VAX LISP Implementation and Extensions to Common LISP

Order Number: AA–MK70A–TE

July 1989

This manual describes VAX LISP implementation-dependent language features and extensions to Common LISP. The information in this manual applies to both VMS and ULTRIX versions of VAX LISP, except where specified otherwise.

Revision/Update Information:	This is a new manual.
Operating System and Version:	VMS Version 5.1 or ULTRIX 3.0
Software Version:	VAX LISP Version 3.0

digital equipment corporation
maynard, massachusetts

July 1989

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

MLO-S832

This document was prepared using VAX DOCUMENT, Version 1.1.

Contents

Preface

VAX LISP is an extended implementation of Common LISP. The Common LISP language is described in *Common LISP: The Language*.* The *VAX LISP Implementation and Extensions to Common LISP* manual describes the ways in which VAX LISP differs from Common LISP.

Some Common LISP language elements are not fully specified in *Common LISP: The Language*. This manual describes VAX LISP's implementation-specific details. VAX LISP extends the Common LISP language in some areas, such as input and output and the definition of window streams. This manual describes VAX LISP's extensions to Common LISP.

Intended Audience

This manual is for programmers with a working knowledge of LISP. Detailed knowledge of VMS is helpful but not essential; familiarity with the *Introduction to VMS* is recommended. Detailed knowledge of ULTRIX–32 is helpful but not essential.

Some sections of this manual require more extensive understanding of the operating system. In such sections, you are directed to the appropriate manual(s) for additional information.

Structure

This manual is organized as follows:

- Chapter 1, VAX LISP Implementation Notes, describes the features of LISP that are defined by or are dependent on the VAX implementation of Common LISP.
- Chapter 2, VAX LISP Extensions to I/O, describes VAX LISP extensions to the Common LISP I/O system.
- Chapter 3, Window Streams, explains how to use the VAX LISP window streams facility.
- Chapter 4, Pretty-Printing and Using Extensions to FORMAT, explains how to use the VAX LISP pretty-printer.
- Chapter 5, Error Handling, describes the VAX LISP error-handling facility.

* Guy L. Steele, Jr., *Common LISP: The Language*, Digital Press (1984), Burlington, Massachusetts.

Associated Documents

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

- *VAX LISP/VMS Program Development Guide*
- *VAX LISP/VMS System Access Guide*
- *VAX LISP/VMS System-Building Guide*
- *VAX LISP/VMS DECwindows Programming Guide*
- *VAX LISP Editor Programming Guide*
- *VAX LISP/VMS Interface to VWS Graphics*
- *Introduction to VMS*
- *VMS DCL Dictionary*
- *VMS System Services Reference Manual*
- *VMS I/O User's Reference Manual: Part I*
- *VMS RTL Library (LIB$) Manual*

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

- *VAX LISP/ULTRIX User's Guide*
- *VAX LISP/ULTRIX System Access Guide*
- *VAX LISP/ULTRIX System-Building Guide*
- *ULTRIX–32 Programmer's Manual*
- *ULTRIX–32 Supplementary Documentation*

The following documents are always relevant:

- *Common LISP: The Language*
- *VAX Architecture Handbook*

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

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*}

Convention	Meaning
{ }*	In function and macro format specifications, braces followed by an asterisk enclose elements that are considered one unit of code, which can be repeated zero or more times. For example: *{keyword value}**
&OPTIONAL	In function and macro format specifications, the word &OPTIONAL indicates that the arguments that follow it are optional. For example: PPRINT *object* &OPTIONAL *stream* Do not specify &OPTIONAL when you invoke a function or macro whose definition includes &OPTIONAL.
&REST	In function and macro format specifications, the word &REST indicates that an indefinite number of arguments may appear. For example: CALL-OUT *external-routine* &REST *routine-arguments* Do not specify &REST when you invoke a function or macro whose definition includes &REST.
&KEY	In function and macro format specifications, the word &KEY indicates that keyword arguments are accepted. For example: COMPILE-FILE *input-pathname* &KEY :LISTING :MACHINE-CODE :OPTIMIZE :OUTPUT-FILE :VERBOSE :WARNINGS Do not specify &KEY when you invoke a function or macro whose definition includes &KEY.
. . .	A horizontal ellipsis in a format specification means that the element preceding the ellipsis can be repeated. For example: *function-name . . .*
. . .	A vertical ellipsis in a code example indicates that all the information that the system would display in response to the function call is not shown; or, that all the information a user is to enter is not shown.
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.

Convention	Meaning
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

VAX LISP Implementation Notes

VAX LISP is an implementation of LISP that is based on Common LISP as described in *Common LISP: The Language*. This chapter describes how implementation-dependent aspects of Common LISP are implemented on the VMS and ULTRIX–32/32w operating systems. This chapter does not describe all the implementation differences between VAX LISP/VMS (VAX LISP as implemented on VMS) and VAX LISP/ULTRIX (VAX LISP as implemented on ULTRIX). For a complete list of such differences, see the *VAX LISP Release Notes*. These release notes are on line in the file SYS$HELP:LISP*nnn*.RELEASE_NOTES (on VMS systems) or /usr/lib/vaxlisp/lisp*nnn*.mem (on ULTRIX systems), where *nnn* is the VAX LISP version number. For example, LISP030.RELEASE_NOTES or lisp030.mem is the file containing the release notes for Version 3.0.

Most of the information in this chapter refers to subjects that *Common LISP: The Language* refers to as implementation dependent. The purpose of this chapter is to clarify the implementation specifics for the following topics:

- Data representation
- The garbage collector
- Input and output
- Interrupt functions (on VMS systems only) and keyboard functions that execute asynchronously when you type a control character
- Functions and macros

NOTE

VAX LISP/VMS supports only symbols that are in the packages named `COMMON-LISP`, `VAX-LISP`, `EDITOR`, `CLX`, `DWT`, and `UIS`. VAX LISP/ULTRIX supports only symbols that are in the packages named `COMMON-LISP`, `VAX-LISP`, `CLX`, and `DWT`.

1.1 Data Representation

Common LISP defines the data types implemented in VAX LISP but Common LISP does not define implementation-dependent information related to the data types. This section provides data type information specific to VAX LISP. Complete descriptions of data types are provided in *Common LISP: The Language*. The following data types require VAX LISP implementation information:

- Numbers
- Characters

- Arrays
- Strings

1.1.1 Numbers

Sections 1.1.1.1 through 1.1.1.3 explain how VAX LISP implements the integer, floating-point, and complex number data types.

1.1.1.1 Integers

Common LISP defines two subtypes of integers: fixnums and bignums. In VAX LISP, the integers in the range –2**28 to 2**28–1 are represented as fixnums; integers not in the fixnum range are represented as bignums. VAX LISP stores bignums as two's complement bit sequences.

In VAX LISP, the EQ function returns T when it is called with two fixnums with the same value.

The values of the Common LISP integer constants are implementation dependent. The names of the constants and the corresponding VAX LISP values follow:

```
MOST-POSITIVE-FIXNUM     268435455
MOST-NEGATIVE-FIXNUM    -268435456
```

NOTE

The range of integers represented as fixnums has been reduced by half from the old range of –2**29 to 2**29–1.

Descriptions of these constants are provided in *Common LISP: The Language.*

1.1.1.2 Floating-Point Numbers

Common LISP defines the following types of floating-point numbers:

- Short floating-point numbers
- Single floating-point numbers
- Double floating-point numbers
- Long floating-point numbers

In VAX LISP, these four types are implemented with VAX floating data types.

Table 1–1 lists the types of Common LISP floating-point numbers, the corresponding VAX data types, and the number of bits allocated for the exponent and significand of each floating-point type. For information on the VAX floating data types, see the *VAX Architecture Handbook*.

Table 1–1: VAX LISP Floating-Point Numbers

Common LISP Type	VAX Type	Exponent	Significand
SHORT-FLOAT	F_floating	8	24
SINGLE-FLOAT	F_floating	8	24
DOUBLE-FLOAT	G_floating	11	53
LONG-FLOAT	H_floating	15	113

On ULTRIX Systems

If your system does not have G and H floating-point instructions, see the *ULTRIX–32 Programmer's Manual* Binder IIIA "System Managers" for information on how to configure your system to use the g/h floating-point emulator.

The values of the Common LISP floating-point constants are implementation dependent. You can use the values of these constants to compare the range of values and the degrees of precision of the VAX LISP floating-point types. Table 1–2 lists the names of the constants and provides the actual hexadecimal values and the decimal approximations for VAX LISP.

Table 1–2: Floating-Point Constants

Constant	Hexadecimal Representation	Approximate Decimal Value
DOUBLE-FLOAT-EPSILON	00000000 00003CC0	1.11d-16
DOUBLE-FLOAT-NEGATIVE-EPSILON	00010000 00003CB0	5.55d-17
LEAST-NEGATIVE-DOUBLE-FLOAT	00000000 00008010	-5.5d-309
LEAST-NEGATIVE-LONG-FLOAT	00000000 00000000 00000000 00008001	-8.41L-4933
LEAST-NEGATIVE-SHORT-FLOAT	00008080	-2.94e-39
LEAST-NEGATIVE-SINGLE-FLOAT	00008080	-2.94e-39
LEAST-POSITIVE-DOUBLE-FLOAT	00000000 00000010	5.56d-309
LEAST-POSITIVE-LONG-FLOAT	00000000 00000000 0000000C 00000001	8.41L-4933
LEAST-POSITIVE-SHORT-FLOAT	00000080	2.94e-39
LEAST-POSITIVE-SINGLE-FLOAT	00000080	2.94e-39
LONG-FLOAT-EPSILON	00000000 00000000 00000000 00003F90	9.63L-35
LONG-FLOAT-NEGATIVE-EPSILON	00010000 00000000 00000000 00003F8F	4.81L-35
MOST-NEGATIVE-DOUBLE-FLOAT	FFFFFFFF FFFFFFFF	-8.99d307
MOST-NEGATIVE-LONG-FLOAT	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF	-5.95L4931
MOST-NEGATIVE-SHORT-FLOAT	FFFFFFFF	-1.70e38
MOST-NEGATIVE-SINGLE-FLOAT	FFFFFFFF	-1.70e38
MOST-POSITIVE-DOUBLE-FLOAT	FFFFFFFF FFFF7FFF	8.99d307
MOST-POSITIVE-LONG-FLOAT	FFFFFFFF FFFFFFFF FFFFFFFF FFFF7FFF	5.95L4931
MOST-POSITIVE-SHORT-FLOAT	FFFF7FFF	1.70e38
MOST-POSITIVE-SINGLE-FLOAT	FFFF7FFF	1.70e38
SHORT-FLOAT-EPSILON	00003480	5.96e-8
SHORT-FLOAT-NEGATIVE-EPSILON	00013400	2.98e-8
SINGLE-FLOAT-EPSILON	00003480	5.96e-8
SINGLE-FLOAT-NEGATIVE-EPSILON	00013400	2.98e-8

Descriptions of these constants are provided in *Common LISP: The Language.*

Common LISP allows an implementation to define a floating-point minus zero. In VAX LISP, floating-point minus zero does not exist.

1.1.1.3 Complex Numbers

VAX LISP Version 3.0 supports complex numbers of types RATIONAL and FLOAT, with subtypes SINGLE-FLOAT, DOUBLE-FLOAT, and LONG-FLOAT. As specified in *Common LISP: The Language*, both parts of a complex number must be of the same type.

1.1.2 Characters

Common LISP defines characters as objects that have three attributes: code, bits, and font. The code attribute is a numerical representation of the character itself. The bits attribute allows extra flags (such as CONTROL or META) to be associated with a character. The font attribute permits a typographical style (such as italic) or font to be associated with a character. Common LISP does not require that an implementation honor bits or font attributes.

VAX LISP implements characters with eight bits for the code attribute, which is encoded using the extended ASCII character set. The bits and font attributes are no longer supported. The types CHARACTER and STRING-CHAR are now the same.

For compatibility, characters with bits attributes can still be read and printed, but, if they are put into a string, they lose their bits attribute. The bits attribute consists of the four Common LISP bits: CONTROL, HYPER, META, and SUPER. Note that the CONTROL bits attribute is not associated with control characters in the ASCII character set.

The VAX LISP implementation of Common LISP functions that take a font argument ignores it.

The VAX LISP implementation of Common LISP functions that compare characters uses the numeric values that correspond to the extended 8-bit ASCII character set. The character predicate functions and the rules that the functions use to compare characters are described in *Common LISP: The Language*.

The ordering of two characters that have the same character code but different bits and font attributes is undefined in VAX LISP.

The Common LISP character constants that are the exclusive upper limits on the code, bits, and font attributes have the following values in VAX LISP:

CHAR-CODE-LIMIT	256
CHAR-BITS-LIMIT	1
CHAR-FONT-LIMIT	1

NOTE

The values of these constants have changed in this release of VAX LISP.

Descriptions of these constants are provided in *Common LISP: The Language*.

You can obtain a table of valid VAX LISP character names by calling the VAX LISP CHAR-NAME-TABLE function described in the *VAX LISP / VMS Object Reference Manual*.

On ULTRIX Systems

The ULTRIX operating system masks the eighth bit, which produces the same effect as having specified 7-bit characters. You can prevent this masking by setting the terminal in RAW mode, but this is not recommended (see **tty**(4) in the *ULTRIX–32 Programmer's Manual*).

1.1.3 Arrays

Common LISP defines an array as an object whose components are arranged according to a Cartesian coordinate system and whose number of dimensions is called its rank. The names of the Common LISP array constants and the corresponding VAX LISP values are:

```
ARRAY-DIMENSION-LIMIT      16777215
ARRAY-RANK-LIMIT               8
ARRAY-TOTAL-SIZE-LIMIT     16777215
```

These constants are described in *Common LISP: The Language*.

Common LISP defines a specialized array as an array that can contain only elements of a specific type. Specialized arrays are more space efficient than general arrays. VAX LISP creates a specialized array when you call the MAKE-ARRAY function with the :ELEMENT-TYPE keyword set to any of the types in the following set:

```
CHARACTER               BIT
(UNSIGNED-BYTE 2)       (UNSIGNED-BYTE 4)       (UNSIGNED-BYTE 8)
(UNSIGNED-BYTE 12)      (UNSIGNED-BYTE 16)      (UNSIGNED-BYTE 24)
(UNSIGNED-BYTE 32)      (UNSIGNED-BYTE 64)
(SIGNED-BYTE 8)         (SIGNED-BYTE 16)        (SIGNED-BYTE 32)
(SIGNED-BYTE 64)
SINGLE-FLOAT            DOUBLE-FLOAT            LONG-FLOAT
```

If the element type is neither a member of this set nor a subtype of a member, VAX LISP creates a general array (element type is T).

In VAX LISP Version 3.0, vectors of characters are implemented as strings.

1.1.4 Strings

Common LISP defines a string as a vector of string characters. In VAX LISP, a string can be composed of as many as ARRAY-DIMENSION-LIMIT minus one character.

A string character is a character that can be stored in a string object. In VAX LISP Version 3.0, all characters are string characters. String characters cannot have the bits or font attributes.

On ULTRIX Systems

You can prevent the ULTRIX operating system from masking the eighth bit of a character by setting the terminal in RAW mode, but this is not recommended (see **tty**(4) in the *ULTRIX–32 Programmer's Manual*).

1.1.5 Functions

The Common LISP constants that are the exclusive upper bounds on the number of arguments to a function, the parameters in a lambda list, and the return values of a function have the following values in VAX LISP:

```
CALL-ARGUMENTS-LIMIT          1000000
LAMBDA-PARAMETERS-LIMIT           255
MULTIPLE-VALUES-LIMIT         1000000
```

1.2 Garbage Collector

When VAX LISP is executing, LISP objects are created dynamically in one area of the total memory pool assigned to LISP by the operating system. Some objects are always used and referred to, while others are referred to only for a short time. When the amount of memory allocated to LISP objects reaches a certain percentage of the LISP memory pool, all LISP processing is suspended. Objects that can still be referred to are copied into another area of dynamic memory, while objects that can no longer be referred to are not copied. The space that such objects occupied is thus reclaimed by the VAX LISP system. This process of reclaiming space is called garbage collection.

The following sections describe aspects of the VAX LISP garbage collector:

- Memory management
- Types of garbage collection
- Tuning garbage collection performance
- Increasing available space
- Garbage collection failure
- Controlling messages
- Effect on static space
- Effect on VMS interrupt functions

You can ignore garbage collections of dynamic memory space when you write LISP programs. Section 1.2.7 explains how to create objects in static rather than dynamic space.

1.2.1 Memory Management

When you start a LISP session, VAX LISP requests memory from the operating system. By default, 5000 512-byte pages are assigned to LISP and 50 percent may be used to allocate LISP objects. (See Section 1.2.4 for information on how to change these defaults.) When the percentage is reached, a garbage collection occurs, copying all live data from the current dynamic allocation area into unused memory (called the unassigned area until used). The memory previously allocated is then returned to the available LISP memory pool (to the unassigned area). Objects that are not referred to are not copied. All LISP processing is suspended during garbage collection.

To provide flexibility in allocation of objects and in garbage collection strategies, the VAX LISP system divides its memory pool into a number of different spaces: `:READ-ONLY`, `:STACK`, `:STATIC`, and `:DYNAMIC`. The `:READ-ONLY` space contains much of the LISP system. This space may not be written to and is shared among multiple processes when installed properly with the VMS INSTALL utility. Users building their own LISP systems with the System Build Utility may have portions of their own application placed in `:READ-ONLY` memory as well. `:STACK` space contains control and binding stacks and tends to remain relatively stable during a LISP session. Users' objects are not allocated in `:STACK` space. `:STATIC` space is used for allocation when users require allocated LISP objects which are guaranteed not to be moved by the garbage collector during the LISP session. `:STATIC` space also contains all nonreadonly portions of user applications built with the System Build Utility.

Garbage collection is not performed in the :READ-ONLY, :STACK, and :STATIC spaces, only in the :DYNAMIC space. Most LISP objects are allocated in :DYNAMIC space. When the ephemeral garbage collector is active, ephemeral areas are created within the dynamic space.

1.2.2 Types of Garbage Collection

VAX LISP uses both full and ephemeral garbage collection methods. In general, full collections happen infrequently (if enough memory is available) and free large amounts of memory. However, full collections take time proportional to the amount of :STATIC space, which must be scanned, and the amount of live data in :DYNAMIC space, which must be copied. In full collections, all objects are scanned and any dynamic objects that can be referred to are copied. In applications with large amounts of dynamic data, the time required can be substantial. This can present annoying pauses in interactive sessions. To remedy this, the ephemeral garbage collector may be used.

Ephemeral garbage collection takes the opposite approach: frequent small collections rather than infrequent large garbage collections. There is some overhead in doing ephemeral collection, because of the frequency of use; applications with much live data will require more total CPU time than they would running only the full garbage collection. On the other hand, user-visible garbage collection activity is kept to a minimum. Also, ephemeral collection does not replace full garbage collection because it only partially reclaims data. Depending on your application, a full garbage collection may be required at some point. In general, however, ephemeral garbage collection can postpone full garbage collections for a long time.

Besides reducing the visibility of garbage collection activity, the ephemeral collector also provides "age tracking" of data. The ephemeral collector maintains three areas of dynamic memory: E0, E1, and E2. Most allocation is done in the E0 area. When E0 is full, an ephemeral collection occurs and live data is copied (promoted) to E1. When E1 is full (presumably after many E0 collections), an ephemeral collection occurs and data that is still live is promoted to E2. When E2 is full, live data is promoted to the :DYNAMIC area where it remains in place until a full collection is required. It is generally accepted that in most LISP applications, most data has a short lifetime and becomes garbage very quickly. The longer data remains live, the more likely it will never become garbage. By using separate areas, ephemeral garbage collection eventually moves long-lived data to more permanent memory.

The ephemeral garbage collector is automatically enabled when you invoke VAX LISP, if enough memory is allocated (see Section 1.2.4 for details).

1.2.3 Tuning Garbage Collection Performance

The frequency of garbage collection is inversely proportional to the size of the current dynamic memory space; the speed is directly proportional. The degree to which the frequency of garbage collection and the size of dynamic memory affect run-time efficiency depends on the program. If a program creates more permanent objects than short-lived objects, the garbage collector has to perform more copy operations, slowing execution.

