

VAX LISP/ULTRIX System Access Programming Guide

Order Number: AA-EV40A-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: ULTRIX-32 Version 1.2
ULTRIX-32m Version 1.2

Software Version: VAX LISP/ULTRIX Version 2.0

**digital equipment corporation
maynard, massachusetts**

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.

© Digital Equipment Corporation 1986.
All Rights Reserved.

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
DECUS
MicroVAX
VAXstation
DECnet
ULTRIX-32
ULTRIX-32m

UNIBUS
VAX
MicroVAX II
VAXstation II
ULTRIX

PDP
VMS
MicroVMS
AI VAXstation
ULTRIX-11

digital™

CONTENTS

vii

PREFACE

PART I
GUIDE TO SYSTEM ACCESS PROGRAMMING

CHAPTER 1	OVERVIEW OF SYSTEM ACCESS FACILITIES	
1.1	THE CALL-OUT FACILITY	1-1
1.2	ALIEN STRUCTURES	1-2
1.3	CONTROLLING INTERRUPTIONS AND SYNCHRONIZING EXECUTION	1-3
CHAPTER 2	CALLING EXTERNAL ROUTINES	
2.1	STEPS TO TAKE IN CALLING AN EXTERNAL ROUTINE	2-2
2.2	STANDARD VAX CALLING CONVENTIONS	2-2
2.2.1	Transfer of Control	2-3
2.2.2	Argument Lists	2-3
2.2.3	Mechanisms for Passing Arguments	2-3
2.2.4	Values Returned by Functions	2-4
2.3	DEFINING AN EXTERNAL ROUTINE	2-4
2.3.1	External Routine Name and Options	2-5
2.3.2	External Routine Name	2-5
2.3.3	External Routine Options	2-5
2.3.3.1	Checking the Return Status	2-6
2.3.3.2	Naming the Entry Point	2-6
2.3.3.3	Naming the Object File and/or Required Libraries	2-7
2.3.3.4	Specifying the Result Data Type	2-7
2.3.3.5	Checking the Argument Data Types	2-7
2.3.4	Documentation String	2-8
2.3.5	Argument Descriptions	2-8
2.3.6	Argument Name	2-8
2.3.7	Argument Options	2-8
2.3.7.1	Access Capability	2-9
2.3.7.2	LISP Data Type	2-9
2.3.7.3	Passing Mechanism	2-10
2.3.7.4	VAX Data Type	2-10
2.4	CALLING AN EXTERNAL ROUTINE	2-10
2.4.1	How to Call an External Routine	2-11
2.4.2	What the CALL-OUT Macro Does	2-11
2.4.3	How the CALL-OUT Macro Uses Internal Data Structures	2-12
2.5	DATA TYPE CONVERSIONS	2-12
2.5.1	Converting LISP Objects to VAX Data Types	2-12
2.5.2	Arguments with :IN-OUT Access	2-14
2.5.3	:ASCIZ VAX Type	2-14
2.5.4	Converting VAX Data Types to LISP Objects	2-14

2.6	ERRORS DURING EXTERNAL ROUTINE EXECUTION	2-15
2.7	SUSPENDING A LISP SYSTEM CONTAINING EXTERNAL ROUTINE DEFINITIONS	2-16
2.7.1	Acquiring Memory	2-16
2.7.2	Initializing Data	2-16
2.7.3	Using Open Files	2-17
2.7.4	Having Rooted File Names	2-17
2.8	EXAMPLES OF USING THE CALL-OUT FACILITY	2-17
CHAPTER 3	DEFINING AND CREATING ALIEN STRUCTURES	
3.1	DEFINING AN ALIEN STRUCTURE DATA TYPE	3-2
3.2	WHAT THE DEFINE-ALIEN-STRUCTURE MACRO DOES	3-3
3.3	ALIEN STRUCTURE NAME, OPTIONS, AND DOCUMENTATION STRING	3-4
3.3.1	Alien Structure Name	3-5
3.3.2	Options	3-5
3.3.2.1	Naming Access Functions	3-6
3.3.2.2	Naming the Constructor Function	3-6
3.3.2.3	Naming the Copier Function	3-7
3.3.2.4	Naming the Predicate Function	3-7
3.3.2.5	Specifying a Print Function	3-8
3.3.3	Documentation String	3-9
3.4	ALIEN STRUCTURE FIELD DESCRIPTIONS	3-10
3.4.1	Field Name	3-10
3.4.2	Field Type	3-10
3.4.2.1	Given Field Types	3-11
3.4.2.2	User-Defined Field Types	3-14
3.4.3	Field Positions	3-14
3.4.3.1	Start and End Positions	3-14
3.4.3.2	Gaps Between Field Positions	3-15
3.4.3.3	Overlapping Fields	3-15
3.4.4	Field Options	3-16
3.4.4.1	Initial Value	3-16
3.4.4.2	Read-Only Value	3-17
3.4.4.3	Repeated Field	3-18
3.4.4.4	Similar-Field Distances	3-18
3.5	EXAMPLES OF ALIEN STRUCTURE DEFINITIONS	3-19
3.6	CREATING AN ALIEN STRUCTURE	3-22
3.6.1	Initializing and Changing Data Fields	3-22
3.6.2	Allocating Memory	3-23
3.7	ADDITIONAL ALIEN STRUCTURE MACRO AND FUNCTIONS	3-24
CHAPTER 4	INTERRUPT LEVELS, CRITICAL SECTIONS, AND SYNCHRONIZATION	
4.1	USING INTERRUPT LEVELS	4-1
4.2	CRITICAL SECTIONS	4-2
4.3	SYNCHRONIZING PROGRAM EXECUTION	4-3

PART II
OBJECT DESCRIPTIONS

ALIEN-FIELD Function	1
ALIEN-STRUCTURE-LENGTH Function	3
CALL-OUT Macro	6
CRITICAL-SECTION Macro	8
DEFINE-ALIEN-FIELD-TYPE Macro	9
DEFINE-ALIEN-STRUCTURE Macro	12
DEFINE-EXTERNAL-ROUTINE Macro	18
WAIT Function	23

