for information on the DOCUMENTATION function.

argument-description

An argument description that is to be passed to the external routine. Include as many descriptions as the arguments you want to call out to. Specify the descriptions in the following format:

(*name options*)

The *name* argument must be a unique symbol in the definition or NIL. The name identifies the argument and is used in some error messages. If you do not specify options, you can specify the argument-description argument as a symbol:

name

DEFINE-EXTERNAL-ROUTINE Macro (cont.)

If you specify options, specify the argument as a list whose first element is the name:

(name {keyword value})*

The *options* arguments define the characteristics of an argument. Specify the options with keyword-value pairs:

keyword value

The option values are not evaluated.

Table 4 lists the keyword-value pairs you can specify.

Table 4: DEFINE-EXTERNAL-ROUTINE Argument Options

Keyword-Value Pair	Description
:ACCESS *value*	Specifies the type of access the external routine needs for the argument. The value can be either :IN or :IN-OUT. The default value is :IN. If you specify :IN, the argument can be read, but not modified by the external routine. If you specify :IN-OUT, the argument can be both read and destructively modified by the external routine.
:LISP-TYPE *type*	Specifies the LISP type of the argument value the call-out facility is to pass to the external routine. See Table 2-4 and Table 2-5 for the values you can specify.
:MECHANISM *value*	Specifies the argument-passing mechanism the external routine is to expect for the argument. The values you can specify are :VALUE, :REFERENCE, and :DESCRIPTOR. The default value is :DESCRIPTOR for VAX-TYPE :TEXT and :REFERENCE for other LISP data types.

DEFINE-EXTERNAL-ROUTINE Macro (cont.)

Table 4 (cont.)

Keyword-Value Pair	Description
:VAX-TYPE *type*	Specifies the VAX data type of the argument value the external routine is to return. See Table 2-4 and Table 2-5 for the values you can specify.

Return Value

The symbol that names the external routine.

Example

```
Lisp> (DEFINE-EXTERNAL-ROUTINE (MTH$ACOSD
                                :FILE "MTHRTL"
                                :RESULT (:LISP-TYPE
                                            SINGLE-FLOAT
                                         :VAX-TYPE
                                            :F-FLOATING))
        "This routine returns the arc cosine
         of an angle in degrees."
        (X :LISP-TYPE SINGLE-FLOAT
           :VAX-TYPE :F-FLOATING))
MTH$ACOSD
```

Defines an RTL routine, called MTH$ACOSD, which returns the arc cosine of an angle in degrees. The routine takes one read-only argument, which is a F_floating number, and returns the result as a F_floating number.

More examples of how to define external routines are provided in Chapter 2. These examples also show you how to call out to defined external routines.

FORCE-INTERRUPT-FUNCTION Function

Forces an AST and thus the invocation of the related interrupt function specified by its argument. FORCE-INTERRUPT-FUNCTION is primarily useful for debugging.

Format

FORCE-INTERRUPT-FUNCTION *iif-id*

Argument

iif-id

An interrupt function identifier previously returned by INSTATE-INTERRUPT-FUNCTION.

Return Value

Undefined.

Example

```
Lisp> (DEFUN TIMER-INTERRUPT-HANDLER ()
        (PRINC "The timer has expired"))
TIMER-INTERRUPT-HANDLER
Lisp> (SETF TIMER-IIF (INSTATE-INTERRUPT-FUNCTION
                        'TIMER-INTERRUPT-HANDLER))
8454198
Lisp> (FORCE-INTERRUPT-FUNCTION TIMER-IIF)
The timer has expired
T
```

- The function TIMER-INTERRUPT-HANDLER is instated as an interrupt function whose *iif-id* is retained as the value of TIMER-IIF
- Passing TIMER-IIF in a call to FORCE-INTERRUPT-FUNCTION causes TIMER-INTERRUPT-HANDLER to execute.

GET-INTERRUPT-FUNCTION Function

Returns information about the interrupt function specified by its argument.

Format

GET-INTERRUPT-FUNCTION *iif-id*

Argument

iif-id

An interrupt function identifier previously returned by INSTATE-INTERRUPT-FUNCTION.

Return Value

Four values:

1. The function definition of the interrupt function
2. The argument list
3. The value of :LEVEL (an integer in the range 0 through 7)
4. The value of :ONCE-ONLY-P (T or NIL)

If the interrupt function represented by *iif-id* has been uninstated, GET-INTERRUPT-FUNCTION returns four values of NIL.

