VAX LISP/VMS System-Building Guide

Order Number: AA–KM37B–TE

This revised manual provides the information needed to create executable LISP images with the VAX LISP System-Building Utility on VMS systems.

Revision/Update Information: This manual supersedes *VAX LISP/VMS System-Building Guide*, AA–KM37A–TE.

Operating System and Version: VMS Version 5.1

Software Version: VAX LISP Version 3.0

digital equipment corporation
maynard, massachusetts

First Printing, July 1987
Revised, July 1989

The postpaid Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

ML–S830

This document was prepared using VAX DOCUMENT, Version 1.1.

Contents

Index

Tables

Preface

The *VAX LISP/VMS System Building Guide* provides the information needed to create executable LISP images with the VAX LISP System-Building Utility on VMS systems.

Intended Audience

Readers of this manual should have a working knowledge of LISP programming and be familiar with the general information about VAX LISP and VMS that appears in the *VAX LISP/VMS Program Development Guide*.

Document Structure

This manual consists of three chapters:

- Chapter 1 provides an overview of the System-Building Utility, the features of the user-built system that the utility creates, and the generic procedure for using the utility.
- Chapter 2 describes the `DEFINE-LISP-SYSTEM` function, which is the center of the System-Building Utility, and the arguments to `DEFINE-LISP-SYSTEM`.
- Chapter 3 explains how to make the read-only portions of a user-built system shareable and how to define a DCL command to invoke a user-built system.

Associated Documents

The following documents are relevant to using the *VAX LISP/VMS System Building Guide*:

- The *VAX LISP/VMS Program Development Guide* provides general information about using VAX LISP and serves as a guide to helpful VAX LISP and VMS documentation.
- The *Guide to Setting Up a VMS System* gives an overview of some VMS utilities that you can use with a user-built LISP executable image.
- The *VAX LISP/VMS Installation Guide* explains the VMS Install Utility, which you can use to make an executable image shareable.
- The *VMS Command Definition Utility Manual* explains the Command Definition Utility, which you can use to create a new DCL command for invoking an executable image.

Conventions

The following conventions are used in this manual:

Convention	Meaning
UPPERCASE	DCL commands and qualifiers and VMS file names are printed in uppercase characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The examples directory (SYS$SYSROOT:[VAXLISP.EXAMPLES] by default) contains sample LISP source files.
`UPPERCASE TYPEWRITER`	Defined LISP functions, macros, variables, constants, and other symbol names are printed in uppercase `TYPEWRITER` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: The `CALL-OUT` macro calls a defined external routine . . . .
`lowercase typewriter`	LISP forms are printed in the text in lowercase `typewriter` characters; however, you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase characters. For example: `(setf example-1 (make-space))`
SANS SERIF	Format specifications of LISP functions and macros are printed in a sans serif typeface. For example: CALL-OUT *external-routine* &REST *routine-arguments*
italics	Lowercase *italics* in format specifications and in text indicate arguments that you supply; however, you can enter them in lowercase, uppercase, or a combination of lowercase and uppercase characters. For example: The *routine-arguments* must be compatible with the arguments defined in the call to the `DEFINE-EXTERNAL-ROUTINE` macro.
( )	Parentheses used in examples of LISP code and in format specifications indicate the beginning and end of a LISP form. For example: `(setq name lisp)`
[ ]	Square brackets in format specifications enclose optional elements. For example: [*doc-string*] Square brackets do not indicate optional elements when they are used in the syntax of a directory name in a VMS file specification. Here, the square bracket characters must be included in the syntax. For example: `(pathname "MIAMI::DBA1:[SMITH]LOGIN.COM;4")`
{ }	In function and macro format specifications, braces enclose elements that are considered one unit of code. For example: {*keyword value*}
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered one unit of code, which can be repeated zero or more times. For example: {*keyword value*}*

Convention	Meaning
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments that follow it are optional. For example: PPRINT *object* &OPTIONAL *stream* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.
&REST	In function and macro format specifications, the word &REST indicates that an indefinite number of arguments may appear. For example: CALL-OUT *external-routine* &REST *routine-arguments* Do not specify &REST when you invoke a function or macro whose definition includes &REST.
&KEY	In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: COMPILE-FILE *input-pathname* &KEY :LISTING :MACHINE-CODE :OPTIMIZE :OUTPUT-FILE :VERBOSE :WARNINGS Do not specify &KEY when you invoke a function or macro whose definition includes &KEY.
. . .	A horizontal ellipsis in a format specification means that the element preceding the ellipsis can be repeated. For example: *function-name* . . .
. . .	A vertical ellipsis in a code example indicates that all the information that the system would display in response to the function call is not shown; or that all the information a user is to enter is not shown.
Return	A word inside a box indicates that you press a key on the keyboard. For example: Return or Tab In code examples, carriage returns are implied at the end of each line. However, Return is used in some examples to emphasize carriage returns.
Ctrl/*x*	Two key names enclosed in a box indicate a control key sequence in which you hold down Ctrl while you press another key. For example: Ctrl/C or Ctrl/S
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1, then press and release another key.
mouse	The term *mouse* refers to any pointing device, such as a mouse, a puck, or a stylus.
MB1, MB2, MB3	By default, MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button. You can rebind the mouse buttons.

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

Chapter 1

Overview of the System-Building Utility

The VAX LISP System-Building Utility enables you to create a LISP system that is a single executable image. This user-built system can serve as:

- A customized VAX LISP development environment that can be shared efficiently by multiple users at the same time
- A delivery vehicle for VAX LISP-based applications

This chapter gives an overview of the System-Building Utility and of the user-built system that it creates. The chapter introduces:

- The purpose of the System-Building Utility, described in terms of the features of the LISP system that it creates.
- The capabilities that the System-Building Utility provides. These capabilities are described in terms of the differences between a LISP system created with the System-Building Utility and a LISP system created with the VAX LISP `SUSPEND` function.
- The generic procedure for building a LISP system with the System-Building Utility.
- Prerequisites for using the System-Building Utility.

1.1 Features of a User-Built System

The LISP system created by the System-Building Utility consists of a single VMS executable (.EXE) file. When you build a LISP system, you have the following options:

- You can exclude certain portions of Digital-provided VAX LISP code from the image, thus making the image smaller. Reducing the image size reduces the frequency of paging and the requirements for disk space and memory.
- You can incorporate LISP code that you write into the image. When possible, your code is built into read-only sections of the image. The read-only sections can be shareable.
- You can specify the image's entry point; that is, the main function to be executed when the image is invoked.
- You can specify the size of the image's dynamic memory and of its control stack and binding stack.

In addition to these optional features, the user-built system shares the normal features of a VMS executable file:

- You can use the VMS Install Utility to make the read-only portion of the image (including the read-only portion of the code you supply) shareable. Making the image shareable reduces the physical memory requirements when more than one user at a time runs the image.
- You can invoke the image with the DCL command RUN, which does not take LISP-specific qualifiers. (A LISP-specific qualifier is any command line qualifier processed by the LISP image.) Alternatively, you can use the VMS Command Definition Utility to create a new DCL command for invoking the image. The new command can take LISP-specific qualifiers.