INDEX

FIGURES

	2-1 Calling External Routines	2-2
	3-1 Internal Storage of FAMILY-REC	3-21

TABLES

	2-1 Keywords Specifying External Routine Options	2-6
	2-2 Keywords Specifying External Routine Argument Options	2-9
	2-3 Values of the :MECHANISM Keyword	2-10
	2-4 Conversion Table from LISP Type to VAX Type	2-12
	2-5 Conversion Table from VAX Type to LISP Type	2-15
	3-1 Alien Structure Field Types	3-11
	3-2 Values Used with Memory-Space Keywords	3-23
	1 DEFINE-ALIEN-STRUCTURE Options	12
	2 DEFINE-ALIEN-STRUCTURE Field Options	15
	3 DEFINE-EXTERNAL-ROUTINE Options	18
	4 DEFINE-EXTERNAL-ROUTINE Argument Options	21



PREFACE

Manual Objectives

The *VAX LISP/ULTRIX System Access Programming Guide* provides information that lets you, as a LISP programmer, make use of the programming interface of the ULTRIX-32 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 ULTRIX-32 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 four 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 shows how you can control the execution of keyboard 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.

PREFACE

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/ULTRIX programming:

- VAX LISP/ULTRIX User's Guide
- COMMON LISP: The Language
- ULTRIX-32 Programmer's Manual
- VAX Architecture Handbook

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: <code>(SETQ NAME LISP)</code>
[]	Square brackets enclose elements that are optional. For example: <code>[doc-string]</code>
...	A horizontal ellipsis means that the element preceding the ellipsis can be repeated. For example: <code>function-name ...</code>
{ }	In function and macro format specifications, braces enclose elements that are considered to be one unit of code. For example: <code>{keyword value}</code>

PREFACE

Convention

Meaning

{ }*

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.

&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

Defined LISP characters, 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.

bold

Names of ULTRIX commands and command options in the text (NOT in the examples) are in **bold** type.

PREFACE

Convention	Meaning
<code>command(n)</code>	The (n) after a command is the section number of the <i>ULTRIX-32 Programmer's Manual</i> that contains a description of that command. For example: <code>vi(1)</code>
<code><RET></code>	A symbol with a 1- to 3-character abbreviation indicates that you press a key on the terminal. For example: <code><RET></code> or <code><ESC></code> In examples, carriage returns are implied at the end of each line. However, the <code><RET></code> symbol is used in some examples to emphasize carriage returns.
<code><CTRL/x></code>	<code>CTRL/x</code> indicates a control key sequence where you hold down the CTRL key while you simultaneously press another key. For example: <code><CTRL/C></code> or <code><CTRL/Y></code> The system echoes control key sequences as <code>^x</code> ; therefore, in examples of output, <code><CTRL/x></code> is shown as <code>^x</code> . For example: <code>^C</code> or <code>^Y</code>
.	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.
Black print	In examples, output lines and prompting characters that the system displays are in black print. For example: <code>Lisp> (CDR '(A B C))</code> <code>(B C)</code> <code>Lisp></code>
Red print	In examples, user input is shown in red print. For example: <code>Lisp> (CDR '(A B C))</code> <code>(B C)</code> <code>Lisp></code>

PART I
GUIDE TO SYSTEM ACCESS PROGRAMMING



CHAPTER 1

OVERVIEW OF SYSTEM ACCESS FACILITIES

The VAX LISP system is layered on top of the ULTRIX 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 ULTRIX operating system offers the following general facilities to any programmer, including the LISP programmer:

- System calls and other library routines. The libraries 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.
- 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 routine written in ULTRIX C.

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 call or library routine that you want to use, or write and debug a routine in another language. Information about system calls and library routines is in

OVERVIEW OF SYSTEM ACCESS FACILITIES

Sections 2 and 3 of the *ULTRIX-32 Programmer's Manual*. This documentation has information about the arguments that each routine expects, its effects, and the value, if any, that it returns.

If you write a routine in another language, you must be aware of the 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.

OVERVIEW OF SYSTEM ACCESS FACILITIES

1.3 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 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, 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 C and other compiled languages supported by ULTRIX. The routines must be provided in an object file acceptable as input to the ld(1) linker.

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, it 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.

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.
- 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.

CALLING EXTERNAL ROUTINES

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. Define it in LISP.
5. Call it from LISP.

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

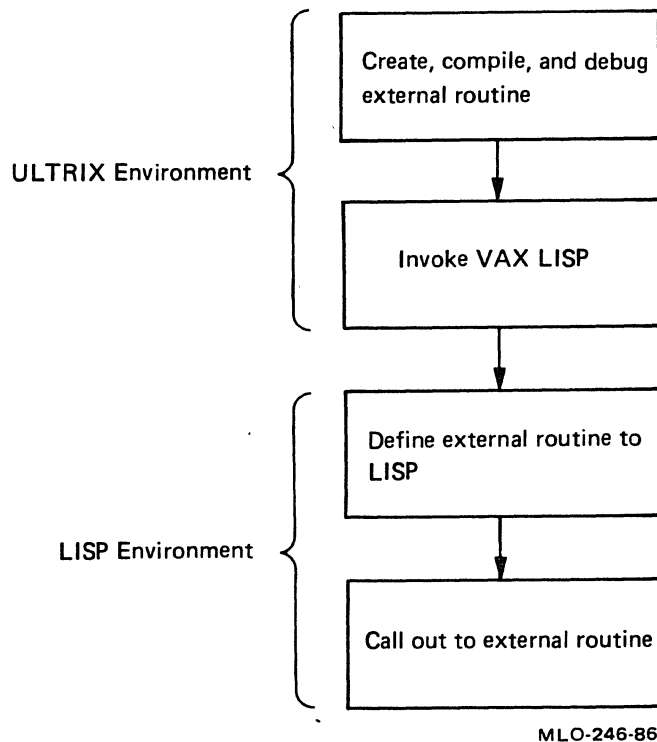


Figure 2-1: Calling External Routines

2.2 STANDARD VAX CALLING CONVENTIONS

The VAX Procedure Calling Standard defines a uniform method for language routines to call one another -- see the *VAX Architecture Handbook*. However, Interpreted and compiled VAX LISP programs cannot conform to this standard because of the nature of the LISP language.