Example

```
Lisp> (DEFUN TIME-ELAPSED (N)
         (FORMAT T
                 "~@(~R~) second~:P ~:*~[have~;has~:;have~] ~
                  elapsed since setting the timer"
                 N))
Lisp> (SETF T-E-IIF (INSTATE-INTERRUPT-FUNCTION
                      #'TIME-ELAPSED
                      :ARGUMENTS (LIST 5)))
8388671
Lisp> (GET-INTERRUPT-FUNCTION T-E-IIF)
(LAMBDA (N) (BLOCK TIME-ELAPSED (FORMAT T "~@(~R~) second~:P
~:*~[have~;has~:;have~] ~
                   elapsed since setting the timer" N))) ;
(5) ;
2 ;
NIL
```

- The function TIME-ELAPSED, which prints out the number of seconds since a timer was set, is defined. It takes a single argument.

GET-INTERRUPT-FUNCTION Function (cont.)

- TIME-ELAPSED is instated as an interrupt function. The :ARGUMENTS keyword specifies that TIME-ELAPSED is passed one argument, the number 5. The *iif-id* returned by INSTATE-INTERRUPT-FUNCTION is retained as the value of T-E-IIF.

- The call to GET-INTERRUPT-FUNCTION returns four values. The first value is the function definition of TIME-ELAPSED. The second value is a list of the arguments specified with INSTATE-INTERRUPT-FUNCTION. The third value is the interrupt level (2, the default for INSTATE-INTERRUPT-FUNCTION). The fourth value is NIL, indicating that :ONCE-ONLY-P was not specified with INSTATE-INTERRUPT-FUNCTION.

INSTATE-INTERRUPT-FUNCTION Function

Takes as its first argument a function that will later be invoked asynchronously and returns an identification (*iif-id*) for this instance of the function.
The *iif-id* is intended to be passed to a routine that can cause an AST. The AST, when it occurs, invokes the interrupt function identified by the *iif-id*.

The :ARGUMENTS keyword allows you to supply a list of zero or more arguments. The arguments will be passed to the interrupt function when it executes. This allows a single function to take different actions depending on the particular AST that invokes it.

The :LEVEL keyword lets you specify the interrupt level for the interrupt function as an integer in the range 0-7. See Chapter 5 for more information about interrupt levels.

The :ONCE-ONLY-P keyword allows you to specify that this instance of the function will only be invoked once, and then discarded. Specifying :ONCE-ONLY-P T is equivalent to using UNINSTATE-INTERRUPT-FUNCTION on the function after its first invocation. However, :ONCE-ONLY-P does not disable further occurrences of the AST after its first occurrence. If :ONCE-ONLY-P T is specified and the corresponding AST occurs more than once, the second and subsequent ASTs are ignored. (See UNINSTATE-INTERRUPT-FUNCTION for more details.)

For more information about interrupt functions, see Chapter 4.

Format

```
INSTATE-INTERRUPT-FUNCTION function
     &KEY :ARGUMENTS :LEVEL :ONCE-ONLY-P
```

Arguments

function

A function to be invoked asynchronously at a later time.

:ARGUMENTS

A list of zero or more arguments to be passed to the interrupt function when it is invoked.

:LEVEL

An integer in the range 0-7, specifying the interrupt level for the interrupt function. The default interrupt level is 2.

INSTATE-INTERRUPT-FUNCTION Function (cont.)

:ONCE-ONLY-P

T or NIL (the default), specifying whether or not this instance of the function is to be uninstated when it has been invoked once.

Return Value

An integer that identifies this instance of the interrupt function. This integer becomes the *iif-id* argument to functions that require an *iif-id* and the *astprm* argument to external routines that can cause an AST.