You can make full garbage collections occur less frequently by increasing the virtual memory allocated to LISP or increasing the percentage of dynamic space that must be occupied before a full collection takes place. See Section 1.2.4 for information on ways to set the amount of available memory. The DYNAMIC-SPACE-RATIO function returns the current percentage, as a floating-point value, of dynamic space that may be used before a full garbage collection is performed. This ratio may be changed by using the SETF macro. The new value must be greater than zero and less than or equal to one. Setting the dynamic space ratio can help the memory management system make more efficient use of memory, but the ratio is only a guideline. The ratio may be reset when the memory management system finds the current value to be inappropriate for existing conditions.

You can make ephemeral collections occur less frequently by increasing the size of the ephemeral areas. However, having too large an E0 area causes ephemeral garbage collections to take longer thus undoing the desired "user-invisible" effect of frequent, fast ephemeral garbage collections. Having a large E0 area and a too small E1 area may make most E0 collections instantly cause an E1 collection. While collection of the E0 area is optimized for speed, the others are used mostly for age tracking of data. It is recommended that E1 be large enough to give any live but soon-to-die data that survives an E0 collection a chance to die before promoting it to E2 or even :DYNAMIC space. The limits on the sizes of the ephemeral areas can be queried or changed with the AREA-SEGMENT-LIMIT function. For example:

```
Lisp> (area-segment-limit 0)
10
Lisp> (setf (area-segment-limit 0) 40)
```

Under certain circumstances, ephemeral garbage collection is counterproductive, as when loading files that create large amounts of permanent, live data. Disabling ephemeral collections before loading the files causes the data to be created directly in :DYNAMIC space, avoiding the overhead of promotions through the ephemeral areas However, garbage created by the loading is not discarded. You can disable or enable ephemeral collection with the GC-MODE function. For example:

```
(gc-mode :full)
```

1.2.4 Increasing Available Space

Garbage collection generally occurs when a LISP object is being created. If a garbage collection occurs and enough dynamic memory space to allocate the object is not available, the LISP system grows dynamic space as needed by requesting more memory from the operating system. The LISP system requests at least the number of 64K-byte segments in the value bound to the MEMORY-ALLOCATION-EXTENT function, depending on the size of the object. You can change the allocation extent with the SETF macro.

You can increase the total memory addressable by LISP with the ENLARGE-LISP-MEMORY function. Its format is:

ENLARGE-LISP-MEMORY *segments*

where *segments* is the number of 64K-byte segments by which the amount of virtual memory available for LISP allocation should be increased by the operating system.

You can also set the amount of memory available to LISP by specifying the /MEMORY command qualifier (on VMS systems) or the MEMORY (**-m**) option (on ULTRIX systems) when you invoke the LISP system. Garbage collection occurs less often if you use the memory qualifier or option to increase the size of the dynamic memory space. (See Chapter 2 of the *VAX LISP/VMS Program Development Guide* for more information about setting the amount of memory available to LISP.)

The ephemeral garbage collector is automatically turned on when the memory allocation size is larger than approximately 10,000 pages. This size can vary according to how the image was created and how it was started. In addition, normal execution may use up free space so that there is not enough left over after a garbage collection. In this situation, the ephemeral garbage collector is automatically turned off.

1.2.5 Garbage Collection Failure

The ephemeral garbage collector will be disabled when the sum of the size of the `:DYNAMIC` space and the ephemeral area segment limits is more than the `DYNAMIC-SPACE-RATIO` of the LISP-allocated pool. If this is caused by a user action, for example, you have enlarged an area segment limit, a warning message is printed. A full garbage collection is performed regardless of cause. You can reenable the ephemeral garbage collector by using the `GC-MODE` function: `(gc-mode :ephemeral)`. This function may not enable ephemeral garbage collection if there is insufficient memory.

The garbage collection process may fail to complete. If, for example, the LISP system cannot allocate enough memory to contain all the LISP objects that can be referred to without exceeding the process quota, the LISP image is terminated, and control returns to the DCL level (on VMS systems) or to the shell (on ULTRIX systems). The risk of this happening is greater when the value bound to the `DYNAMIC-SPACE-RATIO` function is large (for example, 90 percent) or if garbage collection is turned off (`GC-MODE :NONE`).

In previous versions of VAX LISP, a control stack overflow caused a garbage collection. In Version 3.0, control stack overflow is a fatal error. You can increase the allocation for control stacks with the /CSTACK qualifier to the DCL LISP command. (See Chapter 2 of the *VAX LISP/VMS Program Development Guide* for details.)

1.2.6 Controlling Messages

During a full garbage collection, messages are displayed when the operation begins and when it is finished. You can suppress these messages by changing the value of the VAX LISP `*GC-VERBOSE*` variable to `NIL`. When the value is `NIL`, messages are not displayed.

You can also specify the contents of the messages by changing the values of the VAX LISP `*PRE-GC-MESSAGE*` and `*POST-GC-MESSAGE*` variables. The `*GC-VERBOSE*`, `*PRE-GC-MESSAGE*`, and `*POST-GC-MESSAGE*` variables are described in the *VAX LISP/VMS Object Reference Manual*.

1.2.7 Effect on Static Space

LISP objects that are created in static space are not collected by the garbage collector. These objects do not move and they are not deleted, even if they can no longer be referred to. You can create objects in static space by using the `:ALLOCATION` keyword with the `MAKE-ARRAY` function or by using the constructor functions that are defined by the `DEFINE-ALIEN-STRUCTURE` macro for alien structures. (See the *VAX LISP/VMS Object Reference Manual* for details.)

1.2.8 Effect on VMS Interrupt Functions

LISP processing is suspended during a garbage collection. The VMS operating system queues interrupt functions, such as those defined by the VAX LISP `BIND-KEYBOARD-FUNCTION` and `INSTATE-INTERRUPT-FUNCTION` functions, for delivery after garbage collection is finished. Interrupt functions are discussed in Section 1.4.

1.3 Input and Output

On VMS systems, VAX LISP I/O is implemented with two sets of low level functions. One set of functions handles terminal I/O, using direct QIOs to the terminal driver. The other set of functions handles all other I/O (particularly to disk files), using calls to VAX Record Management Services (RMS). See the *VMS Record Management Services Manual* for information about VAX RMS.

On ULTRIX systems, VAX LISP terminal I/O and file I/O are implemented by low level ULTRIX system I/O routines. See the *ULTRIX–32 Programmer's Manual* for a description of ULTRIX I/O.

The following sections cover the specific VAX LISP I/O implementations of:

- Newline character
- Terminal input
- Terminal output (on ULTRIX systems)
- End-of-file operations
- Record length (on VMS systems)
- File organization
- Functions

1.3.1 Newline Character

Common LISP defines the `#\NEWLINE` character as a character that is returned from the `READ-CHAR` function as an end-of-line indicator. In VAX LISP, the character code for the `#\NEWLINE` character has an integer value of 255.

In VAX LISP, the `WRITE-CHAR` and `WRITE-STRING` functions interpret the `#\NEWLINE` character as follows:

- When the `WRITE-CHAR` function is called with the `#\NEWLINE` character as its argument value, the function starts writing a new line. This call is equivalent to a call to the `TERPRI` function (see *Common LISP: The Language*).

- When the WRITE-STRING function is called with an argument string that contains the #\NEWLINE character, the function divides the string into multiple lines. The following example shows the output that is displayed by the WRITE-STRING function when the #\NEWLINE character is **not** used:

```
Lisp> (write-string (concatenate 'string
                                 "NEW"
                                 "LINE"))
NEWLINE
"NEWLINE"
```

  Both strings, "NEW" and "LINE", are displayed on the same line. A call to the WRITE-STRING function with a string argument that **does** include the #\NEWLINE character, produces the following output:

```
Lisp> (write-string (concatenate 'string
                                 "NEW"
                                 (string #\newline)
                                 "LINE"))
NEW
LINE
"NEW
LINE"
```

  This call to the WRITE-STRING function displays the strings "NEW" and "LINE" on separate lines.

The #\NEWLINE character is the only character that moves the cursor to the beginning of the next line. #\LINEFEED moves the cursor down one line; #\RETURN moves the cursor to the beginning of the current line.

1.3.2 Terminal Input

In VAX LISP, terminals perform input operations in line mode. Input is returned by the READ-CHAR function only after you press the Return key.

On VMS Systems

The READ-CHAR function returns ASCII characters as data unless one of the following conditions exists:

1. A character is used by the VMS terminal driver for terminal control.
2. A character is defined to invoke an interrupt function.

See the *VMS I/O User's Reference Manual: Part I* for information on terminal control characters, and see Section 1.4 of this manual for information about interrupt functions.

You can change the mode in which your terminal performs input operations by invoking the VAX LISP SET-TERMINAL-MODES function with the :PASS-THROUGH keyword set to T (see the *VAX LISP/VMS Object Reference Manual*).

If the value of the :PASS-THROUGH keyword is T, then the VAX LISP system does not recognize control characters processed by the VMS system and characters defined to invoke interrupt functions. In addition, the READ-CHAR function performs input operations differently than it does when the terminal is in line mode. In line mode, the READ-CHAR function does not return a character until you press the Return key; in pass-through mode, that function returns a character as soon as the character is typed. See *Common LISP: The Language* for a description of the READ-CHAR function.

To put your terminal back into line mode, invoke SET-TERMINAL-MODES with :PASS-THROUGH set to NIL.

On ULTRIX Systems

The READ-CHAR function returns ASCII characters as data unless a character is used by the ULTRIX terminal driver for terminal control. See the *ULTRIX–32 Programmer's Manual* (particularly **ioctl**(2), **stty**(1), and **tty**(4)) for information on terminal control characters.

1.3.3 Terminal Output on ULTRIX Systems

ULTRIX truncates terminal output rather than wrapping it. To make output more readable, set the *PRINT-PRETTY* variable to T or *PRINT-RIGHT-MARGIN* to the width of the screen.

1.3.4 End-of-File Operations

In VAX LISP, read operations from a file do not indicate the end of the file until the operation after the last character in the file is performed.

Read operations from a terminal do not indicate the end of a file in VAX LISP.

In VAX LISP, you can close a stream that is connected to your terminal if the stream is not related to the stream bound to the *TERMINAL-IO* variable. If you attempt to close the stream bound to *TERMINAL-IO*, no action is performed.

1.3.5 Record Length on VMS Systems

VAX LISP/VMS uses VAX RMS to process file I/O. Therefore, the maximum record length in VAX LISP must conform to the maximum record length in RMS. A maximum of 32,767 characters can be written to a single record of a disk file, and a maximum of 9995 characters can be written to a single record of a magnetic tape. If you exceed these record-length limits, an error is signaled and nothing is written to the file.

The WRITE-CHAR function causes an immediate operation when it is called with a terminal stream. As a result, there is no limit on the number of calls you can make to the WRITE-CHAR function before you invoke the TERPRI function if you are writing to a terminal.

Your user-buffered I/O byte limit quota determines the maximum string length you can write to your terminal. You can find out what the quota is by invoking the VAX LISP/VMS GET-PROCESS-INFORMATION function with the :BIO-BYTE-QUOTA keyword (see *VAX LISP/VMS Object Reference Manual*). For example:

```
Lisp> (get-process-information "SMITH" :bio-byte-quota)
(:BIO-BYTE-QUOTA 30000)
```

NOTE

You can prevent your buffered I/O byte limit quota from overflowing and avoid the RMS record length limits by including calls to the TERPRI function, by inserting the #\NEWLINE character in your strings, or by setting *PRINT-RIGHT-MARGIN* or *PRINT-PRETTY*.

1.3.6 File Organization

VAX LISP/VMS reads RMS files sequentially. Character files created by VAX LISP/VMS have sequential organization, variable-length records, and the implied carriage-return attribute. Files created for binary output (with, for example, the WRITE-BYTE function) have sequential organization, variable-length records, and no carriage-control attributes.

VAX LISP/ULTRIX creates ULTRIX files that are sequential streams.

1.3.7 I/O Functions

Some Common LISP functions used for I/O have VAX LISP dependencies and need further explanation:

- OPEN
- WRITE-CHAR
- FILE-LENGTH (on VMS systems only)
- FILE-POSITION (on VMS systems only)

Unless otherwise noted, the implementation information in the following sections applies to both ULTRIX and VMS systems.

1.3.7.1 OPEN Function

Before you can access a file, you must open it with the OPEN function or the WITH-OPEN-FILE macro. The OPEN function can be specified with keywords that determine the type of stream that is to be created and how errors are to be handled. The keywords you can specify are:

```
:DIRECTION
:ELEMENT-TYPE
:IF-EXISTS
:IF-DOES-NOT-EXIST
```

VAX LISP restricts the values you can specify for these keywords. The rest of this section explains the restrictions.

For the :IF-EXISTS keyword values of :RENAME, :RENAME-AND-DELETE, and :SUPERSEDE, the old file is renamed to the same name with the string "old" appended to the file type. On closing files opened with any of these three values, and specifying :ABORT T, the new version is deleted and the old is restored to its former name. On closing files with :ABORT NIL, on :RENAME, there is no action; with :RENAME-AND-DELETE or :SUPERSEDE, the old file is deleted.

VAX LISP supports all the values for the :ELEMENT-TYPE keyword specified by Common LISP. VAX LISP lets you open binary streams, but the maximum byte size for a stream is 512 8-bit bytes.

On VMS Systems

In VAX LISP/VMS, you can specify the :IO value for the :DIRECTION keyword only if the specified stream is connected to a terminal or mailbox. When you specify the :IO value, the target device must exist before the OPEN function is called. Therefore, if you specify this value for the :DIRECTION keyword, you cannot specify the :IF-EXISTS keyword, and you can specify the :IF-DOES-NOT-EXIST keyword only with the :ERROR value.

The :IF-EXISTS :OVERWRITE option is not supported in VAX LISP/VMS.

On ULTRIX Systems

In VAX LISP/ULTRIX, if a file is opened with :DIRECTION :IO, the user must set the file position with the FILE-POSITION function when changing from reading to writing and vice versa. Not setting the file position will cause the file to be left in an inconsistent state.

1.3.7.2 WRITE-CHAR Function

The WRITE-CHAR function disregards the bit and font attributes of characters.

1.3.7.3 FILE-LENGTH Function on VMS Systems

The length of a file is measured in units of the OPEN function's :ELEMENT-TYPE keyword. In VAX LISP/VMS, files cannot be measured in these units for all the supported element types. Therefore, the FILE-LENGTH function returns NIL.

You can determine the total number of 8-bit bytes in a file by invoking the GET-FILE-INFORMATION function with the :END-OF-FILE-BLOCK and :FIRST-FREE-BYTE keywords, and then performing the following steps:

1. Multiply the value returned for the :END-OF-FILE-BLOCK keyword minus one by 512.
2. Add the value you get in step 1 to the value returned for the :FIRST-FREE-BYTE keyword.

For example:

```
(defun size-in-bytes (file)
  (let ((blocks (cadr (get-file-information file :end-of-file-block)))
        (bytes (cadr (get-file-information file :first-free-byte))))
    (+ bytes (* 512 (- blocks 1)))))
```

For the more information on the GET-FILE-INFORMATION function, see the *VAX LISP/VMS Object Reference Manual*.

1.3.7.4 FILE-POSITION Function on VMS Systems

The FILE-POSITION function returns or sets the current position within a random-access file. VAX LISP/VMS does not support random-access files; therefore, this function returns NIL.

1.4 VAX LISP/VMS Interrupt and Keyboard Functions

An interrupt function is a function that is invoked when a specific event occurs. If an interrupt function is defined for an event, the VAX LISP/VMS system interrupts the current LISP processing and invokes the interrupt function when the event occurs. When the interrupt function exits, the VAX LISP/VMS system resumes processing at the point where it was interrupted.

VAX LISP/VMS provides two functions you can use to define interrupt functions: INSTATE-INTERRUPT-FUNCTION and BIND-KEYBOARD-FUNCTION. The INSTATE-INTERRUPT-FUNCTION function is part of a general mechanism that lets your program respond to asynchronous events (ASTs) in the VMS operating system. This mechanism is described in the *VAX LISP/VMS System Access Guide*.

BIND-KEYBOARD-FUNCTION is a more specialized function that binds an ASCII control character to an interrupt function. Once a control character is bound to a function, you can cause the VAX LISP/VMS system to interrupt the current evaluation and call the function asynchronously by typing the control character. Functions bound using BIND-KEYBOARD-FUNCTION are also called keyboard functions.

Interrupt functions are not always called as soon as the defined event occurs. If a low-level LISP function, such as WRITE-CHAR or CONS, is being evaluated or a garbage collection is being performed, interrupt functions are placed in a queue until they can be evaluated. Delays in interrupt function evaluation are generally not perceptible. An example of when you might perceive a delay is when the system performs a garbage collection.

VAX LISP also provides a means by which you can assign different priorities for interrupt and keyboard functions. These priorities, called interrupt levels, are described in the *VAX LISP/VMS System Access Guide*.

If you suspend the LISP system when interrupt functions are defined, the functions that are defined by the BIND-KEYBOARD-FUNCTION function are still defined when the system is resumed. The key/function bindings are not lost. Any other interrupt functions that you may have defined are uninstated when the system is suspended and are not reinstated when the system is resumed.

Besides the BIND-KEYBOARD-FUNCTION function are the VAX LISP functions GET-KEYBOARD-FUNCTION and UNBIND-KEYBOARD-FUNCTION. The GET-KEYBOARD-FUNCTION function returns information about a function that is bound to a control character, and the UNBIND-KEYBOARD-FUNCTION function removes the binding of a function from a control character.

Descriptions of the BIND-KEYBOARD-FUNCTION, GET-KEYBOARD-FUNCTION, and UNBIND-KEYBOAPD-FUNCTION functions are provided in the *VAX LISP/VMS Object Reference Manual*.

1.5 VAX LISP/ULTRIX Keyboard Functions

A keyboard function is a function that is invoked when the user types a particular control key. The BIND-KEYBOARD-FUNCTION function binds an ASCII control character to a function, creating a keyboard function. A keyboard function interrupts the current LISP processing when the specified control key is typed. When the keyboard function exits, the VAX LISP/ULTRIX system resumes processing at the point where it was interrupted.

Note that you can use the BIND-KEYBOARD-FUNCTION to bind only three characters (Ctrl/C, Ctrl/\, and Ctrl/Z. See Chapter 2 of the *VAX LISP/VMS Program Development Guide* for more information on these characters.

Keyboard functions are not always called as soon as the specified control key is typed. If a low-level LISP function, such as WRITE-CHAR or CONS, is being evaluated or a garbage collection is being performed, keyboard functions are placed in a queue until they can be evaluated. Delays in keyboard function evaluation are generally not perceptible. An example of when you might perceive a delay is when the system performs a garbage collection.

VAX LISP also provides a means by which you can assign different priorities for keyboard functions. These priorities, called interrupt levels, are described in the *VAX LISP/ULTRIX System Access Guide*.

If you suspend the LISP system when keyboard functions are defined, the functions are still defined when the system is resumed. The key/function bindings are not lost.

Besides the BIND-KEYBOARD-FUNCTION function are the VAX LISP functions GET-KEYBOARD-FUNCTION and UNBIND-KEYBOARD-FUNCTION. The GET-KEYBOARD-FUNCTION function returns information about a function that is bound to a control character, and the UNBIND-KEYBOARD-FUNCTION function removes the binding of a function from a control character.

Descriptions of the BIND-KEYBOARD-FUNCTION, GET-KEYBOARD-FUNCTION, and UNBIND-KEYBOARD-FUNCTION functions are provided in the *VAX LISP/VMS Object Reference Manual*.

1.6 Functions and Macros

Several functions and macros described in *Common LISP: The Language* have implementation dependencies. Table 1–3 lists the names of these functions and macros and provides a brief explanation of the type of information that is implementation dependent. For a summary description of these functions and macros, see the *VAX LISP/VMS Object Reference Manual*. Each description consists of the function's or macro's use, implementation-dependent information, format, applicable arguments, return value, and examples of use. See *Common LISP: The Language* for further information regarding these functions and macros.