CALLING EXTERNAL ROUTINES

For this reason, VAX LISP provides a facility that lets you call routines written in other VAX languages that do conform to the standard. Thus, to call an external routine, that routine must follow the VAX Procedure Calling Standard. The next four sections briefly summarize how VAX LISP calls external routines conforming to this standard. These sections cover the following areas with which the standard deals:

- How external routines receive and return control
- How arguments are passed
- Which mechanisms are used to pass arguments
- How function values are returned

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.

CALLING EXTERNAL ROUTINES

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

Section 2.3.7.3 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 DEFINING AN EXTERNAL ROUTINE

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

To call an external routine, you must provide an object (.o) file that is acceptable as input to the ld(1) linker. To do so, compile the external routine. Then, enter VAX LISP and define the routine in LISP, using the DEFINE-EXTERNAL-ROUTINE macro. This definition provides LISP with the information needed to build an argument list, link, and call the routine. A description of the DEFINE-EXTERNAL-ROUTINE macro is provided in Part II.

CALLING EXTERNAL ROUTINES

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 2.4.1.

```
Lisp> (DEFINE-EXTERNAL-ROUTINE (SYSTEM :ENTRY-POINT "_system")
    "This lets you use shell commands from VAX LISP."
    (COMMAND :LISP-TYPE STRING :VAX-TYPE :ASCIZ))
SYSTEM
```

The external routine name SYSTEM in this example identifies the shell command **system**(3). This definition lets you call other shell commands from VAX LISP.

2.3.1 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.

2.3.2 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 2.3.3.2).

Note that COMMON LISP symbols generally have uppercase print names while ULTRIX routine names are usually in lowercase.

2.3.3 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.

CALLING EXTERNAL ROUTINES

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 2.8.

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 object file(s) and any required libraries
:RESULT	To define the data type of the result
:TYPE-CHECK	To check the data types of the arguments

2.3.3.1 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 an integer, the R0 register is assumed to contain a status code. If the result returned by the R0 is equal to that integer, 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.

2.3.3.2 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.) You must add underscores to the name if using a language that adds underscores.

CALLING EXTERNAL ROUTINES

NOTE

By default, COMMON LISP symbols have uppercase print names and by convention, ULTRIX compiled languages have lowercase entry-point names.

2.3.3.3 Naming the Object File and/or Required Libraries - The `:FILE` keyword specifies an external routine's object file name. You must use this keyword unless you are calling a routine in the C library, which is always linked in last. Specify the `:FILE` keyword with a string that represents the ULTRIX file name of the external routine's object file or archive library.

For more than one object file or library, specify them as a list of strings in the order needed. The files are linked in the order specified, and the C library is linked last.

NOTE

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

2.3.3.4 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.

2.3.3.5 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.

CALLING EXTERNAL ROUTINES

NOTE

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

2.3.4 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.

2.3.5 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.

2.3.6 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.

2.3.7 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:

CALLING EXTERNAL ROUTINES

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)
: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)

2.3.7.1 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 2.5.2 for more details.

2.3.7.2 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.

CALLING EXTERNAL ROUTINES

2.3.7.3 **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
<code>:VALUE</code>	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.
<code>:REFERENCE</code>	Reference	The reference mechanism passes the address of the argument in the argument list.
<code>:DESCRIPTOR</code>	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.

2.3.7.4 **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.

2.4 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.

CALLING EXTERNAL ROUTINES

2.4.1 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 is an example of calling out to the external routine SYSTEM defined in Section 2.3. This external routine lets you use shell commands from within LISP. In this example, the shell command **date** gives the time and date:

```
Lisp> (CALL-OUT SYSTEM "date")  
Fri Jun 28 15:52:53 EDT 1985  
Lisp>
```

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.

2.4.2 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.

CALLING EXTERNAL ROUTINES

2.4.3 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.

2.5 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.

2.5.1 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

CALLING EXTERNAL ROUTINES

Table 2-4 (cont.)

LISP Type	Default VAX Type	Mechanisms Allowed	Descriptor Class/ Data Type
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
(ARRAY (UNSIGNED-BYTE 16))	* :UNSIGNED-WORD	R,D	Array/WU
(ARRAY (SIGNED-BYTE 32))	* :LONGWORD	R,D	Array/L

CALLING EXTERNAL ROUTINES

Table 2-4 (cont.)

LISP Type	Default	VAX Type	Mechanisms Allowed	Descriptor Class/Data Type
(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

2.5.2 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.

2.5.3 :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.

2.5.4 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 2.3.3). All other cases require that you specify both the LISP and the VAX types. Since

CALLING EXTERNAL ROUTINES

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

2.6 ERRORS DURING EXTERNAL ROUTINE EXECUTION

VAX LISP/ULTRIX sets up its own error handlers for a number of ULTRIX signals. This allows the binding of control keys (see the VAX LISP/ULTRIX User's Guide, Chapter 2) and allows ULTRIX-caught errors to be signaled in LISP. Changing the default handlers on calling-out can cause unpredictable results. The signals to avoid are:

SIGBUS	SIGILL	SIGPIPE	SIGSYS	SIGXFSZ
SIGEMT	SIGINT	SIGQUIT	SIGTRAP	
SIGFPE	SIGIOT	SIGSEGV	SIGTSTP	

CALLING EXTERNAL ROUTINES

2.7 SUSPENDING A LISP SYSTEM CONTAINING EXTERNAL ROUTINE DEFINITIONS

You can suspend a LISP system after having called out to external routines. When you suspend such a system, you must be aware of restrictions to ensure correct operation on resuming. The restrictions exist because external routines and their data are loaded into memory outside the LISP environment, and all such memory is lost in a suspended system. When the system is resumed, external routines are reloaded when first called, but any other program state will not be maintained. Unpredictable results can occur from the following:

- Memory acquired with `brk(2)`, `sbrk(2)`, and `malloc(3)`
- Data initialization
- Open files
- Unrooted file names in the `:FILE` option of the `DEFINE-EXTERNAL-ROUTINE` macro

2.7.1 Acquiring Memory

Memory acquired in an external routine is deleted when you exit the LISP system and is not restored on resuming LISP. This prevents you from storing data in acquired memory across a suspend/resume cycle. Thus, you should store data as a LISP object with a special variable pointing to the data.

2.7.2 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 is relinked. 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, causing a problem if you want to retain the saved data during a suspend/resume cycle.

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

- Retain data as individual LISP objects and pass them as arguments to the external routine.
- Store data in statically allocated alien structures and pass them as arguments to the external routine.