Finally, an important feature of user-built systems is that *some* systems may be distributed to CPUs that have no VAX LISP software license. See Section 2.5 for further information on Digital licensing requirements for user-built systems.

1.2 Differences from a Suspended System

An executable image created by the System-Building Utility differs from a system created by the LISP function SUSPEND in several important respects. A call to SUSPEND creates a suspended system with a default file type of .SUS instead of an executable image with a default file type of .EXE. (For information on SUSPEND and on using suspended systems, see the *VAX LISP/VMS Program Development Guide*.)

Before Version 2.2 of VAX LISP, only a suspended system enabled you to capture a modified LISP environment. Since VAX LISP Version 2.2, your system-building options have included two choices:

- An executable image created with the System-Building Utility
- A suspended system created from within a user-built system

This section outlines the differences between the two kinds of systems—executable and suspended—that you can create from within VAX LISP. (The discussion does not apply to a suspended system created from within a user-built executable system.)

The differences are:

- A user-built executable image is a stand-alone system; that is, it can run on a system on which VAX LISP has not been installed. (However, a user-built executable image may require a VAX LISP license to run; see Section 2.5.) In contrast, a suspended system can be resumed only in the presence of the executable file from which it was suspended—in this case, the VAX LISP executable file. Although you can create a suspended system from a user built system, it is not possible to suspend the build of a .EXE file and then resume it.
- A user-built executable image can exclude unneeded Digital-provided code. In contrast, all Digital-provided code is included in the system that results when a suspended system is resumed from VAX LISP.
- In a user-built executable image, multiple users can share read-only portions of both the Digital-provided and user-supplied code; in contrast, they cannot share a suspended system.

A suspended system is useful as a personal, single-user development environment and as a way to save some of the state of your work when you need to interrupt

an interactive LISP session. A suspended system created from within VAX LISP is less useful as:

- A multiuser customized development environment, because user-written code cannot be placed in read-only space and made shareable.
- A delivery vehicle for user-written applications, because the suspended system requires the original executable file in order to resume.

A user-built executable file is more useful for both these purposes, because it is a stand-alone system, because read-only portions of user-written code can be made shareable, and because unneeded Digital-provided code can be excluded.

On the other hand, creating an executable image involves considerably more overhead than creating a suspended system. Also, a suspended system saves the present LISP environment, while a user-built executable file must build the environment from compiled files.

1.3 Generic System-Building Procedure

The system-building procedure has two steps:

1. From within VAX LISP, call the `DEFINE-LISP-SYSTEM` function. Use keyword arguments to specify features of the system. The function `DEFINE-LISP-SYSTEM` creates a VMS command procedure and other files required to build your system.
2. From DCL, execute the command procedure. Commands in the procedure build the executable image file.

You can treat the user-built system like any other VMS executable image. For instance, you can install it to make its read-only portions shareable, and you can define a special DCL command to invoke it.

An example of the generic system-building procedure is:

1. From LISP, call the `DEFINE-LISP-SYSTEM` function with the desired arguments:

   ```
   Lisp>  (define-lisp-system "my-program" ... )
   ```

   This step produces a command file named LISP$BUILD-MY-PROGRAM.COM and a compiled file named LISP$BUILD-MY-PROGRAM.FAS.
2. From DCL, execute the command procedure:

   ```
   $ @LISP$BUILD-MY-PROGRAM
   ```

   This step uses the compiled file (and a number of other special-purpose files) to produce an executable file named MY-PROGRAM.EXE. (This step may produce a number of linker warning messages, which may be ignored.)
3. Optionally, use VMS utilities to make the executable file shareable and/or to define a new DCL command to invoke the executable file. (See Chapter 3 for a description and examples of these steps.)
4. From DCL, invoke the executable image:

   ```
   $ RUN MY-PROGRAM
   ```

   The executable image can also be transferred to and used on another CPU on which Digital-supplied VAX LISP is not installed. Such a transfer may require a Digital software license. (See Section 2.5.)

1.4 Preparing to Use the System-Building Utility

Before using the System-Building Utility, you should make sure that you have access to the necessary components of the Digital-supplied VAX LISP kit and that your account has an adequate quota of disk space, as described in the following sections.

1.4.1 Digital-Supplied Files

The command procedure in the System-Building Utility uses the following four files, which are supplied with the VAX LISP system:

- LISP$LIBRARY:LISP$FASLIB.TLB
- LISP$LIBRARY:LISP$OBJLIB.OLB
- LISP$LIBRARY:LISP$BUILD-VAXLISP.EXE
- LISP$LIBRARY:LISP$BUILD-VAXLISP.CLD

These files must be available to your process. Because only the VAX LISP System-Building Utility uses these files, you can save space by removing them from your system (leaving them on the Digital distribution medium) when you are not using the System-Building Utility.

1.4.2 Disk Space and Memory

When you execute the System-Building Utility's command procedure, it starts a new LISP process from DCL. It also generates a large number of intermediate files that are later deleted and the executable file.

For you to use the System-Building Utility, your account needs a quota of disk space large enough for the command file, the compiled file, and the executable file that the utility creates, plus additional blocks for the intermediate files that the command procedure generates. The number of blocks that these files require depends on options specified with DEFINE-LISP-SYSTEM. Building a full VAX LISP system requires approximately 34,000 blocks.

The virtual memory required to execute the System-Building Utility's command procedure includes the virtual memory required by the new LISP process that it generates. If you use the VAX LISP SPAWN function to execute the command procedure in a subprocess, your account will require process quotas large enough to run two LISP processes simultaneously.

Chapter 2

DEFINE-LISP-SYSTEM Function

This chapter describes the DEFINE-LISP-SYSTEM function, the center of the System-Building Utility. The following sections describe the function and its effects:

- Section 2.1 describes the format of the DEFINE-LISP-SYSTEM function.
- Section 2.2 shows how to specify the names of the executable image file and the files used to build the executable image.
- Section 2.3 shows how to customize the image by specifying files of user-written code to be included in the image, by changing the image's entry point (that is, the function that executes when the image starts), and by either preventing the standard VAX LISP welcome message from being displayed or changing the contents of that message. Section 2.3 also lists restrictions on actions that user-written code can perform as it is loaded.
- Section 2.4 shows how to exclude components of Digital-provided code from the image.
- Section 2.5 shows how to specify whether the image will require a VAX LISP software license for distribution.
- Section 2.6 shows how to specify the default size of the image's dynamic memory and the sizes of the control stack and binding stack.

2.1 Format and Behavior of DEFINE-LISP-SYSTEM

The DEFINE-LISP-SYSTEM function creates a VMS command procedure and returns a string that is the VMS file specification of that procedure. When you execute the command (.COM) file from DCL, an executable image containing a LISP custom system is produced.

The DEFINE-LISP-SYSTEM function also creates a compiled (.FAS) file that the command procedure uses. The command file and the compiled file remain on your system—along with the new executable file—after the system-building procedure ends. The format of the DEFINE-LISP-SYSTEM function is shown below.