Table 1–3: Summary of Implementation-Dependent Functions and Macros

Name	Type	Implementation-Dependent Information
APROPOS	Function	Optional argument and DO-SYMBOLS macro
APROPOS-LIST	Function	Optional argument and DO-SYMBOLS macro
BREAK	Function	Facility invoked
COMPILE-FILE	Function	Keywords and return value
DESCRIBE	Function	Displayed output
DIRECTORY	Function	Argument merged with wildcards
DRIBBLE	Function	Cannot nest calls
		On VMS systems, also terminal I/O while in the Editor is not saved
ED (on VMS systems only)	Function	Arguments
GET-INTERNAL-RUN-TIME	Function	Meaning of return value
LOAD	Function	Finds latest file
LONG-SITE-NAME	Function	On VMS systems, the logical name and return value
		On ULTRIX systems, the location of information for string returned
MACHINE-INSTANCE	Function	Return value
		On VMS systems, also the logical name
MACHINE-VERSION	Function	Return value

(continued on next page)

Table 1–3 (Cont.): Summary of Implementation-Dependent Functions and Macros

Name	Type	Implementation-Dependent Information
MAKE-ARRAY	Function	:ALLOCATION keyword
REQUIRE	Function	Modules
ROOM	Function	Displayed output
SHORT-SITE-NAME	Function	On VMS systems, the logical name and return value On ULTRIX systems, the location of information for string returned
TIME	Macro	Displayed output
TRACE	Macro	Keywords
WARN	Function	Facility invoked

NOTE

T, NIL, and keywords are not legal function names in VAX LISP.

Chapter 2

VAX LISP Extensions to I/O

VAX LISP provides a number of extensions to the Common LISP I/O system. These extensions fall into the following three categories:

- A facility for defining new stream types. VAX LISP lets you define new types of character streams. Section 2.1 describes this facility.
- Information about streams. VAX LISP data types and functions provide more information about streams than is possible using only Common LISP facilities. Section 2.2 describes these data types and functions.
- New I/O functions. VAX LISP provides a number of I/O functions in addition to those defined in Common LISP. Section 2.3 describes these functions.

2.1 Defining New Types of Streams

Common LISP provides several types of streams; for example, synonym streams, broadcast streams, and echo streams. VAX LISP lets you define new types of streams that have different characteristics from those defined in Common LISP. You may want to define a new type of stream when you need input or output operations to have side effects unavailable with Common LISP streams.

To define a new type of stream, do the following:

- Design the stream. You need to decide how instances of the stream should respond to each valid I/O function.
- Define a means of creating instances of the stream. See Section 2.1.2.
- Define the action of each I/O function when acting on streams of that type. See Section 2.1.3.

2.1.1 Overview of VAX LISP I/O

In the VAX LISP I/O system, every instance of a stream includes a function called the stream dispatch function. Whenever an I/O function is called with a stream as its argument, the stream dispatch function for that stream executes. The stream dispatch function requires as its first argument an I/O request specifier whose value indicates the I/O function that was called. The stream dispatch function must, for every valid I/O request specifier, take the appropriate action for that I/O function operating on that type of stream.

In VAX LISP, streams are implemented as structures. Every structure definition of a stream type includes the STREAM structure definition provided in VAX LISP. In addition, stream type definitions may contain slots specific to the type of

stream. For example, the structure that implements a synonym stream contains a slot for the *symbol* to whose value the synonym stream is equated.

2.1.2 Defining Stream Structures

To define a new stream type, use the DEFSTRUCT macro to create a structure definition that includes the STREAM structure definition. Define additional slots as needed to satisfy the requirements of the stream type you have designed.

You must not use any of the following names for your slots:

```
CLASS                   IMMEDIATE-OUTPUT-P    OUTPUT-BUFFER
DISPATCH-FUNCTION       INPUT-BUFFER          OUTPUT-BUFFER-LIMIT
DOES-INPUT-P            INPUT-BUFFER-INDEX    RIGHT-MARGIN
DOES-OUTPUT-P           INPUT-BUFFER-LIMIT    SPECIFIC-STATE-1
ELEMENT-SIZE            LINE-POS-BASE         SPECIFIC-STATE-2
```

These slots are used by the VAX LISP I/O system and must not be modified.

Your structure definition must set the following slots in the STREAM structure by specifying default values in the :INCLUDE *stream* form:

- The DOES-INPUT-P and DOES-OUTPUT-P slots, whose values indicate whether input operations and output operations, respectively, are valid on the new stream type.
- The DISPATCH-FUNCTION slot, whose value is a function you write that performs each of the input and/or output operations that can result from function calls on the stream. (Section 2.1.3 describes stream dispatch functions.) The value of the DISPATCH-FUNCTION slot may also be a symbol with a function definition.

The following example defines a new stream type called SAMPLE-STREAM:

```
(defstruct (sample-stream
                (:constructor make-sample-stream
                              (input-stream output-stream))
                (:copier nil)
                (:include stream (does-input-p t)
                                 (does-output-p t)
                                 (dispatch-function
                                    'sample-stream-dispatch)))
  (input-stream nil :type stream :read-only t)
  (output-stream nil :type stream :read-only t))
```

This definition results in the following:

- A definition for the structure type SAMPLE-STREAM, instances of which contain the slots INPUT-STREAM and OUTPUT-STREAM in addition to those slots inherited from the STREAM structure definition
- A new type, SAMPLE-STREAM
- A new predicate, SAMPLE-STREAM-P
- A by-position constructor function whose format is:

  MAKE-SAMPLE-STREAM *input-stream output-stream*
- Accessor functions SAMPLE-STREAM-INPUT-STREAM and SAMPLE-STREAM-OUTPUT-STREAM

2.1.3 Stream Dispatch Functions

When an I/O function is called on a stream, the dispatch function for that stream type executes. Each stream type's dispatch function must perform the operations for each I/O function that can be called on streams of that type.

When it executes, the stream dispatch function receives at least two arguments:

1. An I/O request specifier (a keyword that corresponds to the I/O function that was called on the stream)
2. The stream on which the function was called

The stream dispatch function receives additional arguments that correspond to the additional arguments with which the I/O function was called. A stream dispatch function must be able to receive any number of arguments without error, although it need not process all arguments it receives.

For example, if MY-SAMPLE-STREAM is an instance of SAMPLE-STREAM, the following call:

```
(read-char my-sample-stream nil 'eof-encountered nil)
```

results in a call to SAMPLE-STREAM-DISPATCH with five arguments. The first argument is the I/O request specifier, :READ-CHAR in this case. The second through fifth arguments are the value of MY-SAMPLE-STREAM, NIL, 'EOF-ENCOUNTERED, and NIL.

Table 2–1 lists the I/O request specifiers that stream dispatch functions must handle. Not all I/O functions have a corresponding specifier, because some I/O functions (such as WRITE-LINE and TERPRI) are defined and implemented in terms of lower level functions.

Table 2–1: I/O Request Specifiers

Must Be Handled by All Streams		
:CLOSE	:ELEMENT-TYPE	
Must Be Handled by All Input Streams		
:CLEAR-INPUT	:LISTEN2	:NREAD-LINE
:READ-CHAR	:READ-LINE	:UNREAD-CHAR
Must Be Handled by All Output Streams		
:CLEAR-OUTPUT	:FINISH-OUTPUT	:FORCE-OUTPUT
:FRESH-LINE	:IMMEDIATE-OUTPUT-P	:LINE-POSITION
:RIGHT-MARGIN	:WRITE-CHAR	:WRITE-STRING

NOTE

See Section 2.3 for a description of the functions IMMEDIATE-OUTPUT-P, LINE-POSITION, LISTEN2, NREAD-LINE, and RIGHT-MARGIN. All other functions are described in *Common LISP: The Language*.

The :ABORT flag argument to CLOSE is passed as the first argument with the :CLOSE request.

The stream dispatch function SAMPLE-STREAM-DISPATCH might be written as follows. Note the use of &REST to ensure that SAMPLE-STREAM-DISPATCH can be called with an indefinite number of arguments without error.

```
(defun sample-stream-dispatch
       (request stream &optional arg1 arg2 arg3 &rest arg4)
  (declare (ignore arg4))
  (case (the keyword request)
    (:read-char
        (let ((char (read-char
                        (sample-stream-input-stream stream)
                        arg1 arg2 arg3)))
          (unless (eq char arg2)
            (write-char char
                        (sample-stream-output-stream stream)))
          char))
    (:write-char (write-char arg1
                             (sample-stream-output-stream stream)))
    (:unread-char (unread-char arg1
                               (sample-stream-input-stream stream)))
    .
    .
    .
    (t (error "~A does not recognize the ~A request"
              stream request))))
```

SAMPLE-STREAM-DISPATCH provides special handling when READ-CHAR is called on a stream of type SAMPLE-STREAM. For other I/O functions, SAMPLE-STREAM-DISPATCH simply calls the same function on either the input stream or the output stream. SAMPLE-STREAM-DISPATCH signals an error if it is called with an unrecognized I/O request specifier.

2.2 Getting Information About Streams

VAX LISP provides access to more detailed information about streams than is called for in *Common LISP: The Language*. VAX LISP provides a separate data type for each stream type, a predicate for each stream type, and functions to retrieve elements that were used to construct streams.

Table 2–2 lists the stream data types and predicates. The STREAMP predicate is satisfied by objects of any of the stream data types.

Table 2–2: Stream Data Types and Predicates

Data Type	Predicate Function
BROADCAST-STREAM	BROADCAST-STREAM-P *object*
CONCATENATED-STREAM	CONCATENATED-STREAM-P *object*
DRIBBLE-STREAM	DRIBBLE-STREAM-P *object*
ECHO-STREAM	ECHO-STREAM-P *object*
FILE-STREAM	FILE-STREAM-P *object*
STRING-STREAM	STRING-STREAM-P *object*
SYNONYM-STREAM	SYNONYM-STREAM-P *object*
TERMINAL-STREAM	TERMINAL-STREAM-P *object*
TWO-WAY-STREAM	TWO-WAY-STREAM-P *object*

Table 2–3 lists functions that retrieve information from streams. You cannot use SETF with these functions.

Table 2–3: Stream Informational Functions

Function Call	Return Value
BROADCAST-STREAM-STREAMS *broadcast-stream*	List of streams
CONCATENATED-STREAM-STREAMS *concatenated-stream*	List of streams
ECHO-STREAM-INPUT-STREAM *echo-stream*	Stream
ECHO-STREAM-OUTPUT-STREAM *echo-stream*	Stream
SYNONYM-STREAM-SYMBOL *synonym-stream*	Symbol
TWO-WAY-STREAM-INPUT-STREAM *two-way-stream*	Stream
TWO-WAY-STREAM-OUTPUT-STREAM *two-way-stream*	Stream

2.3 Additional I/O Functions

VAX LISP provides several I/O functions in addition to those defined in *Common LISP: The Language*. Most of these functions are variations on existing Common LISP functions. This section describes the functions in alphabetical order. The optional arguments *input-stream* and *output-stream* have the defaults specified in *Common LISP: The Language*:

- *input-stream* defaults to `*STANDARD-INPUT*`. If a value of `T` is supplied, the value of `*TERMINAL-IO*` is used.
- *output-stream* defaults to `*STANDARD-OUTPUT*`. If a value of `T` is supplied, the value of `*TERMINAL-IO*` is used.

The `IMMEDIATE-OUTPUT-P` predicate indicates whether an output stream does not buffer its output. The VAX LISP I/O system uses this function to improve output performance by buffering output when the stream itself does not perform buffering. Its format is:

IMMEDIATE-OUTPUT-P [*output-stream*]

The `LINE-POSITION` function returns the number of characters that have been output on the current line if that number can be determined and `NIL` otherwise. Its format is:

LINE-POSITION [*output-stream*]

The function returns a fixnum or `NIL`.

The `LISTEN2` function returns two values instead of the one returned by the Common LISP `LISTEN` function, enabling you to find out if end-of-file was encountered on the input stream. You can use this function wherever you would normally use `LISTEN`.

LISTEN2 [*input-stream*]

The function returns two values:

- `T` if a character is immediately available from *input-stream* and `NIL` otherwise.
- `T` if end-of-file was encountered on *input-stream* and `NIL` otherwise.

The NREAD-LINE function, a destructive version of the Common LISP READ-LINE function, places the characters that were read into the string supplied as its first argument. NREAD-LINE returns the number of characters read, a flag indicating whether end-of-file was encountered, and a string containing the line if the line could not fit into the string supplied. The format of the NREAD-LINE function is:

NREAD-LINE *string* [*input-stream*
eof-error-p
eof-value-p
recursive-p]

The *string* argument is a character string. NREAD-LINE updates *string* with the line that was read. If *string* has a fill pointer, the fill pointer is adjusted so that *string* appears to contain exactly what was read from the stream. If *string* is adjustable and the size of the line exceeds the size of *string*, then *string* is extended. Since NREAD-LINE does not return *string*, you must maintain a pointer to *string*.

The optional arguments correspond to the arguments to READ-LINE documented in *Common LISP: The Language*.

The NREAD-LINE function returns three values:

- A fixnum indicating the number of characters that were in the line.
- T if the line was terminated by end-of-file and NIL otherwise.
- NIL if the line fit into *string*: otherwise a string containing the line.

The OPEN-STREAM-P predicate indicates whether a stream is open. This function takes one argument, which must be a stream.

The RIGHT-MARGIN function returns the default right margin used by the pretty printer when printing to the stream. The current margin used by the pretty printer is controlled by the variable *PRINT-RIGHT-MARGIN*. Its format is:

RIGHT-MARGIN [*output-stream*]

The function returns a nonnegative fixnum.

Chapter 3

Window Streams

VAX LISP lets you create a stream that can receive typed input from a window and send character output to a window. This type of stream is called a window stream and has the following characteristics:

- By default, it is a valid target for all Common LISP and VAX LISP input and output functions that operate on character streams. (You can restrict window streams to output operations when you create them.)
- Output to the stream appears in the window.
- Input from the stream is taken from the keyboard and echoed in the window. A cursor marks the place where the next input character will be echoed.
- You can specify the font to be used for output and echoing to the stream.
- For output-only streams, you can specify whether to wrap, scroll, or truncate overflow from the bottom of the window.
- You can specify whether to wrap or truncate output beyond the right edge of the window.
- You can edit input.
- You can recall, edit, and reenter previous input.

This chapter describes the window stream facility in the following sections:

- Section 3.1 explains how to create window streams and describes their characteristics in detail.
- Section 3.2 describes input from window streams and shows how to edit typed input.
- Section 3.3 describes output to window streams.
- Section 3.4 lists window stream features and restrictions with each window system that supports them. (For this release, the supporting window systems are VMS Workstation Software and DECwindows.)
- Section 3.5 contains reference information on the window stream functions and macros.

3.1 Creating and Changing Window Streams

The MAKE-WINDOW-STREAM function creates and returns a window stream. The single required argument is a window. Keyword arguments let you specify a number of characteristics. You can also create a window stream, using the WITH-WINDOW-STREAM macro. This macro creates a window stream and binds it to a variable while the body of the macro executes. When the body returns, the stream is closed. The keyword arguments taken by MAKE-WINDOW-STREAM are used to specify the characteristics of the stream created by WITH-WINDOW-STREAM.

The following sections describe the characteristics of window streams and ways to change these characteristics.

3.1.1 Validity for Input Operations

By default, a window stream is a valid target for input and output operations. You can use the :DOES-INPUT-P keyword with a value of NIL to make the window stream an output-only stream. Input operations on such a stream cause an error.

You can use the Common LISP INPUT-STREAM-P function to determine whether a window stream is valid for input operations.

3.1.2 Viewing Area

A window stream's viewing area is the portion of the window in which window-stream output or echoing of its input appears. By default, this area is the entire window. You can use the :VIEWING-AREA keyword to specify a viewing area that is smaller than the entire window, although the viewing area must be large enough to contain at least one complete text character. When the viewing area is smaller than the window, output or echoing never appears outside the viewing area.

The value of the :VIEWING-AREA keyword is a list of four numbers. These numbers define a rectangle in the window's coordinate system. The format of the list is:

(*xmin ymin xmax ymax*)

where *min* and *max* refer to numeric values, not to spatial relationships in the window. Since windows in different window systems have different coordinate systems, specifying a viewing area necessarily introduces a dependence on the window system being used.

You can modify the size of the viewing area and you can also modify its contents in several ways besides input from or output to the window stream.

3.1.2.1 Viewing Area Coordinate Systems

Every viewing area has two coordinate systems. One coordinate system is the native coordinate system; that is, the coordinate system used by the supporting window system. The units and origin of the native coordinate system depend on the window system.

For example, you can create a window stream with a viewing area as follows:

```
(setq *ws* (window-stream:make-window-stream
             *win* :viewing-area '(100 100 200 200)))
```

The native coordinate system of this viewing area extends from 100,100 at one corner to 200,200 at the other.

The other coordinate system is the character-cell coordinate system. The character-cell coordinate system is independent of the window system. In the character-cell coordinate system, the viewing area is divided into character cells, each the width and height of a character. The character cell at the upper-left corner of the viewing area is 1,1, with coordinate values increasing to the right and down. Character-cell coordinate values are always positive fixnums.

The dimensions and placement of the character-cell coordinate system inside the viewing area are influenced by several factors:

- The font. The size of characters in the font determines the size of each character cell. Changing the font used by the window stream (see Section 3.1.4) may also change the dimensions of the character cells.
- The relationship of the viewing area's size to the character cell's size. If the width and height of the viewing area are not exactly divisible by the width and height of a character cell, there is space at the right and bottom of the viewing area where no characters can be written.
- The position of the cursor when the character-cell coordinate system is established or altered. Figure 3–1 shows this effect.

3.1.2.2 Obtaining Viewing Area Dimensions

You can obtain the dimensions of the viewing area in either native coordinates or character-cell coordinates. The `VIEWING-AREA-HEIGHT` and `VIEWING-AREA-WIDTH` functions return the height and width in native coordinate units. The `VIEWING-AREA-HEIGHT-CELL` and `VIEWING-AREA-WIDTH-CELL` functions return the height and width in chararacter-cell units.

The `WINDOW-STREAM-VIEWING-AREA` function returns the list of native-coordinate values that defines the viewing area. This function always returns a list of numbers, even if the viewing area was created using `NIL` to specify the entire window.

3.1.2.3 Changing Viewing Area Dimensions

You can use the `WINDOW-STREAM-VIEWING-AREA` function with the `SETF` macro to specify a new viewing area for a window stream. Use the same values as when creating the window stream; that is, a list of four numbers or `NIL` to specify the entire window. You always use native coordinates to specify a new viewing area.

Changing the viewing area positions the cursor to the upper-left corner of the new viewing area. The character-cell coordinate system is automatically realigned to the new viewing area.

It is an error to make the viewing area so small that it cannot contain at least one character cell.

3.1.2.4 Erasing Viewing Area Contents

Use the `ERASE-VIEWING-AREA` function to erase the contents of a window stream's viewing area. Erasing the viewing area repositions the cursor to the upper-left corner of the viewing area.

3.1.2.5 Scrolling Viewing Area Contents

Use the SCROLL-VIEWING-AREA and SCROLL-VIEWING-AREA-CELL functions to scroll the contents of the viewing area horizontally or vertically. SCROLL-VIEWING-AREA scrolls by native-coordinate units, while SCROLL-VIEWING-AREA-CELL scrolls by character cells. Without optional arguments, these functions scroll the viewing area contents up one native-coordinate unit or one line:

```
(window-stream:scroll-viewing-area-cell *ws*)
```

You can supply two optional arguments. The first argument specifies horizontal motion, with positive values moving viewing area contents to the left. The second argument specifies vertical motion, with positive values moving viewing area contents up. The following example moves the contents of *WS*'s viewing area five characters to the left and down three lines:

```
(window-stream:scroll-viewing-area-cell wc-stream 5 -3)
```

The SCROLL-VIEWING-AREA and SCROLL-VIEWING-AREA-CELL functions may perform vertical and horizontal scrolling separately when both are specified. This assures correct results in the viewing area.