CALLING EXTERNAL ROUTINES

Undesired side effects do not occur if external routines are defined with the `DEFINE-EXTERNAL-ROUTINE` macro and the resulting system is suspended before a call to any external routine.

2.7.3 Using Open Files

When a LISP system is suspended and later resumed, any files opened by external routines before the suspend are not reopened by VAX LISP. Thus, make sure you explicitly open and close files if you need to.

2.7.4 Having Rooted File Names

If the suspended system is resumed in a process attached to a different working directory than at the original suspend time, the initial call-out after resume will fail to find unrooted file names. To avoid this problem, always use rooted file names (complete pathnames) in the `:FILE` option of the `DEFINE-EXTERNAL-ROUTINE` macro.

2.8 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 (SYSTEM
                                   :ENTRY-POINT "_system"
                                   :TYPE-CHECK T)
          (COMMAND :LISP-TYPE STRING
                   :VAX-TYPE :ASCIZ))
```

SYSTEM

The external routine `SYSTEM` (also defined in Section 2.3) lets you use shell commands from VAX LISP.

```
Lisp> (CALL-OUT SYSTEM "date")
Fri Jun 28 15:52:53 EDT 1985
Lisp>
```

Calling out to `SYSTEM` with the shell command `date` gives the time and date.

CALLING EXTERNAL ROUTINES

```
2. Lisp> (DEFINE-EXTERNAL-ROUTINE (RENAME
                                   :ENTRY-POINT "_rename"
                                   :CHECK-STATUS-RETURN -1)
      "This external routine renames the specified
      file to the specified new name."
      (FROM :LISP-TYPE STRING :VAX-TYPE :ASCIZ )
      (TO   :LISP-TYPE STRING :VAX-TYPE :ASCIZ ))
```

RENAME

```
Lisp> (CALL-OUT SYSTEM "ls *.lsp")
test.lsp
```

```
Lisp> (CALL-OUT RENAME "test.lsp"
                  "program.lsp")
```

0

```
Lisp> (CALL-OUT SYSTEM "ls *.lsp")
```

program.lsp

```
Lisp> (CALL-OUT RENAME "program.lsp"
                  "/etc/program.lsp")
```

Continuable error

Debug 1>

The first call-out in this example is to the external routine SYSTEM defined in the first example. The ls shell command lists any files with a lsp file type, which happens to be one file, test.lsp.

The second call-out is to the external routine RENAME. This routine lets you use the shell subroutine rename from LISP. Since no error occurs, the value 0 is returned. The next call-out to the ls command shows that the RENAME routine was successful.

In the last call-out to the external routine RENAME, the routine returned a -1 indicating a protection violation. The consequent error invoked the debugger.

```
3. Lisp> (DEFINE-EXTERNAL-ROUTINE (DELETE-DIRECTORY
                                   :ENTRY-POINT "_rmdir"
                                   :CHECK-STATUS-RETURN -1)
      "This lets you delete a directory"
      (DIRECTORY-NAME :LISP-TYPE STRING
                      :VAX-TYPE :ASCIZ))
```

DELETE-DIRECTORY

```
Lisp> (CALL-OUT DELETE-DIRECTORY "subdirectory_a")
```

0

CALLING EXTERNAL ROUTINES

Lisp> (CALL-OUT DELETE-DIRECTORY "subdirectory_b")

Continuable error

Debug 1>

In this example, subdirectory_a is successfully deleted, indicated by the routine returning a 0. However, subdirectory_b does not exist, causing a continuable error, and the debugger is invoked.



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

DEFINING AND CREATING ALIEN STRUCTURES

- 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.

DEFINING AND CREATING ALIEN STRUCTURES

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.

DEFINING AND CREATING ALIEN STRUCTURES

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.

DEFINING AND CREATING ALIEN STRUCTURES

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

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

DEFINING AND CREATING ALIEN STRUCTURES

- :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-INTEGERS 0 4)
      (AREA-2 :UNSIGNED-INTEGERS 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:

DEFINING AND CREATING ALIEN STRUCTURES

```
Lisp> (DEFINE-ALIEN-STRUCTURE (SPACE (:CONSTRUCTOR CREATE-SPACE))
      (AREA-1 :UNSIGNED-INTEGERS 0 4)
      (AREA-2 :UNSIGNED-INTEGERS 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 constructor 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-INTEGERS 0 4)
      (AREA-2 :UNSIGNED-INTEGERS 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.

DEFINING AND CREATING ALIEN STRUCTURES

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-INTEGERS 0 4)
      (AREA-2 :UNSIGNED-INTEGERS 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 structure 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-INTEGERS 0 4)
      (AREA-2 :UNSIGNED-INTEGERS 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

DEFINING AND CREATING ALIEN STRUCTURES

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.

DEFINING AND CREATING ALIEN STRUCTURES

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.

DEFINING AND CREATING ALIEN STRUCTURES

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-INTEGERS	Signed two's complement integer
:UNSIGNED-INTEGERS	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 equivalent to a VAX :G-FLOATING type.

:H-FLOATING	H_floating data
-------------	-----------------

DEFINING AND CREATING ALIEN STRUCTURES

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)
```


DEFINING AND CREATING ALIEN STRUCTURES

: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-INTEGERS 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.

DEFINING AND CREATING ALIEN STRUCTURES

```
Lisp> (SETF (MAP-STATE GEO) "CALIFORNIA")  
"CALIFORNIA"  
Lisp> (ALIEN-FIELD GEO :UNSIGNED-INTEGERS 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 INTEGERS 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.