DEFINE-LISP-SYSTEM

Format

DEFINE-LISP-SYSTEM *image-name*
&KEY :BUILD-FILE-NAMES :INPUT-FILES
:INIT-FUNCTION :MAIN :HERALD
:EXCLUDE :REQUIRES-LICENSE :MEMORY
:BUILD-MEMORY :CONTROL-STACK-SIZE
:BINDING-STACK-SIZE

Arguments

image-name
A string, symbol, or pathname used to create the name of the executable file and (by default) command file and the compiled file. (See Section 2.2.1.)

:BUILD-FILE-NAMES
A string, symbol, or pathname used to name the command file and the compiled file. (See Section 2.2.2.)

:INPUT-FILES
A file specification or a list of file specifications (symbols, strings, or pathnames). The file type defaults to .FAS or .LSP, with the more recently created file being used in case of a conflict. Code in the files is built into the system. (See Section 2.3.1.)

:INIT-FUNCTION
A symbol or string that names the function that executes after the welcome message (if one is printed) and before the function named by `:MAIN` or the read-eval-print loop. (See Section 2.3.2.)

:MAIN
A symbol or string that names the function that executes after the function named by `:INIT-FUNCTION` and whose return terminates the image. The default value is the VAX LISP read-eval-print loop. A value of `NIL` specifies this default. (See Section 2.3.2.)

:HERALD
The value `T` (the default), `NIL`, or a string specifying whether the standard VAX LISP welcome message, no message, or a user-supplied message should be displayed when the system starts. (See Section 2.3.2.)

:EXCLUDE
A keyword or list of keywords specifying VAX LISP components to be excluded from the system. (See Section 2.4.)

:REQUIRES-LICENSE
The value `T` (the default) or `NIL`, specifying whether the system may run only on CPUs with VAX LISP software licenses. (See Section 2.5.)

:MEMORY
An integer specifying, in 512-byte pages, the default total size of the system's dynamic memory when the executable file is invoked. (See Section 2.6.)

:BUILD-MEMORY
An integer specifying, in 512-byte pages, the total size of dynamic memory used during the build of the executable file. (See Section 2.6.)

:CONTROL-STACK-SIZE
An integer specifying, in 512-byte pages, the size of the system's control stack. (See Section 2.6.)

:BINDING-STACK-SIZE
An integer specifying, in 512-byte pages, the size of the system's binding stack. (See Section 2.6.)

2.2 Naming the Output Files

The System-Building Utility creates three files that remain on your system after the entire procedure is completed. These three files are:

- A command procedure (.COM file) that is created by the DEFINE-LISP-SYSTEM function, then executed to create the customized system
- A compiled file (.FAS file) that is created by the DEFINE-LISP-SYSTEM function, then used by the command procedure to create the customized system
- An executable file (.EXE file) that is the customized system created by the command procedure

In addition to these three files, the System-Building Utility creates (and later deletes) a number of intermediate files that it uses to create the executable file.

The DEFINE-LISP-SYSTEM function's *image-name* argument and :BUILD-FILE-NAMES keyword specify the names of the three permanent output files. Both values can be specified as symbols, strings, or pathnames.

As with all such file specifications in VAX LISP, you can supply a full VMS file specification or just a file name. The system supplies default values for any other components of a full file specification that you do not supply. (See the *VAX LISP/VMS Program Development Guide* or the VAX/VMS documentation set for more information on VMS file specifications.)

2.2.1 Using the Argument *image-name*

The required *image-name* argument names the executable file that the System-Building Utility creates. The file type is .EXE, and the device and directory components, if not specified, are the default device and directory when the DEFINE-LISP-SYSTEM function is called.

For instance, the following two examples have equivalent effects:

```
(define-lisp-system "disk1:[smith.build]expert.exe")
```

and

```
(setf (default-directory) "disk1:[smith.build]")
(define-lisp-system "expert")
```

Both examples direct the System-Building Utility to produce an executable file with the name:

DISK1:[SMITH.BUILD]EXPERT.EXE

By default, the *image-name* argument also names the command procedure and the compiled file that the DEFINE-LISP-SYSTEM function produces. The file name specified by *image-name* is prefixed with LISP$BUILD-, and the file types are .COM and .FAS, respectively. You can supply a value for the keyword :BUILD-FILE-NAMES to override this default.

For example, the preceding calls to the DEFINE-LISP-SYSTEM function produce "build" files named:

DISK1:[SMITH.BUILD]LISP$BUILD-EXPERT.COM

and

DISK1:[SMITH.BUILD]LISP$BUILD-EXPERT.FAS

NOTE

If the value you supply for *image-name* corresponds to a logical name that is currently defined, that logical name is translated and the result is used to form the names of the output files.

2.2.2 Using the :BUILD-FILE-NAMES Keyword

If you want some or all components of the build file specifications to differ from the default, supply a :BUILD-FILE-NAMES argument. For example:

```
(define-lisp-system "disk1:[smith.build]expert.exe"
                    :build-file-names
                       "disk1:[smith.test]expert-test1")
```

or

```
(setf (default-directory) "disk1:[smith.build]")

(define-lisp-system "expert"
                    :build-file-names "[smith.test]expert-test1")
```

Both calls to the DEFINE-LISP-SYSTEM function ultimately produce an executable file named:

DISK1:[SMITH.BUILD]EXPERT.EXE

The build files, however, are named:

DISK1:[SMITH.TEST]EXPERT-TEST1.COM

DISK1:[SMITH.TEST]EXPERT-TEST1.FAS

If you supply a value for the :BUILD-FILE-NAMES keyword but do not specify a device and/or directory, the build files are placed in the same directory as the executable image file.

You can supply just a device and/or a directory with the :BUILD-FILE-NAMES keyword, allowing the file names to be constructed from the *image-name* argument. For example:

```
(define-lisp-system "expert"
                    :build-file-names "disk2:[smith.test]")
```

The two resulting build files are named:

DISK2:[SMITH.TEST]LISP$BUILD-EXPERT.COM

DISK2:[SMITH.TEST]LISP$BUILD-EXPERT.FAS

The disk on which the build files are placed must also have room for a number of other files that are needed to build the system. The total size of these files is about 34,000 blocks for a full VAX LISP system.

The examples in this section illustrate only the name-related arguments to the DEFINE-LISP-SYSTEM function. Since they do not change image contents or behavior in any way, these examples produce an executable image that is functionally identical to the Digital-supplied VAX LISP.

2.3 Customizing the Image

To customize the content and the behavior of the user-built system, use the :INPUT-FILES, :INIT-FUNCTION, :MAIN, and :HERALD keywords. The values of these keywords specify the following items:

- :INPUT-FILES—File(s) of user-written code to be built into the image
- :INIT-FUNCTION—The function that executes after the welcome message and before the function named by :MAIN or the read-eval-print loop
- :MAIN—The function that executes after the function named by :INIT-FUNCTION and whose return terminates the image
- :HERALD—The presence of the standard message "Welcome to VAX LISP, Vx.x," a user-supplied message, or no message when the image starts

2.3.1 Using the Keyword :INPUT-FILES

The :INPUT-FILES keyword specifies one or more files of LISP code to be built into the image. The value of :INPUT-FILES is a file specification or a list of file specifications. A file specification can be a symbol, a string, or a pathname. The files you specify are loaded into the image in the order you list them.