Examples

1.
```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SYS$SETIMR
                                :CHECK-STATUS-RETURN T)
        (EFN :MECHANISM :VALUE)
        (DAYTIM :VAX-TYPE :QUADWORD)
        (ASTADR :MECHANISM :VALUE)
        (ASTPRM :MECHANISM :VALUE))
SYS$SETIMR
Lisp> (DEFINE-EXTERNAL-ROUTINE (SYS$BINTIM
                                :CHECK-STATUS-RETURN T)
        (TIMBUF :VAX-TYPE :TEXT :LISP-TYPE STRING)
        (TIMADR :VAX-TYPE :QUADWORD :ACCESS :IN-OUT))
SYS$BINTIM
Lisp> (DEFUN SET-TIMER (DELTA-TIME)
        (LET ((IIF-ID (INSTATE-INTERRUPT-FUNCTION
                        #'TIMER-INTERRUPT-HANDLER
                        :ONCE-ONLY-P T)))
          (CALL-OUT SYS$SETIMR NIL DELTA-TIME
                                COMMON-AST-ADDRESS IIF-ID))
        T)
SET-TIMER
Lisp> (DEFUN TIMER-INTERRUPT-HANDLER ()
        (PRINT "The timer has expired"))
TIMER-INTERRUPT-HANDLER
Lisp> (SETQ DELTA 0)  ; DELTA must be bound before CALL-OUT
0
Lisp> (CALL-OUT SYS$BINTIM "0 ::5" DELTA)
1
Lisp> (SET-TIMER DELTA)
T
Lisp> (five seconds pass) "The timer has expired"
```

- The external routine SYS$SETIMR is defined. SYS$SETIMR is a system service that sets a timer and causes an AST when the timer expires. The ASTADR and ASTPRM arguments are both passed with :MECHANISM :VALUE.

INSTATE-INTERRUPT-FUNCTION Function (cont.)

- The external routine SYS$BINTIM is defined. SYS$BINTIM is a system service that converts a time specified as a string to a binary format acceptable to SYS$SETIMR.

- The function SET-TIMER is defined. SET-TIMER's argument is the binary-formatted time before a timer should expire. SET-TIMER calls INSTATE-INTERRUPT-FUNCTION to instate TIMER-INTERRUPT-HANDLER as an interrupt function. The T value for :ONCE-ONLY-P requests that the interrupt function be uninstated after it executes once. SET-TIMER then calls out to SYS$SETIMR, passing the binary time as the second argument. The third argument is (and must be) the COMMON-AST-ADDRESS parameter; the fourth argument is the *iif-id* returned by INSTATE-INTERRUPT-FUNCTION.

- The function TIMER-INTERRUPT-HANDLER is defined. It simply prints a message on the terminal.

- After the binary format for 5 seconds is stored in DELTA, the call to SET-TIMER sets a timer to expire in 5 seconds. SET-TIMER returns. Five seconds later, the timer expires and the interrupt function TIMER-INTERRUPT-HANDLER executes, printing the message.

2. The following example shows the use of arguments with interrupt functions. The external routines SYS$SETIMR and SYS$BINTIM have the same definitions as shown in Example 1.

```
Lisp> (DEFUN SET-TIMER (SECONDS)
        (LET ((DELTA 0)
              (IIF (INSTATE-INTERRUPT-FUNCTION
                       #'TIME-ELAPSED
                       :ONCE-ONLY-P T
                       :ARGUMENTS (LIST SECONDS))))
          (CALL-OUT SYS$BINTIM (TIME-STRING SECONDS) DELTA)
          (CALL-OUT SYS$SETIMR NIL DELTA
                                COMMON-AST-ADDRESS IIF))
        T)
SET-TIMER
Lisp> (DEFUN TIME-STRING (N)
        (FORMAT NIL "0 :~D:~D" (TRUNCATE N 60) (MOD N 60)))
TIME-STRING
Lisp> (DEFUN TIME-ELAPSED (N)
        (FORMAT T
                "~@(~R~) second~:P ~:*~[have~;has~:;have~] ~
                   elapsed since setting the timer"
                N))
TIME-ELAPSED
```

INSTATE-INTERRUPT-FUNCTION Function (cont.)

```
Lisp> (SET-TIMER 5)
T
Lisp> (five seconds elapse) Five seconds have elapsed since
setting the timer
```

- The new definition of SET-TIMER accepts an integer argument that is the number of seconds to wait (not a binary-formatted time). SET-TIMER instates a function called TIME-ELAPSED as an interrupt function, requesting that one argument (the number of seconds) be passed to TIME-ELAPSED. SET-TIMER then calls out to SYS$BINTIM to convert the seconds to binary format. (An auxiliary function, TIME-STRING, converts the integer argument to a string acceptable to SYS$BINTIM. TIME-STRING cannot format an argument larger than 3599 seconds properly.) Finally, SET-TIMER calls out to SYS$SETIMR, passing the binary-formatted time (the second argument) and the *iif-id* for TIME-ELAPSED (the fourth argument).