DEFINING AND CREATING ALIEN STRUCTURES

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-INTEGERS 0 4)
      (BIT-0  :UNSIGNED-INTEGERS 0 1/8)
      (BIT-1  :UNSIGNED-INTEGERS 1/8 2/8)
      (BIT-2  :UNSIGNED-INTEGERS 2/8 3/8)
      (BIT-3  :UNSIGNED-INTEGERS 3/8 4/8)
      (BIT-4  :UNSIGNED-INTEGERS 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>
```

DEFINING AND CREATING ALIEN STRUCTURES

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

DEFINING AND CREATING ALIEN STRUCTURES

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 definition.

DEFINING AND CREATING ALIEN STRUCTURES

```
(DEFINE-ALIEN-STRUCTURE (SPACE (:PRINT-FUNCTION #'SPACE-PRINT))
                        (AREA-1 :UNSIGNED-INTEGERS 0 4 :DEFAULT 2)
                        (AREA-2 :UNSIGNED-INTEGERS 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-INTEGERS 0 4)
      (AREA-2 :UNSIGNED-INTEGERS 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 24 8-bit bytes come between each occurrence of the `CHILD-NAME` field:

```
(CHILD-NAME :TEXT 72 92 :OCCURS 20 :OFFSET 24)
```

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

DEFINING AND CREATING ALIEN STRUCTURES

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 C record structure definition:

```
STRUCT PERSON { /* A structure representing a person. */  
    CHAR NAME[20];  
    UNSIGNED INT AGE;  
};
```

```
STRUCT FAMILYREC { /* A structure representing a family. */  
    CHAR SURNAME[20];  
    STRUCT PERSON FATHER;  
    STRUCT PERSON MOTHER;  
    UNSIGNED INT NUMCHILDREN;  
    STRUCT PERSON CHILD[20];  
};
```

DEFINING AND CREATING ALIEN STRUCTURES

An equivalent LISP record structure definition:

```
Lisp> (DEFINE-ALIEN-STRUCTURE FAMILY-REC
      "A record structure definition."
      (SURNAME :ASCIZ 0 20)
      (FATHER-NAME :ASCIZ 20 40)
      (FATHER-AGE :UNSIGNED-INTEG 40 44)
      (MOTHER-NAME :ASCIZ 44 64)
      (MOTHER-AGE :UNSIGNED-INTEG 64 68)
      (NUM-CHILDREN :UNSIGNED-INTEG 68 72 :DEFAULT 2)
      (CHILD-NAME :ASCIZ 72 92 :OCCURS 20 :OFFSET 24)
      (CHILD-AGE :UNSIGNED-INTEG 92 96 :OCCURS 20
                :OFFSET 24))
```

FAMILY-REC

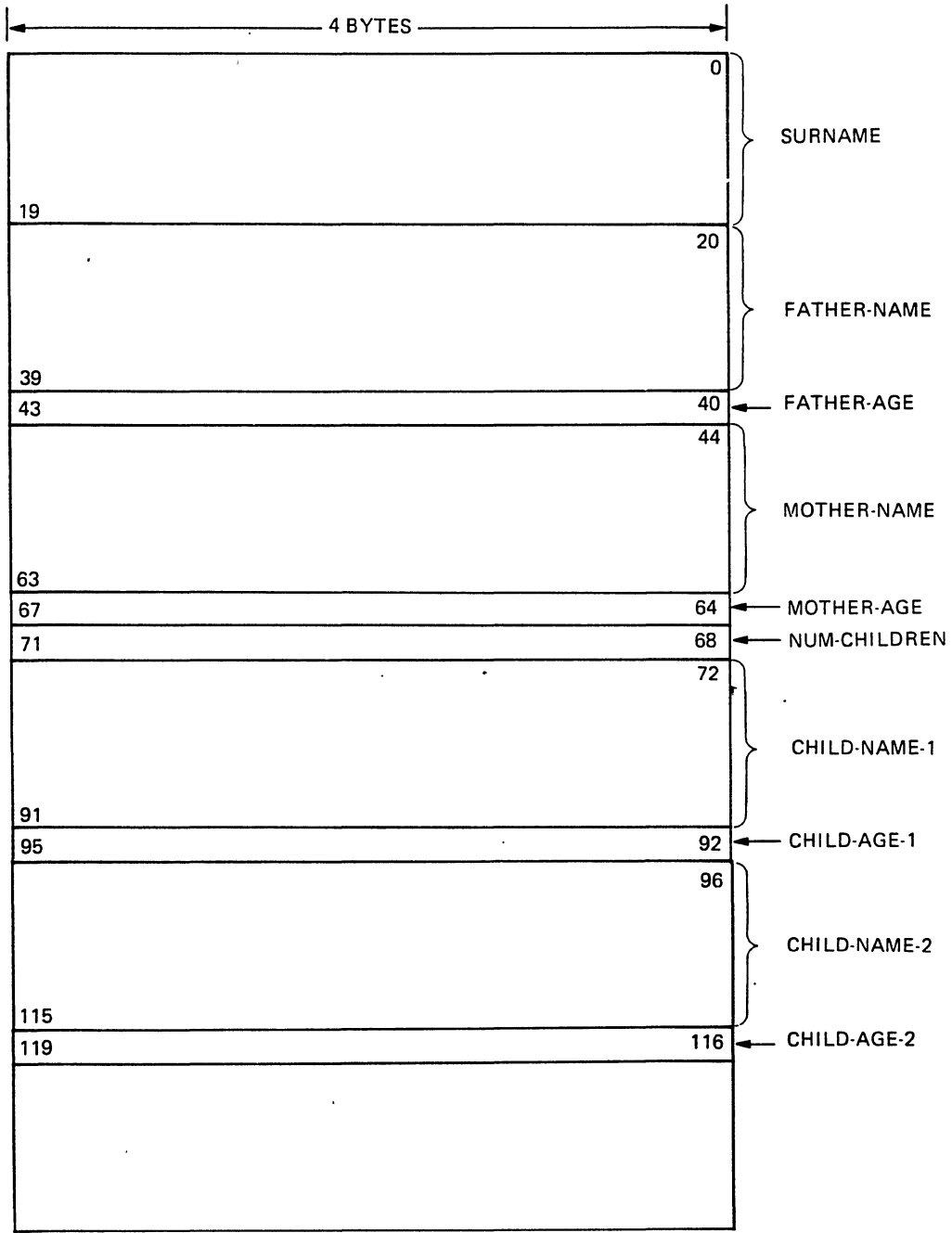
This form defines an alien structure named FAMILY-REC which has 46 fields indicating the members of a family and their ages. The definition contains the :DEFAULT, :OCCURS, and :OFFSET keywords.

The only fields that repeat are the CHILD-NAME and the CHILD-AGE 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 can be up to four characters in length. There is a gap between each occurrence of the CHILD-NAME field since the field contains 20 bytes but repeats itself every 24 bytes. This gap is filled by the CHILD-AGE field.

The following diagram 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.