Using the :INPUT-FILES keyword to include user-written code in an image is like using the LOAD function to load code into a running VAX LISP system, with two important differences:

- Code included in a user-built system with the :INPUT-FILES keyword becomes part of the image file; code loaded into a running system with the LOAD function must be in a separate file.
- Portions of the code that you supply with the :INPUT-FILES keyword are placed in read-only sections of the image; for example, parts of compiled function definitions are placed in read-only sections. In contrast, code loaded into a running system with the LOAD function can never be placed in read-only sections.

You can include some user-written development tools in the image to create a customized VAX LISP development environment. For example:

```
(setf (default-directory) "disk1:[smith]")

(define-lisp-system
    "graphics-lisp"
    :input-files '("editor-extensions"
                   "disk2:[jones]graphics-editor"
                   "menu-package"))
```

The image ultimately produced by this call to DEFINE-LISP-SYSTEM contains three files of user-written code.

You can also use :INPUT-FILES to create a customized execute-only system for distribution to other VAX CPUs. (See Section 2.5 for information on execute-only systems.)

The DEFINE-LISP-SYSTEM function provides defaults for any file specification components that are not supplied in the argument. The file type defaults to .FAS or to .LSP. If two files differ only in their file types, :INPUT-FILES uses the most recently created file. In this respect, :INPUT-FILES behaves just like the LOAD function.

Files specified with the :INPUT-FILES function can contain calls to the LOAD function. Thus, one file specified with :INPUT-FILES can actually bring many files into the image; this is a convenient alternative to entering a long list of files with :INPUT-FILES. If you use LOAD in this way, the code included in the system is still placed in read-only space whenever possible.

In general, files specified with :INPUT-FILES can contain any VAX LISP code. However, you should observe the following restrictions and guidelines:

- While they load, the files must not create a static alien structure that contains pointer fields. (However, the files can contain code that creates such structures after the image starts.)
- Any data structures with pointers to them, for example, an array that is the value of a symbol, become part of the image, even if they are not useful. Therefore, be sure that only useful data structures are still accessible when all your code has been executed. Data structures with no pointers to them will be garbage-collected before the image is built.
- When the image starts, certain aspects of its state are lost, in the same way that a suspended system's state is lost when it resumes. In particular, streams that were left open when the system was built are closed, and callout state may be lost. (See the description of the SUSPEND function in the *VAX LISP/VMS Object Reference Manual* for more information.)
- Callout initialization occurs each time the image is run, even if the input files caused callout initialization to occur before the system was built. As a result, the first use of CALL-OUT in the image causes a short delay while callout initialization takes place.

2.3.2 Using the :MAIN, :INIT-FUNCTION, and :HERALD Keywords

In a system that is intended to be a delivery vehicle for a user-built application, you may want either to change the entry point and the welcome message or to perform some computation after the welcome message but before the read-eval-print loop starts.

The value of the :INIT-FUNCTION keyword specifies the function that executes after the welcome message and before either the read-eval-print loop or the function specified by the value of the :MAIN keyword. If you omit the :INIT-FUNCTION keyword or the value of :INIT-FUNCTION is NIL, the value of the :MAIN keyword specifies the function that executes when the image starts.

When the function specified by :INIT-FUNCTION returns, either the read-eval-print loop or the function specified by the value of the :MAIN keyword executes. When the function specified by the value of the :MAIN keyword returns, the image terminates. Therefore, the functions you specify with the :INIT-FUNCTION and :MAIN keywords control the operation of the system.

:INIT-FUNCTION or :MAIN must perform system initialization when they start, and :MAIN must perform any necessary cleanup before it exits. The function specified by the value of the :INIT-FUNCTION keyword will not run if the /COMPILE command line qualifier is present and the compiler has not been excluded. (See Section 3.2.2.3 for further information.) If you omit the :MAIN keyword, or if you specify a value of NIL with the :MAIN keyword, the VAX LISP read-eval-print loop executes and controls the operation of the system.

The values of the :MAIN and :INIT-FUNCTION keywords can be symbols or strings. A string is interpreted as the name of a symbol that names the function. Specify only a single symbol. The symbol must exist in the system that is being built, but it need not be defined in the system in which you are using the DEFINE-LISP-SYSTEM function.

To specify a function in a package that does not exist in the VAX LISP system from which you are calling DEFINE-LISP-SYSTEM, use a string argument to :INIT-FUNCTION or :MAIN. However, the package must exist in the system that results from the DEFINE-LISP-SYSTEM call. For example:

```
(define-lisp-system
    "expert"
    :input-files '("expert-system" "[jones]expert-rules")
    :main "expert-tools:expert-command-loop")
```

The package EXPERT-TOOLS need not exist when this call to DEFINE-LISP-SYSTEM is made, but the files specified with the :INPUT-FILES keyword must create the package EXPERT-TOOLS and define the EXPERT-COMMAND-LOOP function in that package, so that EXPERT-TOOLS:EXPERT-COMMAND-LOOP exists in the new system.

The default package for a function name you specify with the :INIT-FUNCTION or :MAIN keyword depends on whether you use the symbol or string form of the argument. If you use the symbol form, the default package is the current package when the symbol is read. If you use the string form, the default package is always the USER package.

If you use the :MAIN keyword to specify a function, that function processes all the command line qualifiers. If you define a DCL command to invoke the system after having specified a function with the :MAIN keyword, you have to write LISP code to process them. (See Section 3.2.2.3 for more information.)

The value of the :HERALD keyword controls whether the standard VAX LISP welcome message, a user-supplied message, or no message is printed when the image starts. The default value of the symbol T requests the standard message:

```
Welcome to VAX LISP, Vx.x
```

When the value of :HERALD is a string, that string prints as the welcome message. A value of NIL suppresses this message.

The following example demonstrates the use of the :MAIN and :HERALD keywords:

```
(define-lisp-system
   "expert"
   :input-files '("expert-system" "[jones]expert-rules")
   :herald nil
   :main 'expert-command-loop)
```

The image eventually produced by this call includes code from two files. When this image starts, it does not print the standard VAX LISP welcome message, and the function it calls is EXPERT-COMMAND-LOOP. When the value of the EXPERT-COMMAND-LOOP function returns, the image terminates.

2.4 Excluding Digital-Supplied Code

You can use the :EXCLUDE keyword to prevent certain parts of standard VAX LISP from becoming part of your image. When you exclude code from the image, the image takes less space on disk and has smaller memory and run-time requirements.

The value of the :EXCLUDE keyword is a keyword or list of keywords. Each keyword specifies a part of VAX LISP to exclude from your image. Table 2–1 lists the values for the keyword :EXCLUDE. With each value, the table lists the component(s) of Digital-supplied code that are excluded from the image when that value is specified and the approximate amount of disk space saved.

Some components of VAX LISP are automatically excluded from systems defined with :REQUIRES-LICENSE NIL. These components are noted in Table 2–1. (See Section 2.5 for more information.)