Text that disappears as a result of scrolling cannot be recovered by scrolling in the opposite direction. Once it disappears from the viewing area, it is gone.

Scrolling moves everything within the viewing area, not just text that results from operations on the window stream.

3.1.3 Vertical and Horizontal Overflow

By default, a window stream scrolls when the output position goes past the bottom of the viewing area. That is, everything in the viewing area moves up; whatever was at the top of the viewing area disappears and is lost. The new output appears at the bottom of the viewing area. Unless you specify :DOES-INPUT-P NIL when you create the stream, this is the only available way to treat vertical overflow.

Window streams that are restricted to output allow more flexible treatment of vertical overflow. With the :WRAP value for :VERTICAL-OVERFLOW, you can specify that output is to wrap around to the top of the viewing area, overwriting what was there before. :TRUNCATE discards output that would be off the bottom of the viewing area.

By default, output that would go past the right edge of the viewing area is wrapped to the next line. You can change this by specifying :TRUNCATE with the :HORIZONTAL-OVERFLOW keyword. Excess output is then discarded until a #\NEWLINE character is encountered.

You can change the horizontal and vertical overflow behaviors after the window stream has been created. Use the SETF macro with the WINDOW-STREAM-VERTICAL-OVERFLOW and WINDOW-STREAM-HORIZONTAL-OVERFLOW functions. However, for window streams capable of input, values other than :SCROLL for WINDOW-STREAM-VERTICAL-OVERFLOW produce an error.

3.1.4 Font

When you create a window stream, you can use the :FONT keyword to specify a font. The value you supply for this keyword depends on the supporting window system. See Section 3.4 for information that depends on the window system.

You must specify a fixed-width font in a window stream. If you specify a variable-width font, the results are unpredictable.

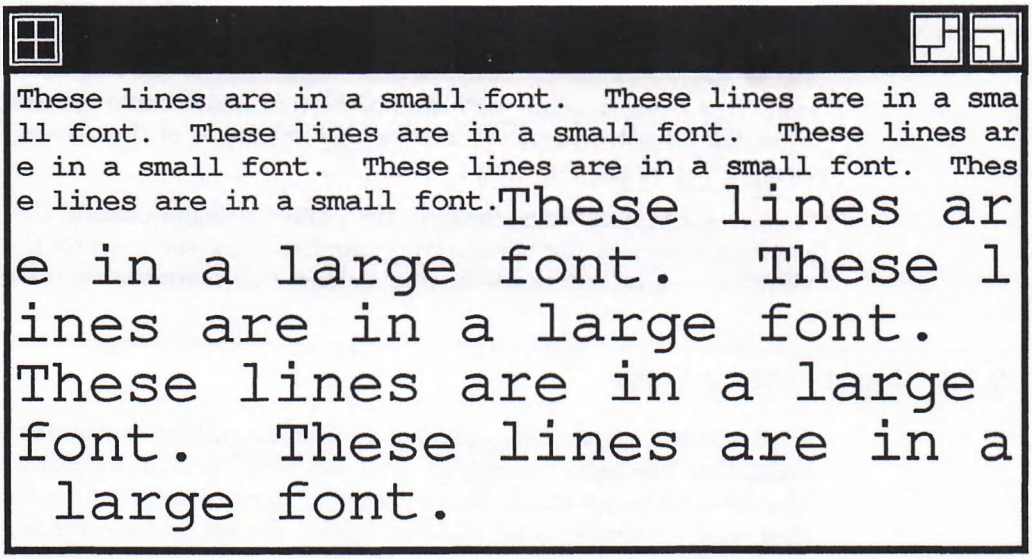
Specifying a font has two effects:

- It establishes the appearance of output and echoing characters on the screen.
- It establishes the dimensions and location of the character-cell coordinate system that is superimposed on the viewing area.

You can use the WINDOW-STREAM-FONT function with the SETF macro to change a window stream's font. If the character-cell size of the new font is different, the upper-left corner of the first character in the new font will be at the same place that the upper-left corner of a character in the old font would have occupied.

If you change the font when the cursor is in the middle of the viewing area, the new character-cell coordinate system may not align with the left edge of the viewing area. As a result, text using the new font also will not align at its left margin with previously output text. Figure 3–1 shows this effect in a CLX-based window.

Figure 3–1: Text Misalignment As a Result of Changing Fonts



MLO–003303

If changing to a larger font causes the viewing area to be too small for at least one complete character cell, an error results.

3.1.5 Cursor Position

A window stream's cursor indicates the position of the next input character in the window. You can determine the position of the cursor and change its position in the viewing area.

The cursor does not appear when a window stream is not capable of input. However, you can still obtain and change the cursor position for such streams.

Cursor position can be expressed in two ways: as a native-coordinate position or as a cell-coordinate position. A native-coordinate cursor position is expressed in native-coordinate units. It is always the cursor's position in the window, even when the stream operates in a viewing area that is smaller than the window. The cell-coordinate cursor position is always relative to the viewing area.

In window systems where the distinction is meaningful, the native-coordinate cursor position corresponds to the upper-left corner of the cell-coordinate cursor position.

The following accessor functions return the X or Y component of the cursor position:

```
WINDOW-STREAM-X-POSITION
WINDOW-STREAM-X-POSITION-CELL
WINDOW-STREAM-Y-POSITION
WINDOW-STREAM-Y-POSITION-CELL
```

You can use the SETF macro with each of these functions to change a window stream's cursor position.

When you change the native-coordinate cursor position, the position you specify may not align properly with the viewing area's character-cell coordinate system. For example, the new native-coordinate cursor position could be in the middle of one of the old character cells. When this happens, the character-cell coordinate system is reset to align with the native-coordinate cursor position you specify. This may result in space at the top and left edges of the viewing area that is not occupied by character cells.

It is an error to attempt to move the cursor position outside the viewing area. It is also an error to specify a native-coordinate cursor position that results in the viewing area no longer containing at least one complete character cell.

3.1.6 Input History Limit

Each window stream that performs input operations maintains a history of the input that has been typed to it. You can recall previous input to edit and reenter. The units of input in the history may be lines terminated by the Return key, or they may be complete input forms whose syntax is determined by a parser. Section 3.2.2 describes input recall and the scope of units of input.

The default number of input units maintained by the input history is 100. When you create a window stream, you can specify this number by using the :INPUT-HISTORY-LIMIT keyword. After a window stream has been created, you can change this number by using the SETF macro with the WINDOW-STREAM-INPUT-HISTORY-LIMIT function. A value of NIL specifies no limit. If you specify 0, no input history is maintained.

3.1.7 Window Exposure and Keyboard Connection

By default, calling an input function on a window stream makes the associated window visible and connects the keyboard to the window. Calling an output function on a window stream does not affect the associated window's visibility.

To make the window visible, the stream takes whatever action is appropriate under a particular window system. This may include bringing the window to the front of the screen, expanding it from an icon, or retrieving it from off-screen storage.

When you create a window stream, you can use the `:SHOW-ON` keyword to specify when its window becomes visible and when the keyboard is connected to it. The `:SHOW-ON` keyword takes the following values:

- `:INPUT` requests the default behavior.
- `:IO` makes the window visible and tries to connect the keyboard when an input function is called; it makes the window visible when an output function is called.
- `:OUTPUT` requests that the window be made visible for output, but not for input.
- `NIL` keeps window visibility and keyboard connection from being affected for either input or output.

The `WINDOW-STREAM-SHOW-ON` function returns the current setting of this value for a window stream. You can change the setting after you have created a window stream by using the `SETF` macro with `WINDOW-STREAM-SHOW-ON`.

3.2 Input from a Window Stream

Input from a window stream is the same as input from any stream defined by Common LISP, with the following additions:

- Input characters are echoed in the window stream's viewing area.
- You can edit the input after you have typed it and before it is read by the input function.
- The stream maintains a history of units of input. You can recall, edit, and reenter these input units later.

Section 3.2.1 describes input editing. Section 3.2.2 describes input recall.

3.2.1 Input Editing

When you type text as input to a window stream, certain keys and key sequences are bound to input editing commands. The commands reposition the cursor, delete units of text, or recall previously entered text. As you enter text or editing commands, the window stream's viewing area is updated to reflect the current input. The cursor shows where new characters will be inserted.

Input editing is described in the following subsections:

- Section 3.2.1.1 describes input editing without the `WITH-INPUT-EDITING` macro. In this mode, the Return key delimits units of input.
- Section 3.2.1.2 describes input editing with the `WITH-INPUT-EDITING` macro. In this mode, units of input are determined by code in the body of the macro.

- Section 3.2.1.3 describes options to the WITH-INPUT-EDITING macro.
- Section 3.2.1.4 lists the editing commands and their key bindings.

3.2.1.1 Editing Without WITH-INPUT-EDITING

When an input function that is not enclosed by the WITH-INPUT-EDITING macro reads from a window stream, you can edit your typed input until you press the Return key. When you press the Return key, all your input is made available to the input function. Your input is also added to the stream's input history.

Even though the Return key ends your input, the input may occupy more than one line in the window as a result of:

- Text wrapping at the right edge of the viewing area
- The insertion of a #\NEWLINE character with Ctrl/J or Ctrl/O

An input line that wraps is considered a single line; that is, it does not contain a #\NEWLINE character. On the other hand, input that breaks because you type Ctrl/J or Ctrl/O does contain embedded #\NEWLINE characters, which delimit logical lines within the input. The distinction is important because you can use the up-arrow key to move the cursor to preceding logical lines, but not to text on the same, wrapped line. (However, you can move over line-wrap breaks with the left-arrow and right-arrow keys.)

3.2.1.2 Using the WITH-INPUT-EDITING Macro

The WITH-INPUT-EDITING macro lets you enter, edit, and recall units of text that are not unconditionally terminated when you press the Return key. Instead, forms in the body of the macro determine when the input unit is complete. As long as the body of the macro is still consuming input, you can edit anywhere in the input. When the last form in the body returns, all the text that was consumed by the body is added to the stream's input history as a single unit.

The chief advantage of the WITH-INPUT-EDITING macro is that you can specify the syntactic content of an input unit by the code in the body of the macro. Typically, this code reads characters from the stream and parses them. As soon as the input satisfies the syntactic requirements of the code in the body, the body returns.

The READ function is an example of a parser that reads characters from a stream and does not return until it has parsed a syntactically complete unit. For the READ function, the unit of input is a LISP form. The following example reads complete LISP forms from the window stream *WS*:

```
(window-stream:with-input-editing (*ws*)
  (read *ws*))
```

If you type to the stream *WS* while the READ function in the example is taking input, you can break your input lines with the Return key. The READ function does not return until you have entered a complete LISP form, followed by Return. The READ function then returns the LISP object it has read, and this value is the return value of the WITH-INPUT-EDITING form.

If you notice an error in a line that you have already typed while you are typing the LISP form, you can use the up-arrow key to move the cursor to a previous line, then edit within the line. You can edit all your input until you have entered a complete form. If you later recall your input, you will recall the entire LISP form that you typed, not just individual lines.

You can write your own parser in the body of a WITH-INPUT-EDITING macro. Example 3–1 shows a parser that reads a specified number of words from a specified window stream and returns a list of these words. For the purposes of this example, a word consists of contiguous nonwhitespace characters, where the whitespace characters are #\SPACE and #\NEWLINE.

Example 3–1: Using the WITH-INPUT-EDITING Macro

```
(defun read-n-words (stream n)
  (let ((str (make-array 40 :element-type 'string-char
                            :adjustable t :fill-pointer t))
        (whitespace '(#\space #\newline)))      ; Define whitespace
    (window-stream:with-input-editing (stream)
      (do ((word-list '())                      ; Initialize list
           (c)                                  ; Input characters
           (word-count 0 (1+ word-count)))
          ((= word-count n) (nreverse word-list)) ; Stop after N words
        (setf (fill-pointer str) 0)

        (loop                                   ; Find non-whitespace
          (unless (member
                    (setq c (read-char stream))
                   whitespace
                   :test #'char=)
            (return)))

        (loop                                   ; Form word
          (vector-push-extend c str)
          (when (member
                  (setq c (read-char stream))
                 whitespace
                 :test #'char=)
            (return)))

        (push (copy-seq str) word-list)))))     ; Add word to list
```

If you write your own parser, it should not perform output or other side effects during execution. (In particular, it should not perform output to the window stream, as this will disrupt the echoed input.) To understand why, it is necessary to understand how the WITH-INPUT-EDITING macro operates.

Calling an input function on a window stream invokes an input editor. The input editor takes characters from the keyboard and places them in an editing buffer, echoing them as it does so. The input editor responds to certain characters, keys, or sequences by editing within the editing buffer and displaying the edited buffer in the window. The input editor does not pass the characters in the editing buffer to the input function until a terminating character is typed. By default, the terminating character is #\RETURN.

If WITH-INPUT-EDITING has not been used, typing Return passes the contents of the editing buffer to the input function, adds the input to the input history, and terminates the input editor. However, if the input function is in a WITH-INPUT-EDITING macro, Return passes the contents of the editing buffer to the input function but does not add the input to the input history or terminate the input editor. Instead, if the input function exhausts the editing buffer and still wants more characters, the input editor again takes characters from the keyboard and places them in the editing buffer. This cycle continues until the body of the WITH-INPUT-EDITING macro returns.

While typing input, the user can move the cursor to previous lines and edit them. But those lines have already been consumed by the input function; editing them invalidates the processing performed so far by the body. Therefore, when previous lines are edited, the WITH-INPUT-EDITING macro reexecutes its entire body, using the edited text typed thus far as input. This process is called rescanning.

Because the body of a WITH-INPUT-EDITING macro may be executed more than once, it should not perform output or other side effects as it executes. Output or side effects would occur each time the input was rescanned. Instead, the body of a WITH-INPUT-EDITING macro should build a data structure as it parses the input characters, then return that data structure when the parse completes. The body must initialize the data structure, since it must build the data structure from the beginning each time rescanning occurs.

In Example 3–1, the body of the WITH-INPUT-EDITING macro consists of a single DO macro. The DO macro initializes the list WORD-LIST each time it executes. When the DO terminates, indicating a successful parse, it returns WORD-LIST as its value.

3.2.1.3 Using Options to WITH-INPUT-EDITING

WITH-INPUT-EDITING takes a number of options. You specify these options as keyword-value pairs following the *window-stream* argument. The options let you:

- Specify a prompt for input.
- Specify which characters are considered terminators.
- Specify whether input should be rescanned if previous lines are edited.

The :PROMPT option lets you specify a string that is printed in the viewing area before input is echoed. For example, you could change Example 3–1 to read in part:

```
(window-stream:with-input-editing
    (stream :prompt (format nil "type ~d words: " n))
```

The :TERMINATORS option lets you supply a list of terminating characters. When the input editor encounters a terminating character, it passes the contents of its editing buffer to the input function.

By default, the only terminating character is #\RETURN. You may want to specify other characters as terminators. For example, when reading LISP forms you may want the right-parenthesis character to be a terminating character. That way, the READ function can examine the input at every right-parenthesis, instead of waiting for Return. For example:

```
(window-stream:with-input-editing
    (*ws* :terminators '(#\return #\)))
  (read *ws*))
```

This form reads from stream *WS*. Typing the final right-parenthesis character of a LISP form immediately causes the READ function to return. #\RETURN remains a terminating character.

In Example 3–1, in the second LOOP macro, you might want to have input passed to the READ-CHAR function as soon as a whitespace character is typed, instead of waiting for Return. You could change Example 3–1 to read in part:

```
(window-stream:with-input-editing
    (stream :terminators (cons #\return whitespace))
```

Note that you must specify #\RETURN as a terminator if you want the Return key to signal termination. The input editor receives a #\RETURN character from the keyboard, even though the input function reading from the stream receives a #\NEWLINE.

Do not specify NIL as the value of :TERMINATORS. If you do not specify terminating characters, the input editor can never return.

The :RESCAN option lets you specify whether your input should be subject to rescanning. The default value for this option is T. Specifying NIL defeats much of the purpose of the WITH-INPUT-EDITING macro. You can no longer edit lines already consumed by the body, and the input units retained by the input history are lines instead of complete forms. In other words, specifying :RESCAN NIL is much like reading from a window stream without using WITH-INPUT-EDITING. You can, however, use the other options to WITH-INPUT-EDITING to specify a prompt or a set of terminating characters.

3.2.1.4 Window Stream Editing Commands and Key Bindings

In general, keys and key sequences that perform editing operations are bound as in the EMACS editor. Table 3–1 lists the editing commands and the keys bound to them. Table 3–1 also includes keys such as Ctrl/S and Ctrl/Q that are not editing keys, but that have the effect described when the keyboard is connected to the window associated with the window stream.

The "logical line" referred to in Table 3–1 is delimited by a #\NEWLINE character. A logical line can span more than one line in the viewing area as a result of line wrapping. An input unit can contain more than one logical line, either as a result of Ctrl/J or Ctrl/O or because the input is being read using WITH-INPUT-EDITING.

If you use the BIND-KEYBOARD-FUNCTION function to bind a control key, this binding overrides any editing binding for that key. For example, if you bind Ctrl/E to invoke the VAX LISP Editor, typing Ctrl/E during input to a window stream invokes the Editor, rather than moving the window stream cursor to the end of a line.

The only control key bound by default, Ctrl/C, retains its binding when you are typing input to a window stream. That is, it clears the input and calls the condition system ABORT function. You can use BIND-KEYBOARD-FUNCTION to override this binding.

NOTE

> The default behavior of Ctrl/C has changed in Version 3.0. The previous default was to call the CLEAR-INPUT function on *TERMINAL-IO* and throw to the current CANCEL-CHARACTER-TAG.

Keypad keys 0 through 9, period (.), comma (,), and minus sign (–) insert those characters at the cursor position.

Generate an Escape key by typing the combination Ctrl/[. A letter following an Escape key may be either uppercase or lowercase.

Table 3–1: Window Stream Editing Commands and Key Bindings

Name	Binding	Description
	Cursor Movement Commands	
Move Forward Character	Ctrl/F or →	Moves cursor forward one character
Move Backward Character	Ctrl/B or ←	Moves cursor backward one character
Move Forward Word	Escape f	Moves cursor forward one word
Move Backward Word	Escape b	Moves cursor backward one word
Move to End of Line	Ctrl/E	Moves cursor to end of logical line
Move to Beginning of Line	Ctrl/A	Moves cursor to beginning of logical line
Move to Next Line	Ctrl/N or ↓	Moves cursor to next logical line of current input unit (down arrow recalls previously recalled input unit if you are not currently editing an input unit)
Move to Previous Line	Ctrl/P or ↑	Moves cursor to previous logical line of current input unit (up arrow recalls previous input unit if you are not currently editing an input unit)
Move to End of Input	Escape >	Moves cursor to end of current input unit
Move to Beginning of Input	Escape <	Moves cursor to beginning of current input unit
	Text Deletion Commands	
Delete Next Character	Ctrl/D	Deletes the character at the cursor position
Delete Previous Character	<X or Ctrl/H or F12	Deletes the character preceding the cursor position
Delete Forward Word	Escape d	Deletes from the cursor position through the next end of word
Delete Backward Word	Escape h or Escape <X	Deletes from the character preceding the cursor position back to the preceding beginning of word
Delete to End of Line	Ctrl/K	Deletes from the cursor position to the end of the logical line
Delete to Beginning of Line	Escape k	Deletes from the cursor position to the beginning of the logical line
Clear Input	Ctrl/G	Deletes all input currently being entered or edited

(continued on next page)

Table 3–1 (Cont.): Window Stream Editing Commands and Key Bindings

Name	Binding	Description
	Line Breaking and Input Termination	
Insert Newline	Ctrl/J or F13	Inserts new line, moving cursor to beginning of new line; does not terminate input editor
Open Current Line Here	Ctrl/O	Inserts new line, leaving cursor at end of current line; does not terminate input editor
Insert Newline at End and Terminate	Return or Ctrl/M	Inserts new line at end of input unit and terminates input editor (unless `WITH-INPUT-EDITING` is used, in which case input editor may not terminate); cursor is at beginning of new line
	Input Recall	
Recall Previous Input	↑	When you are not entering or editing text, recalls previous input units
Recall Previously Recalled Input	↓	When recalling input units, recalls the unit following the one currently displayed
	Miscellaneous	
Escape	Ctrl/[or F11	Generates an escape prefix
Insert Close Parenthesis and Match	)	Inserts a close parenthesis and highlights the corresponding open parenthesis, if one exists
Insert Multiple Spaces	Tab	Inserts multiple spaces at the cursor position and passes multiple spaces to the input function: does not pass a `#\TAB` character to the input function
Describe LISP Function	Ctrl/?	Describes the current LISP function
Refresh Input	Ctrl/L	Refreshes input, including the prompt if any; forces input editor to recalculate viewing area dimensions
Clear Viewing Area	Escape Ctrl/L	Clears the viewing area associated with the window stream
Help on Input Editor	Escape ?	Prints a list of input editor key bindings

3.2.2 Recalling Input

You can recall input units that were previously entered to a window stream. To do so, you either must not have entered anything to the current input request, or you must clear what you have entered with Ctrl/G. You can then recall previous input units by pressing the up-arrow key. After you have pressed the up-arrow key, you can recall subsequent input units with the down-arrow key.