DEFINING AND CREATING ALIEN STRUCTURES



MLO-245-86

Figure 3-1: Internal Storage of FAMILY-REC

DEFINING AND CREATING ALIEN STRUCTURES

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-INTEGERS 0 4)  
      (AREA-2 :UNSIGNED-INTEGERS 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-INTEGERS 0 4 :DEFAULT 6)  
      (AREA-2 :UNSIGNED-INTEGERS 4 8 :DEFAULT 12))  
SPACE
```

DEFINING AND CREATING ALIEN STRUCTURES

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 <i>integer</i>	<p>The number of bytes of memory to be allocated for the alien structure's data vector.</p> <p>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.</p> <p>See the description of the ALIEN-STRUCTURE-LENGTH function in Part II for an example of default byte allocations.</p>

DEFINING AND CREATING ALIEN STRUCTURES

Table 3-2 (cont.)

Keyword	Value
:ALLOCATION value	<p>The type of allocation to be used for the alien structure. Valid values are :DYNAMIC and :STATIC. :DYNAMIC is the default.</p> <p>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/ULTRIX User's Guide).</p>

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 LEVELS, CRITICAL SECTIONS, AND SYNCHRONIZATION

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

This chapter discusses the following subjects:

- Section 4.1 describes the system of interrupt levels. You can specify an interrupt level with the BIND-KEYBOARD-FUNCTION function.
- Section 4.2 describes critical sections, which you use to prevent a section of code from being interrupted during execution.
- Section 4.3 describes the WAIT function, which you can use to suspend execution of your program until a keyboard function executes.

4.1 USING INTERRUPT LEVELS

You can use the :LEVEL keyword to assign an interrupt level 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, 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.

INTERRUPT LEVELS, CRITICAL SECTIONS, AND SYNCHRONIZATION

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, 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 interfere with VAX LISP's normal operation.
- 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.
- In this framework, choose interrupt levels for your 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.)

4.2 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 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.

INTERRUPT LEVELS, CRITICAL SECTIONS, AND SYNCHRONIZATION

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.

4.3 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 rely on the values of special variables. You should pass it arguments instead.

INTERRUPT LEVELS, CRITICAL SECTIONS, AND SYNCHRONIZATION

One way to use WAIT is with a 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 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
 #\FS
 #'(LAMBDA () (SETF (CAR FLAG) T)))
(WAIT "Wait for CTRL/\ " #'CAR FLAG)
```

In this example, the value of FLAG is a list whose only element is NIL. BIND-KEYBOARD-FUNCTION binds <CTRL/\> 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/\>, the first element of FLAG is set to T, the testing function returns T, the WAIT function returns, and normal execution continues.

PART II
OBJECT DESCRIPTIONS



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.

OBJECT DESCRIPTIONS

ALIEN-FIELD Function (cont.)

Example

```
Lisp> (DEFINE-ALIEN-STRUCTURE SPACE
      (AREA-1 :UNSIGNED-INTEGERS 0 4 :DEFAULT 22)
      (AREA-2 :UNSIGNED-INTEGERS 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-INTEGERS 0 4)
22
Lisp> (ALIEN-FIELD SPACE-RECORD :UNSIGNED-INTEGERS 4 8)
2764
Lisp> (ALIEN-FIELD SPACE-RECORD :UNSIGNED-INTEGERS 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.

OBJECT DESCRIPTIONS

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
```

```
0           20           40           60  
+           +           +           +  
-----  
| name1     | name2     | name3     |  
-----  
|           |           |           |  
+--- offset ---+  
|           |           |           |  
|           |           |           |  
+--- offset = 20
```

OBJECT DESCRIPTIONS

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
```

```
0          20          40
+          +          +
+-----+
| name1   |
+-----+
          +-----+
          | name2   |
          +-----+
|         | +-----+
+-----+ | name3   |
|         | +-----+
|         |
+----- offset = 10
```

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
```

```
0          20          40          60
+          +          +          +
+-----+-----+-----+
| name1   | gap     | name2   |
+-----+-----+-----+
|         |         |         |
+----- offset -----+
|         |         |         |
+----- offset = 40
```

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

OBJECT DESCRIPTIONS

ALIEN-STRUCTURE-LENGTH Function (cont.)

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

Lisp> (ALIEN-STRUCTURE-LENGTH (MAKE-EXAMPLE4))
40

0	20	40
+	+	+
-----+		
gap	name1	
-----+		

OBJECT DESCRIPTIONS

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. 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.

OBJECT DESCRIPTIONS

CALL-OUT Macro (cont.)

Example

```
Lisp> (DEFINE-EXTERNAL-ROUTINE
      (SYSTEM :ENTRY-POINT "_system"
              :CHECK-STATUS-RETURN -1)
      (COMMAND :LISP-TYPE STRING
              :VAX-TYPE :ASCIZ))
SYSTEM
```

In this example, the shell command `system(3)` is defined as an external routine. A call to `SYSTEM` starts the Bourne shell in a separate process and runs the specified command. In the following example of calling out to `SYSTEM`, `date` is the specified command. The `:CHECK-STATUS-RETURN` keyword causes an integer result to be returned to LISP. Zero is a success status for this call to `_system`.

```
Lisp> (CALL-OUT SYSTEM "date")
Fri Jun 28 15:2:53 EDT 1985
0
```

OBJECT DESCRIPTIONS

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 should 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.

OBJECT DESCRIPTIONS

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

OBJECT DESCRIPTIONS

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-INTEGERS 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.

OBJECT DESCRIPTIONS

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

```
2. Lisp> (DEFINE-ALIEN-FIELD-TYPE SELECTION
          T
          :UNSIGNED-INTEGERS
          #'(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,

OBJECT DESCRIPTIONS

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.

OBJECT DESCRIPTIONS

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Table 1 (cont.)

Keyword-Value Pair	Description
:PRINT-FUNCTION <i>function-name</i>	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, <i>name</i> is the name of the alien structure and <i>number</i> 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.

OBJECT DESCRIPTIONS

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:

:ASCIZ	:STRING
:ASCIW	: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 <i>form</i>	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 <i>form</i> is evaluated when the

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Table 2 (cont.)

Keyword-Value Pair	Description
	constructor function is called. 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.

OBJECT DESCRIPTIONS

DEFINE-ALIEN-STRUCTURE Macro (cont.)

Example

```
Lisp> (DEFINE-ALIEN-STRUCTURE ET
      (SPACE-SHIP :ASCIZ 0 10)
      (PHONE-NUMBER :UNSIGNED-INTEGERS 10 17)
      (HOME :ASCIZ 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.