Table 2–1: Values for the Keyword :EXCLUDE

Value	Approx. Savings in Blocks	Component Excluded from the Image
:ALIEN	128	The resulting image loses the ability to define new alien data structures.
:BITBLT	128	The BITBLT function, which allows you to alter bitmaps.
:CALLOUT	6912	The Call-Out facility, which lets you call routines written in other languages. Excluding the Call-Out facility precludes the built system from using the DECwindows interface and also excludes: :CLX :DWT :DECW-DEVELOPMENT-ENVIRONMENT
:CLX	4224	A package of LISP routines that give you access to the capabilities of the X Window System without having to call out to external routines or to define non-LISP data structures. The resulting image can still have a programming environment including the Editor and the DECtoolkit (DWT).
:COMPILE-FILE	varies	The VAX LISP COMPILE-FILE function, which is automatically excluded by :REQUIRES-LICENSE NIL. (See Section 2.5.2.)
:COMPILER	2430	The VAX LISP Compiler. This precludes using the VAX LISP functions COMPILE and COMPILE-FILE.

(continued on next page)

Table 2–1 (Cont.): Values for the Keyword :EXCLUDE

Value	Approx. Savings in Blocks	Component Excluded from the Image
:DEBUGGER	1920	The VAX LISP Debugger, and the STEP, TRACE, and INSPECT functions. The VAX LISP Debugger is automatically excluded by :REQUIRES-LICENSE NIL (see Section 2.5.2) and by any of the following values to the keyword :EXCLUDE: :EVAL :MACROS :REPLOOP You cannot use the VAX LISP Debugger as a DECwindows-based utility if you specify :DECWINDOWS, DECW-DEVELOPMENT-ENVIRONMENT, or :CALLOUT as a value for the :EXCLUDE keyword. (See Section 2.5.2.)
:DECW-DEVELOPMENT-ENVIRONMENT	2304	The DECwindows-based functionality including the Inspector, Listener, and DECwindows Help, and the DECwindows interface to the Debugger and Editor. :DECW-DEVELOPMENT-ENVIRONMENT is automatically excluded by :DECWINDOWS.
:DECWINDOWS	4224	The DECwindows interface and DECwindows-based utilities—the Listener and the Inspector. Including :DECWINDOWS as a value for the keyword :EXCLUDE also excludes: :CLX :DWT :DECWINDOWS-DEVELOPMENT-ENVIRONMENT
:DEFINE-LISP-SYSTEM	128	The DEFINE-LISP-SYSTEM function. The resulting image cannot build another LISP system. The DEFINE-LISP-SYSTEM function is automatically excluded by :REQUIRES-LICENSE NIL. (See Section 2.5.2.)
:DEFMACRO	128	The DEFMACRO macro.
:DWT	2560	The DECtoolkit. The resulting image cannot build any DECwindows applications.
:EDITOR	2302	The VAX LISP Editor. This precludes using the ED function or any functionality in the EDITOR package. The VAX LISP Editor is automatically excluded by :REQUIRES-LICENSE NIL (See Section 2.5.2.)

(continued on next page)

Table 2–1 (Cont.): Values for the Keyword :EXCLUDE

Value	Approx. Savings in Blocks	Component Excluded from the Image
:EVAL	6784	The evaluator. Including :EVAL as a value to the keyword :EXCLUDE precludes the resulting image from using the following items: DECwindows interface Read-eval-print loop Compiler Editor Debugging utilities – Debugger – Stepper – Tracer – Inspector
:MACROS	4224 & More	Macroexpander functions. These functions are created by DEFMACRO and used when a macro is invoked from interpreted code and when the macro is compiled. Since macro invocations in compiled code are fully expanded, a system containing only compiled code needs no macroexpanders.
:ORPHAN-SYMBOLS	Varies	VAX LISP-created symbols that are never referenced. (These symbols exist as a result of building the system or excluding parts of VAX LISP, but they have no further use.) User-created symbols are not affected. The resulting image is identical in functionality to one that does not exclude :ORPHAN-SYMBOLS, but takes up less space. However, it takes longer to build.
:RANDOM	128	Random number generators.
:REPLOOP	1024	The VAX LISP read-eval-print loop. If you exclude the read-eval-print loop, you must use the :MAIN keyword to specify a function to execute when the image starts. The read-eval-print loop is excluded by :REQUIRES-LICENSE NIL. (See Section 2.5.2.)
:SORT	128	The SORT and STABLE-SORT functions.
:SUSPEND	128	The SUSPEND function. The SUSPEND function is excluded by :REQUIRES-LICENSE NIL. (See Section 2.5.2.)

(continued on next page)

Table 2–1 (Cont.): Values for the Keyword :EXCLUDE

Value	Approx. Savings in Blocks	Component Excluded from the Image
:TRANSCENDENTAL	128	The transcendental math functions: EXP LOG SQRT CIS SIN ASIN SINH ASINH COS ACOS COSH ACOSH TAN ATAN TANH ATANH (Space savings may be greater in future releases.)
:UIS	1152	The VAX LISP UIS graphics functionality. This precludes using any functionality in the UIS package, but not in the CLX graphics interface.
:VMS-DEBUG	270	The VMS-DEBUG function. (See the *VAX LISP/VMS System Access Guide* for a description of the VMS-DEBUG function.) The VMS-DEBUG function is excluded by :REQUIRES-LICENSE NIL. (See Section 2.5.2.)
:WSSTREAM	640	The VAX LISP window stream facility. (See Chapter 4 of *VAX LISP Implementation and Extensions to Common LISP*.)

2.5 Making an Execute-Only System

Since the System-Building Utility can create a complete VAX LISP programming environment, the distribution of some user-built systems is restricted. The basic principle is:

- If a user-built system can be used for creating new VAX LISP programs, its use on another CPU requires a valid VAX LISP software license. Such a system is called a development system in this manual.
- If a user-built system is not useful as a VAX LISP programming environment, it may be transferred to another CPU without a VAX LISP software license. Such a system is called an execute-only system in this manual.

The value of the keyword REQUIRES-LICENSE determines whether a user-built system is a development system and whether the user-built system can be transferred to a CPU without a VAX LISP software license.

2.5.1 Development Systems

If the value of the keyword :REQUIRES-LICENSE is T (the default), you may not use a user-built system without a valid VAX LISP software license. This requirement holds no matter which components of VAX LISP you have excluded with the keyword :EXCLUDE.

An example of a development system is the user-built expert system created by the following call:

```
(define-lisp-system
    "expert"
    :input-files '("expert-system" "[jones]expert-rules")
    :herald nil
    :main 'expert-command-loop
    :exclude '(:editor :uis))
```

The file EXPERT.EXE that ultimately results from this form is created with the value T for :REQUIRES-LICENSE. EXPERT.EXE may not legally be used on a CPU that does not have a valid VAX LISP software license.

2.5.2 Execute-Only Systems

To make the above user-built system execute-only, you would write the following:

```
(define-lisp-system
    "expert"
    :input-files '("expert-system" "[jones]expert-rules")
    :herald nil
    :main 'expert-command-loop
    :exclude '(:editor :uis)
    :requires-license nil)
```

Because the value of the keyword :REQUIRES-LICENSE is NIL, this system may be freely distributed.