When you reach an input unit that you want to reenter, either press Return to reenter it as is, or edit it before reentering it. Once you have started to edit an input unit, you cannot use the up-arrow or down-arrow keys to move to other input units unless you clear the current input with Ctrl/G.

You can recall all the input units that were previously entered to a window stream, up to the limit set when the stream was created. The recall buffer is circular; that is, an attempt to recall the input unit preceding the oldest one recalls the input unit most recently entered.

3.3 Output to a Window Stream

Output to a window stream is the same as output to any other Common LISP stream. By default, calling an output function on a window stream does not make the stream's associated window visible. See Section 3.1.7 for information on changing this behavior.

3.4 Window System Dependencies

This section describes aspects of window streams that depend on the window system in use. Each major subsection covers one window system on which the window stream facility is implemented. Within each subsection, the following system-dependent items are described:

- Native coordinate system
- Fonts and font specification

Other window system dependencies, if any, are described following these two.

3.4.1 VMS Workstation Software

This section covers the window stream facility running on a VAXstation using the VMS Workstation Software, also known as UIS.

3.4.1.1 Native Coordinate System

The native coordinate system is the window's device coordinate system. Native coordinate values must be fixnums. The origin is at the lower left corner.

3.4.1.2 Fonts and Font Specification

The default font is the font specified by attribute block 0. You can specify a font in the same way as with the `UIS:SET-ATTRIBUTE` function.

3.4.1.3 Multiple Stream Restriction

You can create multiple window streams to the same window, but only one can perform input. All other streams associated with the window must specify `:DOES-INPUT-P NIL`.

3.4.1.4 Virtual Keyboard Use

Window streams use virtual keyboards to get input from a window. When you create a window stream that performs input, any previous association between the window and a virtual keyboard is broken. After you create a window stream, you must not associate another virtual keyboard with the window, or the association between the window and the virtual keyboard that feeds the stream will be broken.

The `UIS:WINDOW-STREAM-KB` function returns the virtual keyboard used by a window stream that performs input. You cannot use the `SETF` macro with this function. Do not attempt to specify a new interrupt function for a virtual keyboard used by a window stream.

3.4.1.5 Using Attribute Blocks

Each window stream requires the use of two dedicated attribute blocks. A window stream always uses attribute block 255, and all window streams associated with windows into a single display can share attribute block 255.

The other attribute block required by a window stream stores the font and, therefore, cannot be shared among streams. The first window stream associated with a window into a particular virtual display uses attribute block 254. The second stream uses attribute block 253, and so on.

To summarize: Window streams require dedicated attribute blocks and they take the highest-numbered blocks in each virtual display. The highest-numbered attribute block available is 254 minus the number of window streams associated with windows into that virtual display. Do not modify the contents of either attribute block. Results may be unpredictable.

The `UIS:WINDOW-STREAM-ATTRIBUTE-BLOCK` function returns the variable attribute block used by a window stream (that is, the one that is not attribute block 255). You cannot use the `SETF` macro with this function.

3.4.1.6 Resizing Windows

If the user resizes a window associated with a window stream, the window stream facility does not automatically resize the viewing area for streams associated with that window. You can specify an interrupt function that resizes the viewing area or takes other appropriate action by using the `UIS:SET-RESIZE-ACTION` function. You can also prevent the user from resizing the window by using a value of `:DISALLOW` with `UIS:SET-RESIZE-ACTION`.

3.4.2 DECwindows

This section covers the window stream facility running on a VAXstation using DECwindows and implemented in Common LISP X (CLX).

3.4.2.1 Native Coordinate System

The native coordinate system in CLX is pixel coordinates with the origin at the upper-left corner. CLX does not provide world coordinates.

3.4.2.2 Fonts and Font Specification

The default font is Courier14Bold, the same as in the LISP Listener.

The font is determined by the `:GCONTEXT` slot of the window stream structure. You can specify a value for this slot when you create a window stream or use the default value. However, changing the graphics context of an existing window stream does *not* affect the window stream font. You can change the window stream font only by using `SETF` on the `WINDOW-STREAM:WINDOW-STREAM-FONT` function. The window stream has to know that font has changed so that it can do output.

3.4.2.3 Multiple Stream Restriction

You can create multiple window streams to the same window, but only one can perform input. All other streams associated with the window must specify `:DOES-INPUT-P NIL`.

In addition, only one window in a tree can perform input.

3.4.2.4 Using Graphics Contexts

There is one graphics context per window stream, used for all text output operations to that window stream. The CLX:WINDOW-STREAM-GCONTEXT function returns the graphics context used by a window stream. You cannot use the SETF macro with this function.

With the exception of the font slot, you can modify components of a window stream's graphics context. For example, the following code is "legal" and will make the text blue:

```
(let ((gc (window-stream:window-stream-gcontext ws)))
     (setf (clx:gcontext-foreground gc) "blue")
     (format ws "Am I blue?"))
```

3.4.2.5 Resizing Windows

If the user resizes a window associated with a window stream, the window stream facility does not automatically resize the viewing area for streams associated with that window.

3.4.2.6 Repainting Windows

CLX-based windows do not automatically repaint themselves after they have been obscured. It is advisable to draw to an image or drawable and copy from there, rather than drawing directly to a window.

3.5 Window Stream Functions and Macros

This section contains reference descriptions of all the functions and macros exported from the WINDOW-STREAM: package.

ERASE-VIEWING-AREA Function

Erases the viewing area of a window stream, repositioning the cursor to the upper-left corner. See Chapter 4 for more information on using this function.

Format

WINDOW-STREAM:ERASE-VIEWING-AREA *window-stream*

Arguments

window-stream
A window stream.

Return Value

Undefined.

MAKE-WINDOW-STREAM Function

Creates and returns a window stream. Keywords control the characteristics of the stream.

Format

WINDOW-STREAM:MAKE-WINDOW-STREAM *window*
&KEY :DOES-INPUT-P :FONT :HORIZONTAL-OVERFLOW
:INPUT-HISTORY-LIMIT :SHOW-ON
:VERTICAL-OVERFLOW :VIEWING-AREA

Arguments

window
A window.

:DOES-INPUT-P
`T` (the default) or `NIL`, indicating whether this stream is a valid target for input function calls.

:FONT
A font specification; the default and format depend on the window system in use. At least one complete character cell in the specified font must fit into the specified viewing area.

:HORIZONTAL-OVERFLOW
Either `:WRAP` (the default) or `:TRUNCATE`.

:INPUT-HISTORY-LIMIT
A nonnegative fixnum, specifying the number of input units maintained by the window stream's input history; or NIL, indicating no limit on the number of input units. A value of 0 requests that no input history be kept. The default is 100.

:SHOW-ON
A value indicating, for both input and output, whether windows should be made visible and (for input) whether the keyboard should be connected to the window:

:INPUT	makes the window visible (and connects the keyboard) for input only. This is the default.
:IO	makes the window visible on input and output and connects the keyboard for input.
:OUTPUT	makes the window visible for output only.
NIL	prevents the window from being made visible (and the keyboard from connecting) for both input and output.

:VERTICAL-OVERFLOW
:SCROLL (the default), :WRAP, or :TRUNCATE. Unless you specify :DOES-INPUT-P NIL, the value of :VERTICAL-OVERFLOW must be :SCROLL or an error results.

:VIEWING-AREA
Either NIL, indicating that the viewing area should occupy the entire window, or a list of native-coordinate values in the form:

(xmin ymin xmax ymax)

NIL is the default. The viewing area must be able to contain at least one complete character cell.

Return Value

A window stream.

SCROLL-VIEWING-AREA Function

Moves the contents in the viewing area of a window stream by native-coordinate units. The contents can be moved horizontally, vertically, or both. Contents that scroll off the edge of the viewing area are lost and cannot be recovered by scrolling in the opposite direction.

Calling this function without optional arguments scrolls up the contents of the viewing area by one native-coordinate unit.

Format

WINDOW-STREAM:SCROLL-VIEWING-AREA *window-stream*
&OPTIONAL *x y*

Arguments

window-stream
A window stream.

x
A number specifying how many native-coordinate units to scroll horizontally. Positive values move viewing area contents to the left. The default is 0.

y
A number specifying how many native-coordinate units to scroll vertically. Positive values move up the viewing area contents. The default is 1.

Return Value

Undefined.

SCROLL-VIEWING-AREA-CELL Function

Moves the contents in the viewing area of a window stream by cell-coordinate units. Contents can be moved horizontally, vertically, or both. Contents that scroll off the edge of the viewing area are lost and cannot be recovered by scrolling in the opposite direction.

Calling this function without optional arguments scrolls up the contents of the viewing area by one line.

Format

WINDOW-STREAM:SCROLL-VIEWING-AREA-CELL *window-stream*
&OPTIONAL *x y*

Arguments

window-stream
A window stream.

x
A fixnum specifying how many cell-coordinate units to scroll horizontally. Positive values move viewing area contents to the left. The default is 0.

y
A fixnum specifying how many cell-coordinate units (lines) to scroll vertically. Positive values move up the viewing area contents. The default is 1.

Return Value

Undefined.

VIEWING-AREA-HEIGHT Function

Returns the height of a window stream's viewing area in native-coordinate units. You cannot use the SETF macro with this function. Use the WINDOW-STREAM-VIEWING-AREA function to change the size of the viewing area.

Format

WINDOW-STREAM:VIEWING-AREA-HEIGHT *window-stream*

Arguments

window-stream
A window stream.

Return Value

A number.

VIEWING-AREA-HEIGHT-CELL Function

Returns the height of a window stream's viewing area in character-cell units. You cannot use the SETF macro with this function. Use the WINDOW-STREAM-VIEWING-AREA function to change the size of the viewing area.

Format

WINDOW-STREAM:VIEWING-AREA-HEIGHT-CELL *window-stream*

Arguments

window-stream
A window stream.

Return Value

A fixnum.

VIEWING-AREA-WIDTH Function

Returns the width of a window stream's viewing area in native-coordinate units. You cannot use the SETF macro with this function. Use the WINDOW-STREAM-VIEWING-AREA function to change the size of the viewing area.

Format

WINDOW-STREAM:VIEWING-AREA-WIDTH *window-stream*

Arguments

window-stream
A window stream.

Return Value

A number.

VIEWING-AREA-WIDTH-CELL Function

Returns the width of a window stream's viewing area in character-cell units. You cannot use the SETF macro with this function. Use the WINDOW-STREAM-VIEWING-AREA function to change the size of the viewing area.

Format

WINDOW-STREAM:VIEWING-AREA-WIDTH-CELL *window-stream*

Arguments

window-stream
A window stream.

Return Value

A fixnum.

WINDOW-STREAM Type Specifier

Designates objects of type WINDOW-STREAM created by the MAKE-WINDOW-STREAM function.

Format

WINDOW-STREAM:WINDOW-STREAM

Arguments

None.

Return Value

None.

WINDOW-STREAM-ATTRIBUTE-BLOCK Function

See the description of UIS:WINDOW-STREAM-ATTRIBUTE-BLOCK in Part II of *VAX LISP/VMS Interface to VWS Graphics*.

WINDOW-STREAM-FONT Function

Returns a window system-dependent value indicating the font used by a window stream. You can use the SETF macro with this function to change the font. If the size of the new font is different, the viewing area's character-cell coordinate system is realigned; the upper-left corner of the first character output in the new font is placed at the native-coordinate cursor position. If changing to a larger font causes the viewing area to be too small for a complete character cell, an error results.

See Sections 3.1.4 and 3.4.2.2 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-FONT *window-stream*

Arguments

window-stream
A window stream.

Return Value

Depends on the window system in use.

WINDOW-STREAM-GCONTEXT Function

See the description of CLX:WINDOW-STREAM-GCONTEXT in Part IV of *DECwindows Programming Guide*.

WINDOW-STREAM-HORIZONTAL-OVERFLOW Function

Returns the horizontal overflow action for a window stream. You can use the SETF macro with this function to change the horizontal overflow action.

See Section 3.1.3 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-HORIZONTAL-OVERFLOW *window-stream*

Arguments

window-stream
A window stream.

Return Value

:WRAP or :TRUNCATE.

WINDOW-STREAM-INPUT-HISTORY-LIMIT Function

Returns the maximum number of input units maintained by a window stream's input history or NIL if there is no limit. You can use the SETF macro with this function to change the input history limit. Reducing the limit truncates the window stream's input history. If you set the limit to 0, no input history is maintained.

See Section 3.1.6 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-INPUT-HISTORY-LIMIT *window-stream*

WINDOW-STREAM-INPUT-HISTORY-LIMIT Function

Arguments

window-stream
A window stream.

Return Value

A nonnegative fixnum or NIL.

WINDOW-STREAM-KB Function

See the description of UIS:WINDOW-STREAM-KB in Part II of *VAX LISP/VMS Interface to VWS Graphics* or the description of CLX:WINDOW-STREAM-KB in Part IV of *DECwindows Programming Guide*.

WINDOW-STREAM-P Function

Returns T if its argument is a window stream or NIL otherwise.

Format

WINDOW-STREAM:WINDOW-STREAM-P *object*

Arguments

object
Any LISP object.

Return Value

T or NIL.

WINDOW-STREAM-SHOW-ON Function

Returns the type of operation that makes a window stream's window visible and (for input operations) connects the keyboard to the window. You can use the SETF macro with this function.

See Section 3.1.7 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-SHOW-ON *window-stream*

Arguments

window-stream
A window stream.

Return Value

`:INPUT`, `:IO`, `:OUTPUT`, or `NIL`.

WINDOW-STREAM-VERTICAL-OVERFLOW Function

Returns the vertical overflow action for a window stream. You can use the `SETF` macro with this function to change a stream's vertical overflow behavior. However, unless you created the stream with `:DOES-INPUT-P NIL`, any value other than `:SCROLL` results in an error.

See Section 3.1.3 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-VERTICAL-OVERFLOW *window-stream*

Arguments

window-stream
A window stream.

Return Value

`:SCROLL`, `:WRAP`, or `:TRUNCATE`.

WINDOW-STREAM-VIEWING-AREA Function

Returns a list of four numbers representing the native-coordinate viewing area of a window stream. This function returns a list even if the viewing area was specified with `NIL` and thus occupies the entire window.

You can use the `SETF` macro with this function to change a window stream's viewing area. Changing the viewing area moves the cursor to the upper-left corner of the new viewing area and resets the viewing area's character-cell coordinate system. An error results if the specified viewing area is too small to contain at least one complete character cell.

See Section 3.1.2.3 for information on using this function.

WINDOW-STREAM-VIEWING-AREA Function

Format

WINDOW-STREAM:WINDOW-STREAM-VIEWING-AREA *window-stream*

Arguments

window-stream
A window stream.

Return Value

A list of four native-coordinate values in the form:

(*xmin ymin xmax ymax*)

See Section 3.1.2 for details.

WINDOW-STREAM-WINDOW Function

Returns the window associated with a window stream. You cannot use the SETF macro with this function.

Format

WINDOW-STREAM:WINDOW-STREAM-WINDOW *window-stream*

Arguments

window-stream
A window stream.

Return Value

A window in the representation of the supporting window system.

WINDOW-STREAM-X-POSITION Function

Returns the horizontal component of a window stream's native-coordinate cursor position. You can use the SETF macro with this function to change the cursor position. It is an error to specify a cursor position that results in the viewing area no longer containing at least one complete character cell.

See Section 3.1.5 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-X-POSITION *window-stream*

Arguments

window-stream
A window stream.

Return Value

A number.

WINDOW-STREAM-X-POSITION-CELL Function

Returns the horizontal component of a window stream's character-cell coordinate cursor position. You can use the SETF macro with this function to change the cursor position.

See Section 3.1.5 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-X-POSITION-CELL *window-stream*

Arguments

window-stream
A window stream.

Return Value

A fixnum.

WINDOW-STREAM-Y-POSITION Function

Returns the vertical component of a window stream's native-coordinate cursor position. You can use the SETF macro with this function to change the cursor position. It is an error to specify a cursor position that results in the viewing area no longer containing at least one complete character cell.

See Section 3.1.5 for information on using this function.

WINDOW-STREAM-Y-POSITION Function

Format

WINDOW-STREAM:WINDOW-STREAM-Y-POSITION *window-stream*

Arguments

window-stream
A window stream.

Return Value

A number.

WINDOW-STREAM-Y-POSITION-CELL Function

Returns the vertical component of a window stream's character-cell coordinate cursor position. You can use the SETF macro with this function to change the cursor position.

See Section 3.1.5 for information on using this function.

Format

WINDOW-STREAM:WINDOW-STREAM-Y-POSITION-CELL *window-stream*

Arguments

window-stream
A window stream.

Return Value

A fixnum.

WITH-INPUT-EDITING Macro

Establishes a context in which code in the body of the macro determines when a complete input unit has been entered. An input unit is the unit of text that can be recalled and edited.

See Section 3.2.1.2 for more information about using this macro.

Format

WINDOW-STREAM:WITH-INPUT-EDITING (*window-stream* {*option*}*)
{*form*}*

Arguments

window-stream
A window stream.

option
Keyword-value pairs. Table 3–2 lists the keywords and the meanings of their values.

Table 3–2: Keyword Options to WITH-INPUT-EDITING

Keyword	Values and Meaning
:PROMPT	A string, a function of no arguments, or NIL. The string or return value of the function is printed in the viewing area before the input editor is called. If NIL, "Lisp>" is used. By default, no prompt is printed.
:RESCAN	T (the default) or NIL. If T, the input editor can rescan input that has already been consumed by the body of the macro. Rescanning, which occurs when the user edits lines already typed, forces reexecution of the body of the macro. NIL disables rescanning and prevents the user from editing input already consumed.
:TERMINATORS	A list of terminating characters. When the input editor encounters a terminating character, it performs the action specified by the character (if any), then passes the contents of the input buffer to the input function requesting input from the stream. The default value is (#\RETURN). Do not specify NIL as the value of :TERMINATORS. If no terminating characters are specified, the input editor can never return.

form
One or more forms that make up the body of the macro. Usually, the body consumes and parses input from *window-stream*. Since the body is reexecuted whenever rescanning takes place, it should not perform output or have other side effects. The input consumed by the body is added to the input history.

Return Value

The value returned by the last *form*.

WITH-WINDOW-STREAM Macro

Creates a window stream associated with a specified window, binds a variable to the stream, and executes forms with the variable bound to the stream. The stream is automatically closed upon exit from the WITH-WINDOW-STREAM form, and the value of the last form executed in the body is returned as the value of the macro.

Format

WINDOW-STREAM:WITH-WINDOW-STREAM (*var window* {*option*}*)
{*form*}*

Arguments

var window
A window stream associated with *window* is created and bound to *var*.

option
Keyword-value pairs that specify characteristics of the window stream. The keywords are the same ones you specify with MAKE-WINDOW-STREAM.

form
Forms that execute while the window stream is open.

Return Value

The value of the last *form* executed.