OBJECT DESCRIPTIONS

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. Note that if you call out to routines that change the default handler of the ULTRIX signals, the results are not predictable. 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 <i>value</i>	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 an integer or NIL. If you specify an integer,

OBJECT DESCRIPTIONS

DEFINE-EXTERNAL-ROUTINE Macro (cont.)

Table 3 (cont.)

Keyword-Value Pair	Description
	the call-out facility checks the return value. If the return value is the specified value, the LISP system signals a continuable error. If you specify NIL, the call-out facility does not check the return value. NIL is the default value.
:ENTRY-POINT <i>string</i>	Names the external routine's entry point. The value must be a string. You must add lead underscores explicitly. Case is preserved. The default value is the print name of the external routine name, which is normally in upper case. If an entry point of this name is already accessible in the system, a new version of the routine is not linked in.
:FILE { <i>string</i> <i>list-of-strings</i> }	Specifies the object file(s) or archive library(ies) that was created for the external routine. If multiple files are specified, the order is preserved. The C library is always used last. The files must be object files usable as input to the ls(1) linker. You must specify this option unless you are calling a routine in the C library.
:RESULT <i>type</i>	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 <i>LISP-type</i> :VAX-TYPE <i>VAX-type</i>)

OBJECT DESCRIPTIONS

DEFINE-EXTERNAL-ROUTINE Macro (cont.)

Table 3 (cont.)

Keyword-Value Pair	Description
	See Table 2-4 for a list of LISP/VAX types. NIL specifies 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

OBJECT DESCRIPTIONS

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 for all data types (except for the VAX-type of :TEXT) is :REFERENCE.