If the value of :REQUIRES-LICENSE is NIL, the user-built system will not contain the following components of Digital-provided code. The components correspond to the :EXCLUDE keyword in parentheses, when such a keyword exists:

- The VAX LISP debugging facilities (:DEBUGGER and :VMS-DEBUG)
- The VAX LISP Editor (:EDITOR)
- The VAX LISP System-Building Utility (:DEFINE-LISP-SYSTEM)
- The VAX LISP function COMPILE-FILE (:COMPILE-FILE)
- The VAX LISP read-eval-print loop :REPLOOP
- The VAX LISP function SUSPEND (:SUSPEND)
- The VAX LISP functions TIME, DESCRIBE, INSPECT, DISASSEMBLE, DRIBBLE, TRACE, STEP, ROOM, APROPOS, and APROPOS-LIST

If you have not already excluded these components with the keyword :EXCLUDE, DEFINE-LISP-SYSTEM with :REQUIRES-LICENSE NIL will exclude them. While DEFINE-LISP-SYSTEM is executing, it displays warnings that identify the VAX LISP components being excluded because :REQUIRES-LICENSE is NIL.

NOTE

To create a user-built execute-only system, you *must* specify :REQUIRES-LICENSE NIL no matter which system components you have excluded with :EXCLUDE.

2.6 Specifying Memory Requirements

The :MEMORY keyword specifies the default size of a user-built system's dynamic memory; it is equivalent to the LISP command's /MEMORY qualifier. The value of the :MEMORY keyword is the total size of LISP's dynamic memory, in 512-byte pages. The default size is 5000 pages; the minimum size is 2000 pages.

You can use the ROOM function to find out how much dynamic memory your system is using at any time. VAX LISP's dynamic memory space is divided into sections known as areas. Use the ROOM function to print information about dynamic space. By default, a garbage collection occurs whenever 50% or more of the dynamic memory has been consumed. You can change this ratio by using SETF with the DYNAMIC-SPACE-RATIO function.

To determine a value for the :MEMORY keyword, build a trial system with the default size; then, start the system and use the ROOM function to see how much dynamic memory the system consumes. As you use the system, use ROOM from time to time to see how quickly dynamic memory is consumed. Also note the frequency of garbage collection. Frequent or nearly continuous garbage collection activity indicates insufficient memory and a larger value for the :MEMORY keyword is in order.

When you set a default value for dynamic memory, remember the following points:

- A smaller dynamic memory size leads to faster but more frequent garbage collections.
- A larger dynamic memory size leads to slower but less frequent garbage collections. A large dynamic memory also requires more virtual memory and may cause more paging.

The default dynamic memory size you specify can be overridden when your image is invoked, if you define a DCL command with the /MEMORY qualifier to start the image. (See Section 3.2.2 for more information.)

The :BUILD-MEMORY keyword specifies the size, in 512-byte pages, of the amount of memory used during the build. A normal build requires 50,000 pages. If you specify fewer pages, the build will take longer.

The :CONTROL-STACK-SIZE and :BINDING-STACK-SIZE keywords specify the size, in 512-byte pages, of the image's control and binding stacks. The default for :CONTROL-STACK-SIZE is 185 pages; the minimum size is 32 pages. The default for :BINDING-STACK-SIZE is 62 pages; the minimum size is 16 pages.

The sizes required for the control and binding stacks are a rough function of the complexity of your system. Highly recursive systems require larger control stacks.

One way to determine whether you should increase stack sizes is to build a test system with the default stack sizes. If the stacks overflow in normal operation, increase the sizes. (Control stack overflows are treated as errors; binding stack overflows are handled automatically, and program execution continues.)

Chapter 3

Working with User-Built Systems

A user-built system can be treated like any other VMS executable image. This chapter describes:

- Installing the image to allow sharing of read-only code
- Invoking the image

3.1 Installing the Image

If you intend the user-built system to be run concurrently by several processes, it is worthwhile to install the image as shareable. Installing an image improves performance and reduces the requirements for physical memory in the following ways:

- An installed image is permanently "open," which means that directory information on the image file remains permanently resident in memory. An open image does not require the usual directory search to locate the file.
- When an image is installed as shareable, only one copy of its read-only sections need be in physical memory. These global sections can be accessed by more than one user at a time.

The VMS Install Utility installs an executable image, allowing you to make the image open as well as shareable. This utility is described in the VAX/VMS documentation set: *VMS Install Utility Manual* and *Guide to Setting Up a VMS System*.

To install the image:

1. Invoke the VMS Install Utility.
2. Execute the CREATE command with the /SHARED qualifier. (/SHARED also installs the image as permanently open.)
3. Exit the VMS Install Utility.

For example, to make the user-built system MY-LISP.EXE shareable, you would do the following:

1. Invoke the VMS Install Utility. If you are adding a new image to an established VMS system, it is probably most convenient to invoke the utility in interactive mode. To do so, type:

```
$ INSTALL := $SYS$SYSTEM:INSTALL
$ INSTALL/COMMANDMODE
```

2. At the VMS Install Utility prompt, execute the CREATE command with the /SHARED qualifier. The parameter is the name of the executable file. If you omit the device and directory, they default to those indicated by the logical name SYS$SYSTEM. The default file type is .EXE.

```
INSTALL>  CREATE/SHARED DISK1:[SMITH.BUILD]MY-LISP
```

   This step establishes global sections of the specified executable file's read-only contents and makes it permanently open. You need not specify the /OPEN qualifier, since an image installed as shareable is also installed as permanently open.

3. If you later want to delete the global sections associated with the image, use the VMS Install Utility's DELETE command.

```
INSTALL>  DELETE DISK1:[SMITH.BUILD]MY-LISP
```

   This command removes any global sections created for the image MY-LISP.EXE. The image file itself is not affected by this operation.

4. Type either of the following commands to exit the VMS Install Utility:

```
INSTALL> EXIT
```

```
INSTALL>  Ctrl/Z
```

See the VMS documentation set for further information on the VMS Install Utility, especially for other qualifiers to the Install command CREATE.

3.2 Invoking the Image

There are four ways to invoke a user-built system:

- The DCL RUN command
- A user-defined foreign command
- A user-defined DCL command
- The Digital-defined DCL LISP command

If you use RUN or a foreign command, you cannot use LISP-specific command-line entities; that is, you cannot use command qualifiers, parameters, or keywords processed by the LISP image. With a specially defined DCL command—either user-defined or Digital-defined—you can use LISP-specific command-line entities with certain restrictions.

This section discusses these four methods of invoking a user-built system. For each method, the section identifies the restrictions, if any, on the use of LISP-specific command-line entities.

3.2.1 Using RUN or a Foreign Command

You can invoke a user-built LISP system like any VMS executable image with either a RUN command or a user-defined foreign command. For example:

```
$ RUN MY-LISP
```

or

```
$  MYLISP := $DISK1:[SMITH.BUILD]MY-LISP
$  MYLISP
```

See the *VMS DCL Dictionary* for more information on the RUN command and on the facility for defining foreign commands.