Chapter 4

Pretty-Printing and Using Extensions to FORMAT

Pretty-printing clarifies the meanings of LISP objects by modifying their printed representations. It inserts indentation and line breaks at appropriate places, making pretty-printed output easier to read than output produced with standard print functions. Pretty-printing is an alternative to standard printing for all LISP objects, but is particularly useful for printing LISP code, complex data lists, and arrays.*

When pretty-printing is enabled, any function that prints output can potentially perform pretty-printing. The following example contrasts the standard and pretty-printed treatments of a COND structure:

```
Lisp> (setf t-question '(cond ((equal terminal
'vt240) start) (t (prin1 '(what terminal type are you
using?)))))
(COND ((EQUAL TERMINAL (QUOTE VT240)) START) (T (PRIN1
(QUOTE (WHAT TERMINAL TYPE ARE YOU USING?)))))
Lisp> (pprint t-question)
(COND ((EQUAL TERMINAL 'VT240) START)
      (T (PRIN1 '(WHAT TERMINAL TYPE ARE YOU USING?))))
```

The first version (produced by the standard read-eval-print loop) breaks the line at an awkward place and provides no indentation. Only one line is being printed. The line is either wrapped or truncated, depending on the operating system (VMS or ULTRIX–32) and the setting of the terminal. The pretty-printed (PPRINT) version is more readable because it starts a new line at the beginning of a nested list, indenting the list to line up with the structure nested to the equivalent level in the first line.

This chapter describes four ways to print LISP objects:

- Section 4.1 tells how to pretty-print objects.
- Section 4.2 tells how to control the format of pretty-printed objects, using print control variables.
- Section 4.3 tells how to use the VAX LISP FORMAT directives that support pretty-printing.
- Sections 4.4 through 4.9 tell how you can extend the VAX LISP print functions to handle specific structures and types of structures by defining new print functions.

* VAX LISP pretty-printing and the extensions to FORMAT are based on a program described in the paper *PP: A Lisp Pretty Printing System*, A.I. Memo No. 816, December 1984. The paper and the program were written by Richard C. Waters, Ph.D., of the MIT Artificial Intelligence Laboratory.

4.1 Pretty-Printing with Defaults

Three print functions let you pretty-print without explicitly using print control variables:

- PPRINT formats an object and prints it to a stream.
- PPRINT-DEFINITION formats the function object of a symbol and prints it to a stream.
- PPRINT-PLIST formats the property list of a symbol and prints it to a stream.

Use PPRINT when you want to let the system decide how best to format an object. PPRINT prints whatever object is given as its argument. The COND structure at the beginning of this chapter is an example of the output format specified for lists starting with a particular symbol.

You can use PPRINT-DEFINITION to print the definition of a LISP function. Supply the function name as the argument, as follows:

```
Lisp> (defun belongs (this pile) (cond ((null pile) nil) ((equal
this (car pile)) pile) (t (belongs this (cdr pile)))))
BELONGS
Lisp> (pprint-definition 'belongs)
(DEFUN BELONGS (THIS PILE)
  (COND ((NULL PILE) NIL)
        ((EQUAL THIS (CAR PILE)) PILE)
        (T (BELONGS THIS (CDR PILE)))))
```

If the object to be printed is the property list of a symbol, use PPRINT-PLIST, as shown in the following example:

```
Lisp> (setf (get 'places 'cities) '(augusta sacramento))
(AUGUSTA SACRAMENTO)
Lisp> (setf (get 'places 'states) '(maine california))
(MAINE CALIFORNIA)
Lisp> (pprint-plist 'places)
(STATES (MAINE CALIFORNIA)
 CITIES (AUGUSTA SACRAMENTO))
```

PPRINT-PLIST prints only indicator-value pairs for which the indicator is accessible in the current package. PPRINT-PLIST emphasizes the relationships between the indicator-value pairs.

4.2 Pretty-Printing with Control Variables

VAX LISP supports the global print control variables included in Common LISP and provides three additional variables:

- *PRINT-RIGHT-MARGIN*
- *PRINT-MISER-WIDTH*
- *PRINT-LINES*

By changing the values of these variables, you can adjust printed output to suit a variety of situations. Note that *PRINT-MISER-WIDTH* and *PRINT-LINES* affect pretty-printed output only.

You can also specify values for these three variables in calls to the WRITE and WRITE-TO-STRING functions. These functions have been extended to accept the following keyword arguments:

:RIGHT-MARGIN
:MISER-WIDTH
:LINES

If you specify any of these arguments, the corresponding special variable is bound to the value you supply with the argument before any output is produced.

4.2.1 Explicitly Enabling Pretty-Printing

When the Common LISP variable *PRINT-PRETTY* is non-NIL, it enables pretty-printing. If you set *PRINT-PRETTY* to T, you can pretty-print by calling any print function. The LISP read-eval-print loop will also pretty-print when *PRINT-PRETTY* is non-NIL.

The following example shows the effect of a PRIN1 function call when pretty-printing is enabled:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (prin1 '((tiger tiger burning bright) (in the forests of
the night) (what immortal hand or eye) (could frame thy fearful
symmetry)))
((TIGER TIGER BURNING BRIGHT) (IN THE FORESTS OF THE NIGHT)
 (WHAT IMMORTAL HAND OR EYE) (COULD FRAME THY FEARFUL SYMMETRY))
```

You can also enable pretty-printing by specifying a non-NIL value for the :PRETTY keyword in functions such as WRITE and WRITE-TO-STRING.

4.2.2 Limiting Output by Lines

Pretty-printing lets you abbreviate output by controlling the number of lines printed. With the variable *PRINT-LINES* set to any integer value, the print function you use stops after printing the specified number of lines. The output stream replaces omitted output with the characters " . . . ". Abbreviation by number of lines occurs only when pretty-printing is enabled. See Section 4.7 for more details on abbreviating output.

The following example shows pretty-printed output with *PRINT-LINES* set to 1:

```
Lisp> (setf *print-lines* 1)
1
Lisp> (setf *print-pretty* t)
T
Lisp> (PRINT '((in what distant deeps or skies) (burnt the fire
of thine eyes) (on what wings dare he aspire) (what the hand
dare seize the fire)))
((IN WHAT DISTANT DEEPS OR SKIES) (BURNT THE FIRE OF THINE EYES) ...
```

4.2.3 Controlling Margins

The *PRINT-RIGHT-MARGIN* variable lets you adjust the width of printed output. An integer value of *PRINT-RIGHT-MARGIN* specifies how wide to print, even if not pretty-printing. With the left margin at 0, *PRINT-RIGHT-MARGIN* specifies the approximate number of columns in which the print function will try to print. The default value, NIL, causes the print functions to query the output stream for the right margin value. There needs to be a reasonable right margin value when pretty-printing and *PRINT-RIGHT-MARGIN* are NIL. The default varies, but is always appropriate to the output device.

The example below shows how setting *PRINT-RIGHT-MARGIN* affects the width of pretty-printed output:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-right-margin* 38)
38
Lisp> (prin1 '((and what shoulder, & what art) (could twist
the sinews of thy heart) (and when thy heart began to beat) (what
dread hand & what dread feet)))
((AND WHAT SHOULDER, & WHAT ART)
 (COULD TWIST THE SINEWS OF THY HEART)
 (AND WHEN THY HEART BEGAN TO BEAT)
 (WHAT DREAD HAND & WHAT DREAD FEET))
```

Output may exceed the right margin if the printer encounters a long symbol name or string. The left margin is normally 0, but you can change it by using logical blocks with the FORMAT function to indent (see Section 4.3).

4.2.4 Conserving Space with Miser Mode

Miser mode can help you avoid running out of horizontal space when you print complicated structures. Pretty-printing adds line breaks and indentation to output to indicate levels of nesting, so that deeply nested structures often use up much of the line width. Miser mode conserves line width by minimizing indentation and inserting new lines where possible. You can use this feature by setting the variable *PRINT-MISER-WIDTH* to an integer value two or three times the length of the longest symbol in the output (usually a value around 40 is appropriate).

The system subtracts the value of *PRINT-MISER-WIDTH* from the right margin of the output stream to determine the column at which miser mode takes effect. In other words, miser mode becomes effective when the total line width available for printing after indentation is less than the value of *PRINT-MISER-WIDTH*. You can set *PRINT-MISER-WIDTH* to NIL to disable miser mode. See Section 4.8 for more details.

The default value of *PRINT-MISER-WIDTH* is 40.

4.3 Extensions to the FORMAT Function

VAX LISP provides nine FORMAT directives in addition to those specified in Common LISP. The added directives allow you to specify:

- Logical blocks, which are groupings of related output tokens
- Multiline mode new lines, which result in new lines if output cannot fit on one line
- Indentation, which aids in indenting portions of a form

Table 4–1 lists and briefly describes the FORMAT directives that VAX LISP provides. This section provides a guide to their use. The section presupposes a thorough knowledge of the LISP FORMAT function as described in *Common LISP: The Language*.

Table 4–1: FORMAT Directives Provided by VAX LISP

Directive	Effect
~W	Prints the corresponding argument under direction of the current print control variable values. Similar to ~A and ~S in Common LISP.
~!	Begins a logical block. Without modifiers, this directive causes FORMAT to print a single argument following the control string, which should be a list. If the argument is not a list, the entire logical block is effectively a ~W.
	~:! The colon modifier sets values of the print control variables: *PRINT-PRETTY* ⇒ T *PRINT-ESCAPE* ⇒ T *PRINT-LENGTH* ⇒ NIL *PRINT-LEVEL* ⇒ NIL *PRINT-LINES* ⇒ NIL
	~@! The at-sign modifier causes directives within the logical block to print successive elements from the argument that follows the control string.
	~1! The numeral one modifier specifies parentheses for the print prefix and suffix.
~.	Ends a logical block.
	~@. The at-sign modifier effectively inserts an if-needed newline directive (~:_) before each whitespace in the logical block.
~_	Specifies a multiline mode new line. This directive is effective only in a logical block.
	~:_ The colon modifier specifies a conditional, or if-needed, newline.
	~@_ The at-sign modifier specifies a miser-mode newline. This is the least likely to split lines.
~;	Marks the end of the print prefix or the beginning of the print suffix. (The prefix also sets indentation.)
	~@; The at-sign modifier causes the prefix to be printed on every line.
~nI	Sets indentation to *n* columns after the beginning print column of the logical block. This directive is effective only in a logical block.
	~n:I The colon modifier sets indentation to *n* columns after the current print column.
~n/FILL/	Prints the elements of a list with as many elements as possible on each line. If *n* is 1, FORMAT encloses the printed list in parentheses. This directive is effective only in a logical block.
~n/LINEAR/	If the elements of the list to be printed cannot be printed on a single line, this directive prints each element on a separate line. If *n* is 1, FORMAT encloses the printed list in parentheses. This directive is effective only in a logical block.
~n,m/TABULAR/	Prints the list in tabular form. If *n* is 1, FORMAT encloses the printed list in parentheses. *m* specifies the column spacing; the default is 16. This directive is effective only in a logical block.

Use the FORMAT function as follows:

FORMAT *destination control-string* &REST *arguments*

This function prints the *arguments* according to the format you specify with directives in the *control-string*. *destination* specifies the output stream.

The FORMAT directives provide the sole means of performing pretty-printing in VAX LISP. All functions that explicitly perform pretty-printing (for example, PPRINT and PPRINT-DEFINITION) do so by using these directives. Objects printed with FORMAT are printed normally unless pretty-printing is enabled. Pretty-printing is enabled when both the following conditions exist:

- A logical block is started.
- *PRINT-PRETTY* is non-NIL, or the colon modifier is specified in the logical block directive (~:I).

Nothing prevents you from starting a logical block when *PRINT-PRETTY* is NIL. However, any conditional new lines or indentation specified within the logical block will be ignored. This feature results in normal output, as opposed to pretty-printed output. By allowing this flexibility, FORMAT lets you use one control string to format data, and the data is either printed normally or pretty-printed, according to the value of *PRINT-PRETTY*.

The sections that follow describe the application of the nine VAX LISP FORMAT directives and the effects of the colon and at-sign modifiers on them.

4.3.1 Using the Write FORMAT Directive

Use the ~W FORMAT directive to print an element when you want to use the current values of the print control variables. The argument for ~W can be any LISP object. In contrast, ~A and ~S specify the values of print control variables.

You can use up to four prefix parameters with ~W to pad the printed object:

~mincol,colinc,minpad,padcharW

For an explanation of these parameters, see the description under "FORMAT Directives Provided with VAX LISP" in the *VAX LISP/VMS Object Reference Manual*.

The colon modifier (~:W) binds the following print control variables for the duration of the WRITE: *PRINT-ESCAPE* to T, *PRINT-PRETTY* to T, *PRINT-LENGTH* to NIL, *PRINT-LEVEL* to NIL, and *PRINT-LINES* to NIL. The following example contrasts the effects of using ~W and ~:W:

```
Lisp> (setf *print-pretty* nil)
NIL
Lisp> (setf *print-escape* nil)
NIL
Lisp> (setf *print-length* 2)
2
Lisp> (setf colors '(("Yellow" "Purple" "Orange" "Green") ("Aqua"
"Pink" "Beige" "Buff") ("Peach" "Violet" "Chartreuse")))
((Yellow Purple ...) (Aqua Pink ...) ...)
Lisp> (format t "~W" colors)
((Yellow Purple ...) (Aqua Pink ...) ...)
NIL
Lisp> (format t "~:W" colors)
(("Yellow" "Purple" "Orange" "Green") ("Aqua" "Pink" "Beige" "Buff")
 ("Peach" "Violet" "Chartreuse"))
```

The first FORMAT call truncates the first two sublists to two colors and truncates the outer list to two sublists. This truncation occurs because *PRINT-LENGTH* is 2. The first FORMAT call omits quotation marks because *PRINT-ESCAPE* is NIL. The second FORMAT call produces the full list of colors and includes quotation marks, because it implicitly sets *PRINT-LENGTH* to NIL and *PRINT-ESCAPE* to T. The second FORMAT call also indents the lists because it implicitly sets *PRINT-PRETTY* to T.

4.3.2 Controlling the Arrangement of Output

Two concepts support the dynamic arrangement of output for pretty-printing: logical blocks and conditional new lines. Logical block directives divide the total output into hierarchical groupings, which are referred to as logical blocks or subblocks. The goal of FORMAT is to print an entire logical block (including all its subblocks) on one line. If pretty-printing is enabled, the logical block is printed on one line only if the logical block fits between the current left and right margins. Printing all the output on one line is referred to as single-line mode printing.

The output for a logical block may not fit on one line when pretty-printing. In this case, the block must be subdivided into sections at points where it may be split into multiple lines. Conditional new line directives specify these points. Multiline mode printing is the name given to the condition where a logical block must occupy multiple lines.

When pretty-printing is enabled, FORMAT buffers the contents of a logical block until it can decide whether to use single-line mode or multiline mode printing.

A third mode, miser mode, is described briefly in Section 4.2.4 and in detail in Section 4.8.

Use the ~! and ~. directives to specify a logical block in the form:

~!*block*~.

where *block* can include any FORMAT directives. A logical block takes one argument from the FORMAT argument list. If that argument is a list, any directives within the logical block that take arguments take elements from that list, as shown in the following example:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-miser-width* nil)
NIL
Lisp> (setf *print-right-margin* 40)
40
Lisp> (format t "~!~W~." '((stars (betelgeuse deneb sirius)) (planets
(mercury venus earth mars jupiter saturn uranus neptune pluto))))
(STARS (BETELGEUSE DENEB SIRIUS))
NIL
```

The logical block takes the entire list as its argument. The list contains two elements but, since there is only one ~W directive, only the first element is printed.

With two Write directives (~W) and a multiline mode new line directive (~_), both elements of the list are printed:

```
Lisp> (format t "~!~W~_~W~." '((stars (betelgeuse deneb sirius))
(planets (mercury venus earth mars jupiter saturn uranus neptune
pluto))))
(STARS (BETELGEUSE DENEB SIRIUS))
(PLANETS
 (MERCURY VENUS EARTH MARS JUPITER
  SATURN URANUS NEPTUNE PLUTO))
NIL
```

(See the next section for more information on new line directives.)

If the argument is not a list, the logical block is effectively replaced by the ~W directive.

You can alter the directive to start a logical block (~!) by adding two modifiers. When the directive includes a colon (~:!), the directive sets *PRINT-PRETTY* and *PRINT-ESCAPE* to T and *PRINT-LENGTH*, *PRINT-LEVEL*, and *PRINT-LINES* to NIL for all the printing controlled by the logical block.

When the ~! directive includes an at sign (~@!), the directives within the logical block take successive arguments from the FORMAT argument list. The logical block uses up all the arguments, not just a single list argument. Therefore, no directives that take arguments from the argument list can appear after a logical block modified by an at sign in the logical block directive (see the last example in this section). You can use the ~^ directive inside a logical block to check whether the logical block arguments have been reduced to a non-NIL atom. See Section 4.9 for information on handling improperly formed argument lists.

The output associated with any FORMAT directive is subject to pretty-printing when the directive occurs within a logical block and *PRINT-PRETTY* is non-NIL.

A logical block defines an indentation level and can define a prefix and a suffix. By default, when pretty-printing is enabled, the indentation level is the position of the first character in the logical block. Each line following the first line in the logical block is printed preserving indentation and per-line prefixes, so that the first character in the line normally lines up with the first character in the block following the prefix. However, no default prefix or suffix is associated with a logical block.

You can create nested logical blocks within a logical block, using the ~!*block*~. directive. For example:

```
Lisp> (setf *print-right-margin* 70)
70
Lisp> (setf *print-pretty* t)
T
Lisp> (format t "~!Stars: ~!~S ~S~. Planets: ~!~S ~S~.~."
          '((betelgeuse deneb) (mars jupiter)))
Stars: BETELGEUSE DENEB Planets: MARS JUPITER
```

In this example, two logical blocks are created within the principal logical block. Each logical block uses the next argument for printing:

- The enclosing logical block uses the elements of the principal list ((BETELGEUSE DENEB) (MARS JUPITER)) as its arguments.
- The first inner logical block uses the elements of the list (BETELGEUSE DENEB) as its arguments.
- The second inner logical block uses the elements of the list (MARS JUPITER) as its arguments.

```
Lisp> (setf *print-pretty* nil)
NIL
Lisp> (format t "~:!Stars: ~!~S ~S~. Planets: ~!~S ~S~.~."
          '((betelgeuse deneb) (mars jupiter)))
Stars: BETELGEUSE DENEB Planets: MARS JUPITER
```

In this example, the colon in the ~:! directive enables pretty-printing implicitly, producing the same output as the previous example.

```
Lisp> (setf *print-pretty* t)
T
Lisp> (format t "~@!~S ~%~S ~%~S ~%~S~."
          '(betelgeuse deneb sirius) 'polaris 'vega 'algol
           'aldebaran)
(BETELGEUSE DENEB SIRIUS)
POLARIS
VEGA
ALGOL
```

In this example, the at sign causes the logical block to use all following arguments. Unneeded arguments are used up by the logical block but not printed. The first ~S applies to the first argument (the list (BETELGEUSE DENEB SIRIUS)). The remaining three ~S directives apply to POLARIS, VEGA, and ALGOL. ALDEBARAN goes unprinted, because there is no corresponding directive.