- The function TIME-ELAPSED is defined. It accepts an integer argument and uses FORMAT to print the number of seconds represented by that argument.

- SET-TIMER is called with the argument 5. SET-TIMER returns. After 5 seconds elapse, TIME-ELAPSED executes and prints the formatted message on the terminal, including the number of seconds.

3. The following example shows the use of an interrupt function with a VAX LISP-supplied function. This example works only on a VAXstation.

```
Lisp> (DEFUN PRINT-BUTTON (BUTTON TRANSITION)
        (WHEN TRANSITION
          (CASE BUTTON
            (#.UIS:POINTER-BUTTON-1
                (PRINC "Left button pressed"))
            (#.UIS:POINTER-BUTTON-2
                (PRINC "Middle button pressed"))
            (#.UIS:POINTER-BUTTON-3
                (PRINC "Right button pressed")))))
PRINT-BUTTON
Lisp> (SETF BUTTON-IIF
            (INSTATE-INTERRUPT-FUNCTION #'PRINT-BUTTON))
8454171
Lisp> (UIS:SET-BUTTON-ACTION DISPLAY WINDOW BUTTON-IIF)
T
Lisp> (Left button is pressed) Left button pressed
```

INSTATE-INTERRUPT-FUNCTION Function (cont.)

- The function PRINT-BUTTON is defined. Depending on its arguments, it prints one of three lines on the terminal, or it does nothing.

- PRINT-BUTTONS is instated as an interrupt function. The *iif-id* returned by INSTATE-INTERRUPT-FUNCTION is retained as the value of BUTTON-IIF.

- The function SET-BUTTON-ACTION is called with BUTTON-IIF as the third argument. SET-BUTTON-ACTION specifies what should happen when a workstation pointer button is pressed or released while the pointer cursor is in a specified window. If an *iif-id* is passed as the third argument, the associated interrupt function is invoked when a button is pressed or released. SET-BUTTON-ACTION causes an interrupt function to be passed two arguments: the button involved, and T or NIL to indicate whether the button was pressed or released.

- After SET-BUTTON-ACTION returns, a button is pressed. PRINT-BUTTON receives the two arguments passed to it, and prints the message on the screen.

UNINSTATE-INTERRUPT-FUNCTION Function

Informs LISP that the interrupt function identified by *iif-id* will no longer be used. The *iif-id* can no longer be given as the *astprm* argument to a routine that can cause an AST. However, UNINSTATE-INTERRUPT-FUNCTION does not prevent a routine from causing an AST with that *iif-id*. For example, an external routine that was called with that *iif-id* before the use of UNINSTATE-INTERRUPT-FUNCTION might later cause an AST. VAX LISP ignores ASTs for which the corresponding interrupt function has been uninstated.

Format

UNINSTATE-INTERRUPT-FUNCTION *iif-id*

Argument

iif-id

An interrupt function identifier previously returned by INSTATE-INTERRUPT-FUNCTION.

Return Value

Undefined

Examples

1.
```
Lisp> (UNINSTATE-INTERRUPT-FUNCTION TIMER-IIF)
T
```

   Makes the interrupt function represented by TIMER-IIF unavailable for future use.

2.
```
  .
  .
(LET ((BUTTON-IIF (INSTATE-INTERRUPT-FUNCTION
                    #'BUTTON-HANDLER)))
  (UIS:SET-BUTTON-ACTION DISPLAY WINDOW BUTTON-IIF)
    .
    .
  (UIS:SET-BUTTON-ACTION DISPLAY WINDOW NIL)
  (UNINSTATE-INTERRUPT-FUNCTION BUTTON-IIF))
```

   In this code fragment, the interrupt function BUTTON-HANDLER is instated and BUTTON-IIF is bound to its *iif-id*. The first call to SET-BUTTON-ACTION establishes BUTTON-HANDLER as the function to execute when a workstation pointer button is pressed or released. Later, the second call to SET-BUTTON-ACTION requests that no action be taken when buttons are pressed or released. Finally, UNINSTATE-INTERRUPT-FUNCTION removes the interrupt function represented by BUTTON-IIF from the system.