When you use RUN or a foreign command to invoke a LISP image, the LISP image cannot call back to DCL to check for the presence of qualifiers or other command-line entities. For this reason, you cannot use any LISP-specific qualifiers, such as /MEMORY, with RUN or a foreign command.

3.2.2 Defining a DCL Command

You can use the VMS Command Definition Utility (CDU) to create a new DCL command that invokes a user-built system. This command can be used with or without LISP-specific command-line entities. The CDU is described in the *VMS Command Definition Utility Manual*. This section assumes that you are generally familiar with DCL command definition and with writing code to retrieve command string information.

The procedure for creating a DCL command is:

1. Write a command language definition (CLD) file that defines the command and its qualifiers, parameters, and keywords, if any.
2. Be sure that the image to be invoked by the command contains code that calls back to DCL to check for the presence of each such qualifier, parameter, or keyword, then responds appropriately.
3. Use the DCL command SET COMMAND to add your CLD file to the system command table or to an individual process command table.
4. Log out and log in again to make the command available. (This step is not necessary if you used SET COMMAND to add the CLD file to your own process command table.)

The Digital-provided VAX LISP image contains code that checks for and processes the Digital-defined qualifiers to the LISP command. (The *VAX LISP/VMS Program Development Guide* lists and describes these qualifiers.) A user-built system contains all or part of the Digital-provided VAX LISP. Therefore, in defining a DCL command to invoke a LISP image, you face some restrictions when choosing qualifier names, and you may have to redefine some Digital-defined qualifiers. In general terms, the restrictions are:

- If your user-built system contains the Digital-provided code that checks for a given qualifier *and* you want to use that qualifier when invoking the system, your CLD file *must* define the qualifier exactly as it is defined in LISP$SYSTEM:LISP.CLD, the file that defines the DCL LISP command. If you do not want to use the qualifier, your CLD file need not define it.
- If your user-built system does not contain the Digital-provided code that checks for a given qualifier, you can define that qualifier in any way you like. However, your image must include *user-written* code to check for the qualifier. (See Sections 3.2.2.2 and 3.2.2.3 to determine whether your image contains code that checks a given qualifier.)
- Qualifiers with names that are not defined by Digital as qualifiers to the LISP command—and all other command-line entities—can be defined in your CLD file in any way you like. You must, however, include user-written code in your image to check for each such entity.

To check for the presence of command-line entities, use the VAX LISP functions COMMAND-LINE-ENTITY-P and COMMAND-LINE-ENTITY-VALUE. These functions provide interfaces to the VMS utility routines CLI$PRESENT and CLI$GET_VALUE, which retrieve information about the command string that invoked an image. For further information:

- The *VAX LISP/VMS System Access Guide* describes the two VAX LISP functions.
- The *VMS Utility Routines Manual* describes the two VMS routines.

The remainder of this section outlines the specific procedures for writing a DCL command to invoke a LISP image. It also identifies the circumstances under which you are limited in defining command qualifier names.

3.2.2.1 Defining a Command Without Qualifiers

As an example of defining a DCL command without qualifiers or other command-line entities, you might write the following code in a file called MY-COMMANDS.CLD. This code defines a DCL command named EXPERT, which invokes the image EXPERT.EXE. The device and directory of this file are indicated by the logical name EXPERT$SYSTEM.

```
DEFINE VERB EXPERT
    IMAGE "EXPERT$SYSTEM:EXPERT"
```

Once the file MY-COMMANDS.CLD has been added to the appropriate command table, you can invoke EXPERT.EXE by typing EXPERT at the DCL prompt.

Because the command EXPERT has no qualifiers, parameters, or keywords, the image EXPERT.EXE need not contain code that processes DCL command-line entities.

3.2.2.2 Defining the Qualifiers /MEMORY, /RESUME, and /CSTACK

A user-built LISP image *always* contains the Digital-provided code that checks for and processes the /MEMORY, /RESUME, and /CSTACK qualifiers in the command that invoked the image. You cannot alter the action of these qualifiers. Therefore, if you want your DCL command to take qualifiers named /MEMORY, /RESUME, or /CSTACK, you must define them in your CLD file exactly as follows:

```
QUALIFIER MEMORY, NONNEGATABLE, VALUE(TYPE=$NUMBER,REQUIRED)

QUALIFIER RESUME, VALUE(TYPE=$INFILE,REQUIRED)

QUALIFIER CSTACK, NONNEGATABLE, VALUE(TYPE=$NUMBER,REQUIRED)
```

For example, a CLD file that defines the command EXPERT with the qualifiers /MEMORY, /RESUME, and /CSTACK would look like this:

```
DEFINE VERB EXPERT
    IMAGE "EXPERT$SYSTEM:EXPERT"
    QUALIFIER MEMORY, NONNEGATABLE, VALUE(TYPE=$NUMBER,REQUIRED)
    QUALIFIER RESUME, VALUE(TYPE=$INFILE,REQUIRED)
    QUALIFIER CSTACK, NONNEGATABLE, VALUE(TYPE=$NUMBER,REQUIRED)
```

Since these three qualifiers are processed by Digital-provided LISP code, they behave in exactly the same way with the command EXPERT as they do with the command LISP. That is, /MEMORY allows you to invoke EXPERT.EXE with a specified amount of dynamic memory, /RESUME allows you to resume a suspended system that was created from within EXPERT.EXE, and /CSTACK allows you to invoke EXPERT.EXE with a specified size for the default control stack.

If you do not define these qualifiers in your CLD file, you cannot use them with the command EXPERT. You can, of course, continue to use them with the LISP command.

3.2.2.3 Defining Other Digital-Defined Qualifiers

In addition to /MEMORY, /RESUME, and /CSTACK, the LISP command takes 10 qualifiers; they are processed by the VAX LISP function that is the default value of the `DEFINE-LISP-SYSTEM` keyword `:MAIN` or the value of the `DEFINE-LISP-SYSTEM` keyword `:INIT-FUNCTION`. That is, the default entry point in a user-built system processes the following compiler-related qualifiers:

/COMPILE	/OPTIMIZE
/LISTING	/OUTPUT_FILE
/MACHINE_CODE	/WARNINGS

and the following general-purpose qualifiers:

/ERROR_ACTION	/INTERACTIVE
/INITIALIZE	/VERBOSE

If you have not specified a value for `INIT-FUNCTION` or `:MAIN` in your call to `DEFINE-LISP-SYSTEM` and you want to use any of these qualifiers when you invoke the system, you *must* define the qualifiers and their parameters exactly as they are defined in LISP$SYSTEM:LISP.CLD. In addition, the VAX LISP Compiler must be present in your image for the entry point to process the compiler-related qualifiers.

NOTE

Except for the definitions of /MEMORY and /RESUME, the qualifier definitions in LISP$SYSTEM:LISP.CLD are not supported by Digital and may not be upward-compatible between versions.

For example, the following CLD file defines the command EXPERT with the qualifier /INITIALIZE:

```
DEFINE VERB EXPERT
    IMAGE "EXPERT$SYSTEM:EXPERT"
    QUALIFIER INITIALIZE, VALUE(TYPE=$INFILE,REQUIRED,LIST)
```

The qualifier /INITIALIZE is defined as it is in LISP$SYSTEM:LISP.CLD. If EXPERT.EXE contains the default entry point, this qualifier behaves in the same way with the command EXPERT as it does with the command LISP.