```
Lisp> (format t "~@!Stars: ~!~S ~S~. Planets: ~!~S ~S~.~."
          '(betelgeuse deneb) '(mars jupiter))
Stars: BETELGEUSE DENEB Planets: MARS JUPITER
```

In this example, the at sign in the outermost logical block directive (~@!) directs the logical block to use all the arguments. The first inner logical block uses the elements of the list (BETELGEUSE DENEB); the second inner logical block uses the elements of the list (MARS JUPITER).

4.3.3 Controlling Where New Lines Begin

Five FORMAT directives let you specify places where new lines can start according to the demands of the situation. Each directive delimits a section in a logical block.

- The ~% directive produces an unconditional new line. When used within a logical block, the directive preserves indentation and per-line prefixes.
- The ~& directive produces a fresh line. When used within a logical block, the directive preserves indentation and per-line prefixes.
- The ~_ directive produces a multiline mode new line when used within a logical block.
- The ~:_ directive produces an if-needed new line when used within a logical block.
- The ~@_ directive produces a miser-mode new line when used within a logical block.

You can specify unconditional new lines (~%) and fresh lines (~&) if you know in advance how the text should be laid out. If a new line is produced by one of these directives when the FORMAT function is printing a logical block, FORMAT prints the logical block in the multiline mode, preserving indentation and per-line prefixes.

The ~& directive specifies a fresh line, whether or not pretty-printing is enabled. If the ~& directive occurs inside a logical block when pretty-printing is enabled and any output is on the line other than prefixes and indentation, the FORMAT call starts a fresh line, preserving indentation and per-line prefixes.

The following examples show the use of the ~% and ~& directives:

```
Lisp> (format t "Stars~:!;~@;~%~S ~%~S ~%~S~."
          '(betelgeuse deneb sirius))
Stars;
     ;BETELGEUSE
     ;DENEB
     ;SIRIUS
NIL
Lisp> (format t "Stars~:!;~@;~&~S ~&~S ~&~S~."
          '(betelgeuse deneb sirius))
Stars;BETELGEUSE
     ;DENEB
     ;SIRIUS
```

The first FORMAT call starts a new line after the prefix ";", because the ~% directive starts a new line wherever the directive occurs. Replacing the ~% directive with the ~& directive changes the output, because the fresh line is not needed after the prefix.

The remaining three new line directives offer flexibility because they are conditional. However, they have no effect on output (except length abbreviation—see Section 4.7.1) when pretty-printing is not enabled.

The ~_ directive (multiline mode new line) starts a new line if the output for the enclosing logical block is too long to fit on one line or if any other directive in the logical block causes a new line. When the output is too long, FORMAT uses multiline mode, and every ~_ directive in a logical block starts a new line. The ~:_ directive (if-needed new line) produces a new line if it is needed: if the following section of output is too long to fit on the current line. The ~@_ directive (miser-mode new line) produces a new line if pretty-printing is enabled with miser mode in effect (see Section 4.8 for details). The FORMAT function ignores the three conditional new line directives when they occur outside a logical block.

The following example shows how you can specify a multiline mode new line and an if-needed new line:

```
Lisp> (setf *print-miser-width* nil)
NIL
Lisp> (setf *print-right-margin* 16)
16
Lisp> (format t "~:!~S ~_~S ~:_~S ~_~S~."
          '(mercury venus earth mars))
MERCURY
VENUS EARTH
MARS
NIL
```

This FORMAT function produces output in the multiline mode, because the output will not fit on one line with the right margin set to 16. The multiline mode new line directives (~_) produce new lines for VENUS and MARS. The if-needed new line directive (~:_) directs FORMAT to start a new line before EARTH if needed (but no new line is needed).

You can produce printed output that fills up the space available in each line by using the at-sign (@) modifier with the directive that ends the logical block (~!*block*~@.). This modifier causes FORMAT to start a new line if needed following

every blank space or tab and is equivalent to inserting a ~:_ directive after each element to be printed, as shown in the following example:

```
Lisp> (setf *print-right-margin* 25)
25
Lisp> (setf *print-miser-mode* nil)
NIL
Lisp> (format t "~@:!antares alphecca albireo canopus castor
                   pollux mirzam algol bellatrix capella mira
                   mirfak dubhe polaris ~@." )
ANTARES ALPHECCA ALBIREO
CANOPUS CASTOR POLLUX
MIRZAM ALGOL BELLATRIX
CAPELLA MIRA MIRFAK DUBHE
POLARIS
NIL
```

4.3.4 Controlling Indentation

With pretty-printing enabled, a call to FORMAT indents the output for a logical block so that the first character in each succeeding line falls under the first character following the prefix in the first line. When pretty-printing is not enabled, the FORMAT call does not produce indentation, and the indentation directive has no effect.

Use the ~*n*I or the ~*n*:I directive if you want to change the standard pretty-printed indentation. The ~*n*I directive causes FORMAT to indent subsequent lines *n* spaces from the position of the first character in the logical block. The ~*n*:I directive, on the other hand, causes FORMAT to indent subsequent lines *n* spaces from the output column corresponding to the position of the directive. If you omit the parameter *n*, the default is 0. Although this parameter can be less than 0 when used with the colon, the indentation cannot move to the left of the first character in the logical block. An indentation directive affects only indentation produced on subsequent new lines.

The following example shows several variations of the indentation directive:

```
Lisp> (setf *print-miser-mode* nil)
NIL
Lisp> (setf *print-right-margin* 15)
15
Lisp> (format t "~:!~S ~2I~:_~S ~:I~S ~_~S ~1I~_~S~."
          '(betelgeuse deneb sirius vega aldebaran))
BETELGEUSE
  DENEB SIRIUS
        VEGA
 ALDEBARAN
NIL
```

DENEB lines up under the T in BETELGEUSE, because the ~:_ directive produces a new line and ~2I causes an indentation of two spaces past the beginning of the block. The ~:I directive sets the indentation to the column one space after the end of the second argument. This indentation takes effect on the next new line, so that VEGA lines up under SIRIUS. ALDEBARAN lines up with the first E in BETELGEUSE, because the ~1I directive resets the indentation to one column past the first character in the logical block.

The ~I directives only set the indentation. They do not start new lines and they do not take effect until new lines begin. Therefore, in the directives for DENEB and ALDEBARAN, the indentation directives precede the new line directives.

4.3.5 Producing Prefixes and Suffixes

You can specify FORMAT control strings that add prefixes and suffixes to the printed output produced for a logical block. Several options are available.

If you divide the format control string into three sections by inserting the separator directive ~; twice, the string will specify a prefix and a suffix, as follows: ~!*prefix*~;*body*~;*suffix*~.. The first ~; directive marks the end of the prefix; the second marks the beginning of the suffix. If you omit the second ~; directive, no suffix is specified. Although the body can be any FORMAT control string, the prefix and suffix cannot include FORMAT directives.

When a FORMAT call prints output for a logical block that includes a prefix and pretty-printing is enabled, the second line of the output is indented so that the second line lines up with the first character in the block following the prefix. When the logical block includes a suffix, the FORMAT call always prints the suffix at the end, even if abbreviation directives eliminate some of the body of the block.

In the following examples, "Stars <" forms the prefix, and ">" forms the suffix.

```
Lisp> (setf *print-pretty* t)
T
Lisp> (format t "~!Stars <~;~S ~%~S ~_~S~;>~."
          '(sirius vega deneb))
Stars <SIRIUS
       VEGA
       DENEB>
NIL
Lisp> (setf *print-length* 2)
2
Lisp> (format t "~!Stars <~;~S ~%~S ~_~S~;>~."
          '(sirius vega deneb))
Stars <SIRIUS
       VEGA ...>
NIL
```

In the second example, FORMAT truncates the list to two elements, because *PRINT-LENGTH* is set to 2 (see Section 4.7) but it still adds the suffix after the last list element. VEGA lines up under SIRIUS in the first column for the body of the logical block.

You can specify the prefix parameter 1 in the logical block directive (~1!*block*~.), causing the FORMAT call to use parentheses for the prefix and suffix, as shown below.

```
Lisp> (format t "~1:!~S ~%~S~." '(castor pollux))
(CASTOR
 POLLUX)
NIL
```

You can create per-line prefixes in a logical block by specifying the at-sign modifier in the ~; directive used to indicate the end of the prefix (~@;). This modifier causes FORMAT to repeat the prefix at the beginning of each line, as shown in the following example:

```
Lisp> (format t "~:!<<~@;~S ~%~S ~_~S ~_~S~;>>~."
        '(algol antares albireo alphecca))
<<ALGOL
<<ANTARES
<<ALBIREO
<<ALPHECCA>>
```

The prefixes and the list elements line up.

If you nest logical blocks, you can specify a prefix with each block, as shown:

```
Lisp> (format t "~:!Bright stars~; ~@!<<~@;~S ~S ~%~S ~
~S~;>>~.~;still twinkle.~."
'(sirius vega deneb algol))
Bright stars <<SIRIUS VEGA
             <<DENEB ALGOL>> still twinkle.
```

The prefix and suffix for the outer logical block are "Bright stars" and "still twinkle". The prefix for the inner logical block, "<<", is printed on each line after the indentation required by the prefix for the first logical block. The suffix for the inner logical block, ">>", is printed once at the end of the block.

4.3.6 Using Tabs

You can use the tab directive to arrange output in columns. When pretty-printing is enabled, the ~*n*,*m*T tab directive counts spaces, beginning with the indentation of the immediately enclosing logical block. The integer *n* specifies a number of columns. The integer *m* specifies an increment: the number of columns to be added at one time until the column width is at least *n* columns. The at-sign modifier makes the tab directive relative, so that ~*n*,*m*@T counts spaces beginning with the current output column. When pretty-printing is not enabled, on the other hand, the ~*n*,*m*T directive counts spaces from the beginning of the line, as specified in Common LISP. The default for *n* and *m* is 1 (see *Common LISP: The Language* for details).

In the iterative example that follows, the tab directive precedes the if-needed new line directive:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-right-margin* 29)
29
Lisp> (format t "Stars: ~:@!~{~S~^ ~11T~S ~^ ~:_~}~."
          '(polaris dubhe mira mirfak bellatrix capella algol
            mirzam pollux canopus albireo castor alphecca
            antares))
Stars: POLARIS    DUBHE
       MIRA       MIRFAK
       BELLATRIX  CAPELLA
       ALGOL      MIRZAM
       POLLUX     CANOPUS
       ALBIREO    CASTOR
       ALPHECCA   ANTARES
NIL
```

Since the tabs are counted from the indentation of the logical block, the tab directives do not have to account for the fact that the whole block is shifted seven columns to the right.

4.3.7 Directives for Handling Lists

VAX LISP provides three FORMAT directives that simplify the printing of lists. Each implicitly uses the ~W directive repeatedly to print elements.

- If pretty-printing is enabled, the ~*n*/FILL/ directive causes FORMAT to fill the available line width by inserting a space and an if-needed new line after each list element except the last. FORMAT encloses the list in parentheses if *n* is 1. If pretty-printing is not enabled, ~*n*/FILL/ causes FORMAT to print the output in single-line mode.

- If pretty-printing is enabled, the ~*n*/LINEAR/ directive causes FORMAT to print the list on a single line if the list fits. Otherwise, FORMAT prints each element on a separate line. FORMAT encloses the list in parentheses if *n* is 1. If pretty-printing is not enabled, ~*n*/LINEAR/ causes FORMAT to print the output in single-line mode.
- If pretty-printing is enabled, the ~*n*, *m*/TABULAR/ directive causes FORMAT to print the list as a table, using columns of *m* spaces for list elements. The default value for *m* is 16. FORMAT encloses the list in parentheses if *n* is 1. If pretty-printing is not enabled, ~*n*, *m*/TABULAR causes FORMAT to print the output in single-line mode.

The following examples show the kinds of formats you can produce with the list-handling directives:

```
Lisp> (setf *print-miser-width* nil)
NIL
Lisp> (setf *print-right-margin* 36)
36
Lisp> (format t "Stars: ~@:!~/FILL/~."
          '(polaris dubhe mira mirfak bellatrix capella algol
            mirzam pollux canopus albireo castor alphecca
            antares))
Stars: POLARIS DUBHE MIRA MIRFAK
       BELLATRIX CAPELLA ALGOL
       MIRZAM POLLUX CANOPUS
       ALBIREO CASTOR ALPHECCA
       ANTARES
NIL
Lisp> (setf *print-right-margin* nil)
NIL
Lisp> (format t "Stars: ~@:!~/LINEAR/~."
           '(polaris dubhe mira mirfak bellatrix capella algol
             mirzam pollux canopus albireo castor alphecca
             antares))
Stars: POLARIS
       DUBHE
       MIRA
       MIRFAK
       BELLATRIX
       CAPELLA
       ALGOL
       MIRZAM
       POLLUX
       CANOPUS
       ALBIREO
       CASTOR
       ALPHECCA
       ANTARES
NIL
Lisp> (format t "Stars: ~@:!~0,20/TABULAR/~."
           '(polaris dubhe mira mirfak bellatrix capella algol
             mirzam pollux canopus albireo castor alphecca
             antares))
Stars: POLARIS             DUBHE             MIRA
       MIRFAK              BELLATRIX         CAPELLA
       ALGOL               MIRZAM            POLLUX
       CANOPUS             ALBIREO           CASTOR
       ALPHECCA            ANTARES
NIL
```

4.4 Defining Your Own Format Directives

VAX LISP lets you define your own FORMAT directives to supplement the directives supplied with the system. Any FORMAT directive that you define you can use in the control string argument to a FORMAT call.

(DEFINE-FORMAT-DIRECTIVE *name*
(*arg stream colon at-sign*
&OPTIONAL (*parameter1 default*)
(*parameter2 default*)
. . .)
&BODY *forms*)

This macro defines a directive named *name*. After you define a FORMAT directive, you can use it (whether or not pretty-printing is enabled) by including ~/*name*/ in a FORMAT control string.

NOTE

If you do not specify a package with *name* when you define the directive, *name* is placed in the current package. If you do not specify a package when you refer to the directive, the FORMAT directive looks in the USER package for the directive definition.

For the body of the macro call, the symbols you supply for *arg*, *stream*, *colon*, and *at-sign* are bound as follows:

- *arg* is bound to the argument list for the FORMAT directive you define.
- *stream* is bound to the stream on which the printing is to be done.
- The *colon* and *at-sign* arguments are bound to NIL unless the colon and at-sign modifiers are used with the directive.

There must be one optional argument for each prefix *parameter* that is allowed in the directive. A parameter argument will receive the corresponding prefix parameter if it was specified in the directive. Otherwise, the default value will be used, as with all optional arguments.

The body is evaluated to print the argument *arg* on the output *stream*. A user-defined FORMAT directive can be useful because it provides a level of indirection. In addition, you can call the directive repeatedly, which may save you some time coding and debugging. The following example shows a format directive used to produce error messages:

```
Lisp> (define-format-directive evaluation-error
                               (symbol stream colon-p atsign-p
                               &optional (severity 0))
        (declare (ignore atsign-p))
        (fresh-line stream)
        (princ (case severity
                 (0     "Warning: ")
                 (1     "Error: ")
                 (2     "Severe Error: "))
               stream)
        (format stream "~@:!The symbol ~S ~:_does not have an ~
                        integer value.~%Its value is: ~:_~S~."
                        symbol (symbol-value symbol))
        (when colon-p
          (write-char #\BELL stream)))
EVALUATION-ERROR
```

```
Lisp> (setf process nil)
NIL
Lisp> (format t "~1:/evaluation-error/" 'process)
Error: The symbol PROCESS does not have an integer value.
       Its value is: NIL
<BEEP>
NIL
```

This example shows the definition of a FORMAT directive, an application of the directive, and the printed output. It assumes that the current package is USER. The prefix parameter 1 in "~1:/EVALUATION-ERROR/" indicates the severity of the error being signaled. The colon in the FORMAT call produces a beep on the terminal.

4.5 Defining Print Functions for Lists

You can use DEFINE-LIST-PRINT-FUNCTION to define functions to print specific kinds of lists in formats of your choice. Functions that you define are effective only if pretty-printing is enabled. The printer checks the first element of each list that it prints. If the first element of a list matches the name of a list-print function, the list is printed according to the format you have specified. Create a list-print function according to the following format:

DEFINE-LIST-PRINT-FUNCTION *symbol* (*list stream*)
&BODY *forms*

This macro defines or redefines a print function for lists for which the first element is *symbol*. *list* is bound to the list to be printed and *stream* is bound to the stream on which the printing is to be done. The *forms* are evaluated to output *list*.

For example, if you define a list-print function for the symbol MY-SETQ, any list beginning with MY-SETQ will be printed in your format when pretty-printing is enabled:

```
Lisp> (setf *print-pretty* nil)
NIL
Lisp> (define-list-print-function my-setq (list stream)
        (format stream
                "~1!~W~^ ~:I~@{~W~^ ~:_~W~^~%~}~."
                list))
MY-SETQ
Lisp> (setf base '(my-setq hi 3 bye 4))
(MY-SETQ HI 3 BYE 4)
Lisp> (print base)
(MY-SETQ HI 3 BYE 4)
(MY-SETQ HI 3 BYE 4)
Lisp> (pprint base)
(MY-SETQ HI 3
         BYE 4)
```

When pretty-printing is not enabled, the value of BASE is printed without regard to the list-print function defined for MY-SETQ. PPRINT enables pretty-printing, producing a representation of the value of BASE using the specified list-print function.

VAX LISP pretty-printing incorporates predefined list-print functions for many standard LISP functions. However, if you define a list-print function for a LISP keyword, your function will override the one built into the system.

NOTE

When you use DEFINE-LIST-PRINT-FUNCTION, you may encounter two kinds of output that you do not expect:

- In most cases, a list whose first element is the symbol for a defined list-print function will be printed in the format specified, even if the context and meaning of the list are irregular and the format is inappropriate. For example, if your data says (LET IT BE) and LET is the symbol of a defined list-print function, the resulting output may be inappropriate.
- List-print functions are not used when you print a list under control of a user-defined FORMAT directive.

You can disable any defined list-print function by using the UNDEFINE-LIST-PRINT-FUNCTION macro. Its format is:

UNDEFINE-LIST-PRINT-FUNCTION *symbol*

This macro disables the list-print function defined for *symbol*. The following example disables the MY-SETQ list-print function defined in the example at the beginning of this section:

```
Lisp> (undefine-list-print-function my-setq)
MY-SETQ
```

4.6 Defining Generalized Print Functions

Using generalized print functions, you can specify how any object is pretty-printed, regardless of its form. Functions that you define and enable are effective only if pretty-printing is enabled. First you define a function with DEFINE-GENERALIZED-PRINT-FUNCTION. Then you enable the function. You can enable it globally, using GENERALIZED-PRINT-FUNCTION-ENABLED-P. Or you can enable it locally, using WITH-GENERALIZED-PRINT-FUNCTION.

Use the following format when you define a generalized print function:

DEFINE-GENERALIZED-PRINT-FUNCTION *name* (*object stream*) *predicate*
&BODY *forms*

This macro defines or redefines a print function with the name *name*. *object* is bound to the object to be printed. *stream* is bound to the stream to which output is to be sent. *predicate* governs the application of the generalized print function. The predicate is operative on any LISP object. A generalized print function will be used if it is enabled and the predicate evaluates to true on the object to be printed. NULL OBJECT is the predicate in the sample generalized print function shown at the end of this section. The pretty-printer uses your generalized print function to print any object for which the predicate does not evaluate to NIL. Evaluation of the specified *forms* must write a representation of *object* to *stream*.

If a generalized print function and a list-print function for the same symbol are both enabled, the generalized print function will be used.

A related function lets you test whether a specific generalized print function is enabled:

GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*

You can also use this function to globally change the status of the function, using SETF as shown:

(SETF (GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*) T)

or

(SETF (GENERALIZED-PRINT-FUNCTION-ENABLED-P *name*) NIL)

Use the WITH-GENERALIZED-PRINT-FUNCTION macro to locally enable a generalized print function in the following format:

WITH-GENERALIZED-PRINT-FUNCTION *name* &BODY *forms*

This macro locally enables the generalized print function named *name* when it evaluates the specified *forms*.

The printer checks generalized print functions that have been enabled in reverse order from the order of their enabling. This means that in cases where two or more generalized print functions apply, the most recently enabled function is used.

Enabling a generalized print function globally is less efficient than enabling it locally, because the printer must check the predicate of globally enabled print functions against every object to be printed. If you enable the generalized print function locally, the printer checks the function's predicate against the object being printed only during execution of the code within the macro, instead of on every call to a print function. Since the read-eval-print loop is used often, the difference in efficiency can be significant.