WAIT Function

Causes the program that calls it to stop executing until a specified function returns non-NIL. The first argument to WAIT is a reason for waiting, typically a string. The second argument is a function; arguments to the function can be provided as additional arguments to WAIT.

A program that calls the WAIT function stops executing. The function specified in WAIT's second argument is called periodically with the arguments provided in the WAIT call. If the function returns NIL, the program continues to wait. When the function returns non-NIL, WAIT returns an undefined value, and program execution continues.

The testing function you specify with WAIT does not execute in the context of the program that issued the WAIT. Therefore, the testing function cannot depend on the binding of special variables. You should pass the testing function some data structure, such as a cons cell, structure, or array. Pass the same data structure to an interrupt function that modifies the data structure. Chapter 5 contains examples of this technique.

For efficiency and reliability, ensure that the testing function executes quickly and does not cause errors. If the testing function encounters an error while LISP is in a WAIT state, LISP is left in an inconsistent state and will have to be terminated. For this reason, WAIT calls its testing function once before entering the WAIT state. Errors that occur on this initial call can be debugged normally.

Format

WAIT *reason function* &REST *arguments*

Arguments

reason

The reason for the wait, typically a string.

function

A function that will be called periodically to determine if the program should continue to wait.

arguments

Arguments to be supplied to the function given in the second argument.

Return Value

Undefined.

WAIT Function (cont.)

Examples

1.
```
Lisp> (SETF *FLAG* (LIST NIL))
(NIL)
Lisp> (BIND-KEYBOARD-FUNCTION
        #\^F
        #'(LAMBDA () (SETF (CAR *FLAG*) T)))
Lisp> (WAIT "Wait for CTRL/F" #'CAR *FLAG*)
(After a pause, user types CTRL/F)
T
Lisp>
```

   - The special variable *FLAG* is set to a list whose only element is NIL.
   - CTRL/F is bound to a function that sets the first element of *FLAG* to T.
   - The call to the WAIT function specifies CAR as the testing function and *FLAG* as the argument to the testing function. WAIT does not return immediately.
   - When the user types CTRL/F, the keyboard function sets the first element of *FLAG* to T, the testing function returns T, and the call to WAIT returns.

2. The following example uses the definitions of the external routines SYS$SETIMR and SYS$BINTIM and the function TIME-STRING from Examples 1 and 2 under INSTATE-INTERRUPT-FUNCTION.

```
Lisp> (DEFUN SET-TIMER-AND-WAIT (SECONDS)
        (LET* ((DELTA 0)
               (FLAG (LIST NIL))
               (IIF (INSTATE-INTERRUPT-FUNCTION
                        #'SET-FLAG
                        :ONCE-ONLY-P T
                        :ARGUMENTS (LIST FLAG))))
          (CALL-OUT SYS$BINTIM (TIME-STRING SECONDS) DELTA)
          (CALL-OUT SYS$SETIMR NIL DELTA
                                COMMON-AST-ADDRESS IIF)
          (WAIT "Timer wait" #'CAR FLAG))
        (PRINC "The timer has expired")
        T)
SET-TIMER-AND-WAIT
Lisp> (DEFUN SET-FLAG (FLAG)
        (SETF (CAR FLAG) T))
SET-FLAG
```

WAIT Function (cont.)

```
Lisp> (SET-TIMER-AND-WAIT 5)
(Five seconds elapse) The timer has expired
T
Lisp>
```

- The function SET-TIMER-AND-WAIT is defined. It binds the symbol FLAG to a list whose only element is NIL, then causes that list to be passed to the interrupt function SET-FLAG as its only argument. SET-TIMER-AND-WAIT then calls out to the external routine SYS$SETIMR, specifying that the interrupt function SET-FLAG be executed when the timer expires. Finally, SET-TIMER-AND-WAIT calls the WAIT function, specifying CAR as the testing function and the list to which FLAG is bound as the argument to CAR.

- The function SET-FLAG is defined. It sets the first element of the list passed to it to T.

- SET-TIMER-AND-WAIT is called. It executes as far as the WAIT function call. WAIT does not return until the timer expires and causes the first element of FLAG to be set to T.

INDEX

Page numbers in the Index in the form c-n (for example, 2-13) refer to a page in Part I. Page numbers in the form n (for example, 25) refer to a page in Part II.