OBJECT DESCRIPTIONS

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
      (SYSTEM :ENTRY-POINT "_system")
      (COMMAND :LISP-TYPE STRING
               :VAX-TYPE :ASCIZ))
```

This defines the external routine SYSTEM which lets you call out to the shell command `system(3)`. This lets you call other shell commands from VAX LISP.

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.

OBJECT DESCRIPTIONS

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 4 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.

OBJECT DESCRIPTIONS

WAIT Function (cont.)

Example

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

- The special variable `*FLAG*` is set to a list whose only element is `NIL`.
- `<CTRL/\>` 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/\>`, the keyboard function sets the first element of `*FLAG*` to `T`, the testing function returns `T`, and the call to `WAIT` returns.

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.

-A-

Access function, 3-3
 defining field types, 9
 generating, 16
 naming, 13
:ACCESS keyword
 DEFINE-EXTERNAL-ROUTINE macro,
 2-9, 21
Access method, 2-9
Alien structure
 access function, 3-3
 defining field types, 9
 generating, 16
 naming, 13
 constructor function, 3-3, 3-17
 naming, 13
 specifying an initial value,
 16
 copier function, 3-4, 3-7
 naming, 13
 creating, 3-22
 defining, 3-2, 12
 examples, 3-19
 field
 See Field
 field description
 See Field
 internal storage
 (figure), 3-22
 length, 3-24, 3
 modify, 1
 name, 3-4, 12
 options, 12
 predicate function, 3-4, 3-7
 naming, 13
 print function, 3-4, 3-8
Alien structure facility, 3-1 to
 3-24
 See also Call out facility
:ALIEN-DATA-LENGTH keyword
 constructor function, 3-23

ALIEN-FIELD function, 3-24
 description, 1
ALIEN-STRUCTURE-LENGTH function,
 3-24
 description, 3
:ALLOCATION keyword
 constructor function, 3-24
Argument
 access method, 2-9
 list, 2-3
 passing mechanisms, 2-4, 21
argument description, 20
:ASCIW keyword
 alien structure field type,
 3-11, 15
:ASCIZ keyword
 alien structure field type, 15
:ASCIZ-STRING keyword
 alien structure field type,
 3-11

-B-

BIND-KEYBOARD-FUNCTION
 specifying interrupt level, 4-1
:BIT keyword
 VAX data type, 2-10
:BIT-VECTOR keyword
 alien structure field type,
 3-11, 15
BREAK function
 interrupt level for, 4-2
:BYTE keyword
 VAX data type, 2-10

-C-

Call-out facility, 2-1 to 2-19
 See also Alien structure
 facility
CALL-OUT macro, 18
 description, 6
CALLG VAX instruction, 2-3

INDEX

- :CHECK-STATUS-RETURN keyword
 - DEFINE-EXTERNAL-ROUTINE macro, 2-6, 19
- :CONC-NAME keyword
 - DEFINE-ALIEN-STRUCTURE macro, 3-6, 13
- Constructor function, 3-3
 - keywords, 3-22
 - keywords (table), 3-23
 - naming, 13
 - specifying an initial value, 3-17, 16
- :CONSTRUCTOR keyword
 - DEFINE-ALIEN-STRUCTURE macro, 3-6, 13
- Copier function, 3-4, 3-7
 - naming, 13
- :COPIER keyword
 - DEFINE-ALIEN-STRUCTURE macro, 3-7, 13
- Critical sections
 - infinite loops in, 4-3
- Critical sections, 4-2
 - debugging, 4-3
 - errors in, 4-3
- CRITICAL-SECTION macro, 8
 - using, 4-2
- DEBUG function
 - interrupt level for, 4-2
- :DEFAULT keyword
 - DEFINE-ALIEN-STRUCTURE macro, 3-17, 16
- DEFINE-ALIEN-FIELD-TYPE macro, 3-24
 - description, 9
- DEFINE-ALIEN-STRUCTURE macro
 - defining an alien structure, 3-2
 - description, 12 to 17
 - options (table), 12
- DEFINE-EXTERNAL-ROUTINE macro, 6
 - argument options (table), 21
 - defining external routines, 2-4
 - description, 18 to 22
 - routine options (table), 18
- Descriptor (:DESCR)
 - argument-passing mechanism, 21
- Descriptor (:DESCRIPTION)
 - argument-passing mechanism, 2-4
- Descriptor (:DESCRIPTOR)
 - argument-passing mechanism, 2-10
- Documentation string, 14, 20
- Dynamic memory, 3-24

-D-

- :D-FLOATING keyword
 - alien structure field type, 3-11, 15
 - VAX data type, 2-10
- Data
 - initialization, 2-16
 - structure
 - internal, 2-12
 - VAX, 3-1
- Data types
 - alien structure field, 3-10
 - See also Field
 - alien structures, 3-3
 - checking, 2-7, 20
 - conversions, 2-12, 3-10, 15 (table), 2-15
 - LISP, 9, 20
 - :LISP-TYPE keyword, 2-9
 - :RESULT keyword, 2-7
 - VAX, 2-10, 22
 - :RESULT keyword, 2-7
 - :VAX-TYPE keyword, 2-10

-E-

- End position
 - ALIEN-FIELD function, 1
 - field, 3-14, 15
- :ENTRY-POINT keyword
 - DEFINE-EXTERNAL-ROUTINE macro, 2-7, 19
- Errors
 - in critical sections, 4-3
- External routine
 - access method, 21
 - alien structures, 3-1
 - argument type checking, 2-7
 - calling, 2-1, 2-11, 6 (figure), 2-2
 - checking data types, 20
 - checking status return, 19
 - data initialization, 2-16
 - data type conversion, 2-12 (table), 2-15
 - defining, 2-4, 18
 - entry point, 2-7, 19
 - formal argument

INDEX

External routine
formal argument (Cont.)
See formal argument
description
image name, 2-7, 19
name, 2-5, 18
options, 2-5, 18
result data type, 2-7, 20
status return, 2-6
suspending a LISP system, 2-16

-F-

:F-FLOATING keyword
alien structure field type,
3-11, 15
VAX data type, 2-10

Field

accessing, 3-18, 1
defining, 3-24
description, 3-10
end position, 3-14, 1, 15
gaps, 3-15, 16
index, 3-18
initial value, 3-16, 16
name, 3-10, 1, 14
offset, 3-19
options, 3-16
(table), 15
overlapping, 3-15
repeated, 3-18
repeating, 16
setting, 3-18
start position, 3-14, 1, 15
type, 3-10, 3-11, 1, 15
defining, 3-14, 9
given
(table), 3-11
predefined, 9

File

open, 2-17

:FILE keyword
DEFINE-EXTERNAL-ROUTINE macro,
19

Formal argument description, 2-8

-G-

:G-FLOATING keyword
alien structure field type,
3-11, 15
VAX data type, 2-10

Gaps, 3-15, 16

-H-

:H-FLOATING keyword
alien structure field type,
3-11, 15
VAX data type, 2-10

-I-

:IMAGE-NAME keyword
DEFINE- macro, 2-7
Immediate value (:IMMED)
argument-passing mechanism, 21
Immediate value (:VALUE)
argument-passing mechanism, 2-4,
2-10
Initialization keyword, 3-22
Input access (:IN), 2-9, 21
Input-output access (:IN-OUT),
2-9, 21
Interrupt levels, 4-1
guidelines, 4-2

-K-

Keyboard functions
interrupt levels, 4-1
protecting against interruption
by, 4-2
waiting for, 4-3
Keyboard input
interrupt level of, 4-2

-L-

LISP

data type
See Data types
program
calling external routines,
2-4
defining alien structure data
types, 3-1
:LISP-TYPE keyword
DEFINE-EXTERNAL-ROUTINE macro,
2-9, 21
:LONGWORD keyword
VAX data type, 2-10

INDEX

-M-

:MECHANISM keyword
 DEFINE-EXTERNAL-ROUTINE macro,
 2-10, 21
 Memory
 dynamic, 3-24
 static, 3-24

-O-

:OCCURS keyword
 DEFINE-ALIEN-STRUCTURE macro,
 3-18, 16
 :OFFSET keyword
 DEFINE-ALIEN-STRUCTURE macro,
 3-19, 16

-P-

:POINTER keyword
 alien structure field type,
 3-12, 15
 Predicate function, 3-4
 naming, 13
 :PREDICATE keyword
 DEFINE-ALIEN-STRUCTURE macro,
 3-8, 13
 Print function, 3-4, 14
 alien structure, 3-8
 :PRINT-FUNCTION keyword
 DEFINE-ALIEN-STRUCTURE macro,
 3-8, 14

-R-

:READ-ONLY keyword
 DEFINE-ALIEN-STRUCTURE macro,
 3-18, 16
 Reference (:REF)
 argument-passing mechanism, 21
 Reference (:REFERENCE)
 argument-passing mechanism, 2-4,
 2-10
 :RESULT keyword
 DEFINE-EXTERNAL-ROUTINE macro,
 2-7, 20
 RET VAX instruction, 2-3
 Routine argument, 6

-S-

:SELECTION keyword
 alien structure field type, 15
 SETF macro
 access functions, 3-18
 creating alien structures, 3-3,
 3-15
 with ALIEN-FIELD function, 1
 :SIGNED-INTEGER keyword
 alien structure field type,
 3-11, 15
 Start position
 ALIEN-FIELD function, 1
 field, 3-14, 15
 Static memory, 3-24
 Status return, 2-6, 19
 Storage allocation
 alien structures, 3-24
 :STRING keyword
 alien structure field type,
 3-11, 15
 Suspended systems
 including calls to external
 routines, 2-16

-T-

:TEXT keyword
 alien structure field type,
 3-11
 VAX data type, 2-10
 :TYPE-CHECK keyword
 DEFINE-EXTERNAL-ROUTINE macro,
 2-7, 20

-U-

:UNSIGNED-BYTE keyword
 VAX data type, 2-10
 :UNSIGNED-INTEGER keyword
 alien structure field type,
 3-11, 15
 :UNSIGNED-LONGWORD keyword
 VAX data type, 2-10
 :UNSIGNED-WORD keyword
 VAX data type, 2-10

INDEX

-V-

:VARYING-STRING keyword
 alien structure field type,
 3-11
VAX data type
 See Data types
VAX Procedure Calling Standard,
 2-3
:VAX-TYPE keyword
 DEFINE-EXTERNAL-ROUTINE macro,
 2-10

-W-

WAIT function, 23
 arguments, 4-3
 testing function
 debugging, 4-3
 guidelines, 4-3
 using, 4-3
:WORD keyword
 VAX data type, 2-10