If you have specified a value for `:MAIN` in your call to `DEFINE-LISP-SYSTEM`, the Digital-provided code for processing the qualifiers other than /MEMORY, /RESUME, and /CSTACK is inaccessible in your image. You can then define any of these qualifier names in any way you like. You must, however, include user-written code in your image to check for and process each such qualifier that you define.

3.2.2.4 Defining Other Command-Line Entities

Apart from the Digital-defined qualifiers to LISP, you can define any command-line entity—qualifier, parameter, or keyword—in any way you like. For defining such entities, the values of `:INIT-FUNCTION` and `:MAIN` are irrelevant. However, your image must always include user-written code for processing all such entities.

3.2.2.5 Summary of Restrictions on Command Definition

The restrictions on defining command-line entities for a user-defined DCL command to invoke a LISP image are:

- The qualifiers /MEMORY, /RESUME, and /CSTACK are always processed by Digital-provided code. To use these qualifiers, you must define them as they are defined in LISP$SYSTEM:LISP.CLD.
- The VAX LISP default entry-point function processes the remaining qualifiers to the command LISP. If your image includes the default entry point and you want to use these qualifiers, you must define them as they are defined in LISP$SYSTEM:LISP.CLD. Note that several of these qualifiers require the presence of the VAX LISP Compiler in the image.

For example, the following CLD file defines the command EXPERT for invoking the image EXPERT$SYSTEM:EXPERT.EXE. (Assume that EXPERT.EXE contains the default entry point.)

```
DEFINE VERB EXPERT
    IMAGE "EXPERT$SYSTEM:EXPERT"
    PARAMETER P1, PROMPT="FILE(S)", VALUE(TYPE=$INFILE,LIST)
    QUALIFIER MEMORY, NONNEGATABLE, VALUE(TYPE=$NUMBER,REQUIRED)
    QUALIFIER INITIALIZE, VALUE(TYPE=$INFILE,REQUIRED,LIST)
    QUALIFIER QUICK_TRAVERSE, DEFAULT
```

The command EXPERT takes one parameter and three qualifiers.

- The parameter P1 is to be processed by code you supply. You can define this parameter in any way you like. In this example, the command EXPERT prompts the user for one or more input file names.
- The qualifier /MEMORY is to be processed by Digital-provided code. You must define the qualifier as shown.
- Because the entry point to EXPERT.EXE is the default, the qualifier /INITIALIZE is to be processed by Digital-provided code. You must define the qualifier as shown.

  If the entry point is a function you supply, it must contain code to process /INITIALIZE. In this case, the qualifier could be defined in any way you like.
- The qualifier /QUICK_TRAVERSE is to be processed by code you supply. You can define this qualifier in any way you like.

3.2.3 Using the Digital-Defined Command LISP

With certain restrictions, you can use the Digital-defined LISP command—along with any of its qualifiers—to invoke a user-built system. The major restriction on using the LISP command is that your executable file must be named LISP.EXE and must be in a directory pointed to by the logical name LISP$SYSTEM. LISP.EXE is the name of the Digital-supplied VAX LISP executable image, and the LISP command looks in LISP$SYSTEM for LISP.EXE.

NOTE

If you name your image file the same as the Digital-provided image file, take care that you do not accidentally delete the Digital-provided image file.

For example, if you use VAX LISP on a terminal instead of a workstation, and you want to dispense with the graphics functionality and the DECwindows interface in your development environment, you could write:

```
(define-lisp-system "disk1:[smith.build]lisp.exe"
    :exclude '(:uis :decwindows))
```

The executable image that eventually results from this form contains all the functionality of the Digital-supplied VAX LISP except the graphics functionality and the DECwindows interface. Since the image has the same name as the Digital-supplied VAX LISP executable image, you can use the LISP command—with any of its qualifiers—to invoke the image you create.

Before you can use the LISP command to invoke your image, you must redefine LISP$SYSTEM to point to DISK1:[SMITH.BUILD] so that the LISP command can locate the image. However, LISP$SYSTEM also contains other Digital-provided files (such as the documentation string text library, LISPDOC.TLB) that VAX LISP needs in order to run. To avoid copying these files into DISK1:[SMITH.BUILD], you can define the logical name LISP$SYSTEM to be a search list:

```
$  SHOW LOGICAL LISP$SYSTEM
   "LISP$SYSTEM" = "SYS$SYSROOT:[VAXLISP]" (LNM$SYSTEMTABLE)
$ DEFINE LISP$SYSTEM DISK1:[SMITH.BUILD],SYS$SYSROOT:[VAXLISP]
```

The LISP command now starts the image LISP.EXE in DISK1:[SMITH.BUILD]. LISP looks in SYS$SYSROOT:[VAXLISP] for files it needs but cannot find in DISK1:[SMITH.BUILD].

Restrictions on using the LISP command qualifiers are listed below under the set of LISP qualifiers to which they apply.

- The qualifiers /MEMORY, /RESUME, and /CSTACK and their respective parameters:

  No further restrictions. Note that you need not create your own CLD file to use these two qualifiers with the Digital-defined LISP command.

- The qualifiers /INITIALIZE, /VERBOSE, /INTERACTIVE, and /ERROR_ACTION and their respective parameters:

  RESTRICTION: Your image must include either the default entry point or user-written code that processes these qualifiers.

- The qualifiers /COMPILE, /OPTIMIZE, /LISTING, /MACHINE_CODE, /OUTPUT_FILE, and /WARNINGS and their respective parameters:

  RESTRICTIONS: Your image must include either the default entry point or user-written code that processes these qualifiers. In either case, you must not exclude the VAX LISP Compiler from your image by supplying the `:COMPILER` keyword with `:EXCLUDE`.

Index

F

G

H

I

L

M

O

P

Q

R

S

T

U

V

W

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Alaska, Hawaii, or New Hampshire	603–884–6660	Digital Equipment Corporation P.O. Box CS2008 Nashua NH 03061
Rest of U.S.A. and Puerto Rico[1]	800–DIGITAL	
Canada	800–267–6219 (for software documentation) 613–592–5111 (for hardware documentation)	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: Direct Order Desk
Internal orders (for software documentation)	—	Software Supply Business (SSB) Digital Equipment Corporation Westminster MA 01473
Internal orders (for hardware documentation)	DTN: 234–4323 508–351–4323	Publishing & Circulation Services (P&CS) NRO3–1/W3 Digital Equipment Corporation Northboro MA 01532

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

Reader's Comments

VAX LISP/VMS System-Building Guide
AA–KM37B–TE

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

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

What I like best about this manual: _____

_____

What I like least about this manual: _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____

My additional comments or suggestions for improving this manual:

_____

_____

_____

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

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

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

10/87

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

digital™

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

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

POSTAGE WILL BE PAID BY ADDRESSEE

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

— — Do Not Tear — Fold Here — — — — — — — — — — — — — — — — — —

Cut Along Dotted Line