Consider the following examples:

```
Lisp> (setf *print-pretty* nil)
NIL
Lisp> (setf *print-right-margin* 25)
25
Lisp> (generalized-print-function-enabled-p 'print-nil-as-list)
NIL
Lisp> (define-generalized-print-function print-nil-as-list
                                         (object stream)
                                         (null object)
                (princ "( )" stream))
PRINT-NIL-AS-LIST
Lisp> (print nil)
NIL
NIL
Lisp> (pprint NIL)
NIL
Lisp> (with-generalized-print-function 'print-nil-as-list
        (print nil)
        (pprint nil))
NIL
( )
Lisp> (setf (generalized-print-function-enabled-p
               'print-nil-as-list) t)
T
Lisp> (pprint nil)
( )
```

The first PRINT call prints NIL, because pretty-printing is not enabled. The first PPRINT call prints NIL, because the generalized print function PRINT-NIL-AS-LIST is not enabled. The second PRINT call prints NIL, because pretty-printing is again not enabled. The second PPRINT call prints ( ), because the generalized print function is enabled locally and pretty-printing is enabled. The third PPRINT call prints ( ), because the generalized print function is enabled globally and pretty-printing is enabled.

NOTE

A generalized print function controls the pretty-printing of an object only if the following conditions exist:

- The generalized print function is enabled globally or locally.
- The predicate specified with DEFINE-GENERALIZED-PRINT-FUNCTION is true.
- The object to be printed does not come under control of a user-defined FORMAT directive.

In cases where two or more generalized print functions are applicable, only one is chosen. The one chosen is the most recently enabled (globally or locally) generalized print function for which the predicate specified with DEFINE-GENERALIZED-PRINT-FUNCTION is true.

Generalized print functions are not used when you print an object under control of a user-defined FORMAT directive.

4.7 Abbreviating Printed Output

You can abbreviate printed output according to:

- The length of the object to be printed
- The depth of nested logical blocks
- The number of lines in the output

Length and depth abbreviation are supported in Common LISP and are effective whether or not pretty-printing is enabled. In addition, abbreviation based on the number of lines of output is supported in VAX LISP; this is effective only when pretty-printing is enabled.

4.7.1 Abbreviating Output Length

You can control the number of sections of printed output by setting the *PRINT-LENGTH* variable. The value you supply specifies the number of sections to be printed for any affected logical block. The directives ~_, ~%, and ~& mark the sections of a logical block (see Section 4.3.3 for details). After the output stream prints *PRINT-LENGTH* sections of a logical block, it prints an ellipsis (. . .) and stops processing the logical block. If the logical block is nested with other logical blocks, the output stream terminates only the processing of the immediately enclosing logical block. Output is not truncated if the value of *PRINT-LENGTH* is NIL.

The following example shows output abbreviation based on length:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-right-margin* 47)
47
Lisp> (setf *print-length* 11)
11
```

```
Lisp> (format t "Stars: ~@!~{~W~^ ~:_~}~."
          '(polaris dubhe mira mirfak bellatrix capella algol
            mirzam pollux canopus albireo castor alphecca
            antares))
Stars: POLARIS DUBHE MIRA MIRFAK BELLATRIX
       CAPELLA ALGOL MIRZAM POLLUX CANOPUS
       ALBIREO ...
NIL
```

Each star name in the list constitutes a separate logical block section. FORMAT prints " . . . " after the eleventh star name to indicate that the list has been abbreviated at that point.

4.7.2 Abbreviating Output Depth

Use the variable *PRINT-LEVEL* to control the depth of printed output. *PRINT-LEVEL* specifies the lowest level of dynamically nested logical blocks to be printed. When your program calls FORMAT recursively, the output stream keeps track of the actual nesting level and abbreviates output when the level reaches *PRINT-LEVEL*. The printed character # indicates where the stream has truncated the output. You can prevent depth abbreviation by setting *PRINT-LEVEL* to NIL.

Dynamic nesting of logical blocks occurs frequently when you print complicated structures. This nesting may not be obvious as you read the program. For example, if you have defined list-print functions for the primitives IF and PROGN, printing a program that uses a combination of these primitives would involve dynamic nesting of logical blocks, since each list-print function uses the ~W directive implicitly. The following example shows how the output stream abbreviates the printing of a structure in accord with the value of *PRINT-LEVEL*:

```
Lisp> (setf *print-level* 3)
3
Lisp> (pprint '(level1 (level2 (level3 (level4 (level5))))))
(LEVEL1 (LEVEL2 (LEVEL3 #)))
Lisp> (setf *print-level* 2)
2
Lisp> (pprint '(level1 (level2 (level3 (level4 (level5))))))
(LEVEL1 (LEVEL2 #))
Lisp> (pprint '(level1 4 5 6 (level2 (level3 (level4
                (level5))))))
(LEVEL1 4 5 6 (LEVEL2 #))
```

4.7.3 Abbreviating Output by Lines

You can control the number of lines printed in the output by setting the *PRINT-LINES* variable. The value you supply specifies the number of lines to be printed for the outermost logical block. The output stream prints " . . . " at the end of the last line to indicate where it has truncated the output. If *PRINT-LINES* is NIL, the output stream will not abbreviate the number of lines printed. This abbreviation mechanism is effective only when pretty-printing is enabled.

In the following example, printing stops at the end of the fourth line:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-miser-width* nil)
NIL
Lisp> (setf *print-lines* 4)
4
Lisp> (format t "Stars: ~@!~/LINEAR/~."
          '(polaris dubhe mira mirfak bellatrix capella algol
            mirzam pollux canopus albireo castor alphecca
            antares))
Stars: POLARIS
       DUBHE
       MIRA
       MIRFAK ...
NIL
```

4.8 Using Miser Mode

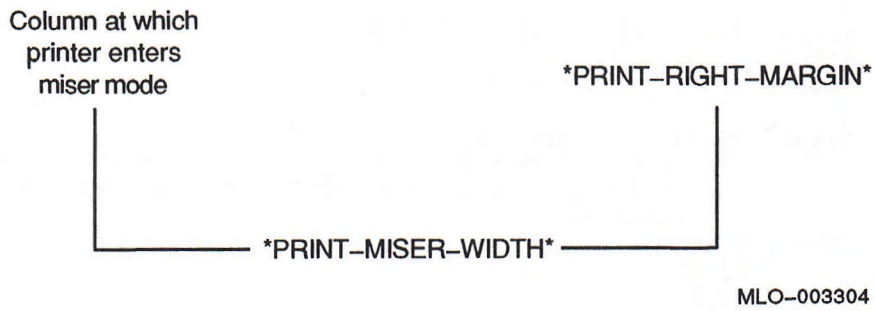
If you print large structures with deeply nested logical blocks, you may find the miser mode useful. Indentation produced in the output by the nesting of logical blocks, prefixes, and the `~nI` directive reduces the line length available for printing. Miser mode helps you avoid running out of space and printing beyond the right margin. Miser mode does not, however, guarantee the elimination of these problems.

Pretty-printing uses single-line mode if the output fits on one line. If the FORMAT control string permits new lines and the output requires two or more lines, pretty-printing normally uses multiline mode. The printer determines whether to print a logical block in miser mode according to the current column of the output at the beginning of the logical block and the values of two variables:

- *PRINT-RIGHT-MARGIN*
- *PRINT-MISER-WIDTH*

PRINT-RIGHT-MARGIN specifies the location of the right margin. *PRINT-MISER-WIDTH* specifies a number of columns before the right margin. When the current output column at the beginning of a logical block is equal to or greater than the difference between *PRINT-RIGHT-MARGIN* and *PRINT-MISER-WIDTH*, then the logical block is printed in miser mode. This condition occurs when the total available line width is less than the value of *PRINT-MISER-WIDTH*, as shown in Figure 4–1.

Figure 4–1: Variables Governing Miser Mode



You can disable miser mode by setting *PRINT-MISER-WIDTH* to NIL.

Miser mode saves space by:

- Ignoring indentation FORMAT directives
- Starting a new line at every conditional new line directive:

  Multiline mode new line (~_)
  If-needed new line (~:_)
  Miser mode new line (~@_)

The next two examples contrast pretty-printing in multiline mode and miser mode:

```
Lisp> (setf *print-pretty* t)
T
Lisp> (setf *print-right-margin* 60)
60
Lisp> (setf *print-miser-width* 35)
35
Lisp> (format t "~:!Stars with Arabic names: ~S ~S ~27I~:_~S ~
                 ~:I~@_~S ~_~S ~1I~_~S~."
          '(betelgeuse (deneb sirius vega)
            aldebaran algol (castor pollux) bellatrix))
Stars with Arabic names: BETELGEUSE (DENEB SIRIUS VEGA)
                           ALDEBARAN ALGOL
                                     (CASTOR POLLUX)
 BELLATRIX
NIL
Lisp> (format t "~!Stars with Arabic names:  ~:@!~S ~:_~S ~
                 ~27I~:_~S ~:I~@_~S ~_~S ~1I~_~S~.~."
          '(betelgeuse (deneb sirius vega)
            aldebaran algol (castor pollux) bellatrix))
Stars with Arabic names: BETELGEUSE
                         (DENEB SIRIUS VEGA)
                         ALDEBARAN
                         ALGOL
                         (CASTOR POLLUX)
                         BELLATRIX
NIL
```

In the first output sample, FORMAT uses multiline mode. Miser mode is never enabled, because the logical block begins at column 0 and miser mode takes effect only if the column begins at column 25 (60 – 35). ALDEBARAN lines up with the T in BETELGEUSE, because the ~27I directive sets the indentation for following lines at column 27 and the ~:_ directive produces a new line. The ~:I~@_~S directive sets the column for the next line at the level of the A in ALGOL. The ~1I directive controls the last argument, BELLATRIX, setting the indentation to column 1.

The second output example shows the effects of miser mode, because the text in the outer logical block, "Stars with Arabic names:", causes the inner logical block to begin at column 26. With *PRINT-MISER-WIDTH* set to 35 and *PRINT-RIGHT-MARGIN* set to 60, FORMAT enables miser mode when the logical block begins past column 25. FORMAT conserves space by starting a new line at every multiline mode new line directive (~_) and every if-needed new line directive (~:_). FORMAT also inserts a new line at the miser mode new line directive (~@_) and ignores the indentation directives (~*n*I).

4.9 Handling Improperly Formed Argument Lists

VAX LISP provides a method for gracefully handling argument lists that are improperly formed. The function of the ~^ directive, when used in a logical block, differs slightly from the corresponding function in Common LISP.

In Common LISP the ~^ directive is used with the iteration directives ~{ and ~} to check whether the argument list has been reduced to NIL. If the list is NIL, iteration stops.

You can also use the ~^ directive to check whether the argument list for a logical block has been reduced to a non-NIL atom. If the check shows that the argument list is a non-NIL atom, the printer prints space-dot-space (.) and uses the ~W directive to print the value of the atom. FORMAT then stops processing the immediately enclosing logical block, after printing the suffix (if one is there). No error condition results. The following example shows the use of FORMAT to print a dotted pair:

```
Lisp> (format t "~1:!~@{~S~^ ~}~."
        '(castor pollux deneb . aldebaran))
(CASTOR POLLUX DENEB . ALDEBARAN)
```

This feature serves as a useful debugging tool, because it lets the FORMAT function work even when the argument list is improperly formed.

NOTE

When the ~^ directive is included in a logical block, the FORMAT function checks whether the argument list is a non-NIL atom, even when pretty-printing is not enabled.

Chapter 5

Error Handling

The LISP system invokes the VAX LISP error handler when errors are signaled during program evaluation. This chapter explains what the error handler does when an error is signaled. Because the system's error handler may not meet your programming needs, VAX LISP lets you create your own error handler. The procedure for creating an error handler is also explained in this chapter.

5.1 Error Handler

The VAX LISP error handler function, `UNIVERSAL-ERROR-HANDLER`, performs four sequential steps:

1. Checks the number of nested errors that have occurred. If three nested errors have occurred, the error handler aborts your program, displays a message, and returns you to the top-level read-eval-print loop; otherwise, the handler continues to the next step.
2. Checks the type of error.
3. Displays an error message that provides you with information about the error.
4. Performs the appropriate operation for the type of error that was signaled.

5.2 VAX LISP Error Types

Three types of errors can occur during the evaluation of a LISP program:

- Fatal error
- Continuable error
- Warning

When an error is signaled, the VAX LISP system displays an error message that provides you with the following information:

- The type of error that was signaled—fatal error, continuable error, or warning
- The name of the function that caused the error
- The name of the function that was used to signal the error—`ERROR`, `CERROR`, or `WARN`
- A description of the error
- If a continuable error, an explanation of what will happen if you continue the program's evaluation from the point at which the error occurred

The format of an error message and the information a message provides depend on the type of error. The next three sections describe the types of errors; each description includes the error type's message format and the operation the error handler performs.

5.2.1 Fatal Errors

When a fatal error is signaled, the error handler displays a message in the following format:

Error in *function-name*: *error-description*.

In this format description, *function-name* is the name of the function that caused the error, and *error-description* is a message telling why the error occurred. The message is generated from the format string and the arguments in the call to the ERROR function; the message can be displayed on more than one line.

An example of a fatal error message follows:

```
Error in SYSTEM::MAKE-ARRAY-NK: Only vectors can have fill pointers.
```

After the message is displayed, the error handler checks the value of the VAX LISP *ERROR-ACTION* variable. This value is set when you invoke VAX LISP and can be either the :EXIT or the :DEBUG keyword. When the value is :EXIT (the default in a batch session), the error handler causes the LISP system to exit on an error; when the value is :DEBUG (the default in an interactive session), the handler invokes the VAX LISP debugger.

On VMS systems, you use a command qualifier to set the value of the *ERROR-ACTION* variable:

$ LISP/ERROR_ACTION=*value*

On ULTRIX systems, you use an option to set the value of the *ERROR-ACTION* variable:

% vaxlisp -V ERROR_ACTION=*value*

In either case, the *value* must be either EXIT or DEBUG.

If the debugger is invoked, you can use it to locate the error in your program. After you locate the error, you can correct it and restart your program's evaluation.

NOTE

> You cannot continue your program's evaluation from the point at which a fatal error occurred.

The *ERROR-ACTION* variable is described in the *VAX LISP/VMS Object Reference Manual*, and the debugger is described in Chapter 4 of the *VAX LISP/VMS Program Development Guide*.

5.2.2 Continuable Errors

When a continuable error is signaled, the error handler displays a message in the following format:

Continuable error in *function-name*: *error-description*.

Control Stack Debugger
If continued: *continue-explanation*.

In the preceding format description, *function-name* is the name of the function that caused the error, and *error-description* is a message telling why the error occurred. The message is generated from the format string and the arguments in the call to the CERROR function; the message can be displayed on more than one line. A line of text that explains what will happen if you continue your program's evaluation follows the error description.

An example of a continuable error message is:

```
Continuable error in FACTORIAL: Too few arguments: 0

Contro Stack Debugger
If continued: Proceed with unsupplied arguments defaulting to NIL.
```

After the message is displayed, the error handler checks the value of the VAX LISP *ERROR-ACTION* variable in the same way it checks the value after a fatal error (see Section 5.2.1).

If the debugger is invoked, you can do one of the following:

- Continue from the error; the CERROR function performs the corrective action that is specified in the error message.
- Locate the error in your program. After you locate the error, you can correct it and restart your program's evaluation.

The *ERROR-ACTION* variable is described in the *VAX LISP/VMS Object Reference Manual*, and the debugger is described in Chapter 4 of your *Program Development Guide*.

5.2.3 Warnings

A warning is an error condition that may or may not affect your program's evaluation. When this type of error occurs, the system displays a message for the following reasons:

- You might want to correct the error later.
- Your program might correct the error, but you should know that the error occurred.

When a warning is signaled, the error handler displays a message in the following format:

Warning in *function-name*: *error-description.*

function-name is the name of the function that caused the error, and *error-description* is a message telling why the error occurred. The message is generated from the format string and the arguments in the call to the WARN function; the message can be displayed on more than one line.

An example of a warning error message is:

```
Warning in function TE: 3 is not a symbol.
```

After the message is displayed, the error handler checks the value of the *BREAK-ON-WARNINGS* variable in the same way it checks the value *ERROR-ACTION* variable after a fatal error (see Section 5.2.1).

NOTE

If the value of the *BREAK-ON-WARNINGS* variable is T, the debugger is invoked when a warning is signaled.

If the debugger is invoked, you can use it to locate the error in your program. After you locate the error, you can correct it, exit the debugger, and then continue your program's evaluation from the point where the error occurred.

The *BREAK-ON-WARNINGS* variable is described in *Common LISP: The Language*.

5.3 Creating an Error Handler

The VAX LISP *UNIVERSAL-ERROR-HANDLER* variable is bound to the system's error handler. This binding provides you with a way to create your own error handler if the system's handler does not meet your programming needs. To create an error handler you must:

1. Define the error handler.
2. Bind the *UNIVERSAL-ERROR-HANDLER* variable to your defined handler.

The *UNIVERSAL-ERROR-HANDLER* variable is described in the *VAX LISP/VMS Object Reference Manual*.

5.3.1 Defining an Error Handler

To define an error handler, you must define an error handler function. This function must be able to accept two or more arguments because the LISP system passes at least two arguments to the error handler each time an error occurs in a program. Therefore, specify the arguments in an error-handler definition in the following format:

(DEFUN *handler* (*function-name error-signaling-function* &REST *args*) . . .)

The arguments provide the error handler with the following information:

- The name of the function that called the error-signaling function
- The name of the error-signaling function
- The arguments that were passed to the error-signaling function

An example of an error handler definition is:

```
Lisp> (defun critical-error-handler (function-name
                                     error-signaling-function
                                     &rest args)
        (when (or (eq error-signaling-function 'error)
                  (eq error-signaling-function 'cerror))
              (flash-alarm-light))
        (apply #'universal-error-handler
               function-name
               error-signaling-function
               args))
CRITICAL-ERROR-HANDLER
```

This error handler checks whether a fatal or continuable error is signaled. If either type of error is signaled, the handler calls the function FLASH-ALARM-LIGHT and then passes the error signal information to the VAX LISP error handler.

When you define an error handler, the definition can include a call to the UNIVERSAL-ERROR-HANDLER function. If the definition does not include a call to this function and you want the handler to check the value of the *ERROR-ACTION* or *BREAK-ON-WARNINGS* variable, you must include a check of the variable in the handler's definition.

If you want an error handler to display error messages in the formats described in Sections 5.2.1 through 5.2.3, include a call to either the UNIVERSAL-ERROR-HANDLER or PRINT-SIGNALED-ERROR function. Descriptions of these functions are provided in the *VAX LISP/VMS Object Reference Manual*.

The next three sections describe the arguments an error handler must be able to accept.

5.3.1.1 Function Name

The *function-name* argument is the name of the function that calls an error-signaling function. This argument enables the error handler to include the function's name in the error message the handler displays.

5.3.1.2 Error-Signaling Function

The *error-signaling-function* argument is the name of the error-signaling function that is called to generate the error signal. Depending on which function is called, a fatal error, continuable error, or warning is signaled.

The error handler uses the *error-signaling-function* argument to determine the contents of the *args* argument.

Table 5–1 lists the functions that can be passed as the *error-signaling-function* argument and briefly describes each function.

Table 5–1: Error-Signaling Functions

Function	Description
CERROR	Signals a continuable error
ERROR	Signals a fatal error
WARN	Signals a warning

See *Common LISP: The Language* for detailed descriptions of the CERROR and ERROR functions. See the *VAX LISP/VMS Object Reference Manual* for a description of the WARN function.

5.3.1.3 Arguments

The *args* argument is the list of arguments passed to the error-signaling function when the error-signaling function is invoked. The contents of the list depends on which function is invoked. The list can include one or two format strings and their corresponding arguments. The format strings and arguments are passed to the FORMAT function, which produces the correct error message.

5.3.2 Binding the *UNIVERSAL-ERROR-HANDLER* Variable

Once you define an error-handling function, you must bind the *UNIVERSAL-ERROR-HANDLER* variable to it. The following example shows how to bind the variable to a function:

```
Lisp> (let ((*universal-error-handler*
             #'critical-error-handler))
        (perform-critical-operation))
```

The LET special form binds the *UNIVERSAL-ERROR-HANDLER* variable to the CRITICAL-ERROR-HANDLER function that was defined in Section 5.3.1 and calls a function named PERFORM-CRITICAL-OPERATION. When the form is exited because the evaluation finished or the THROW function is called, the *UNIVERSAL-ERROR-HANDLER* variable is restored to its previous value.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

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 Implementation and Extensions to Common LISP
AA–MK70A–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