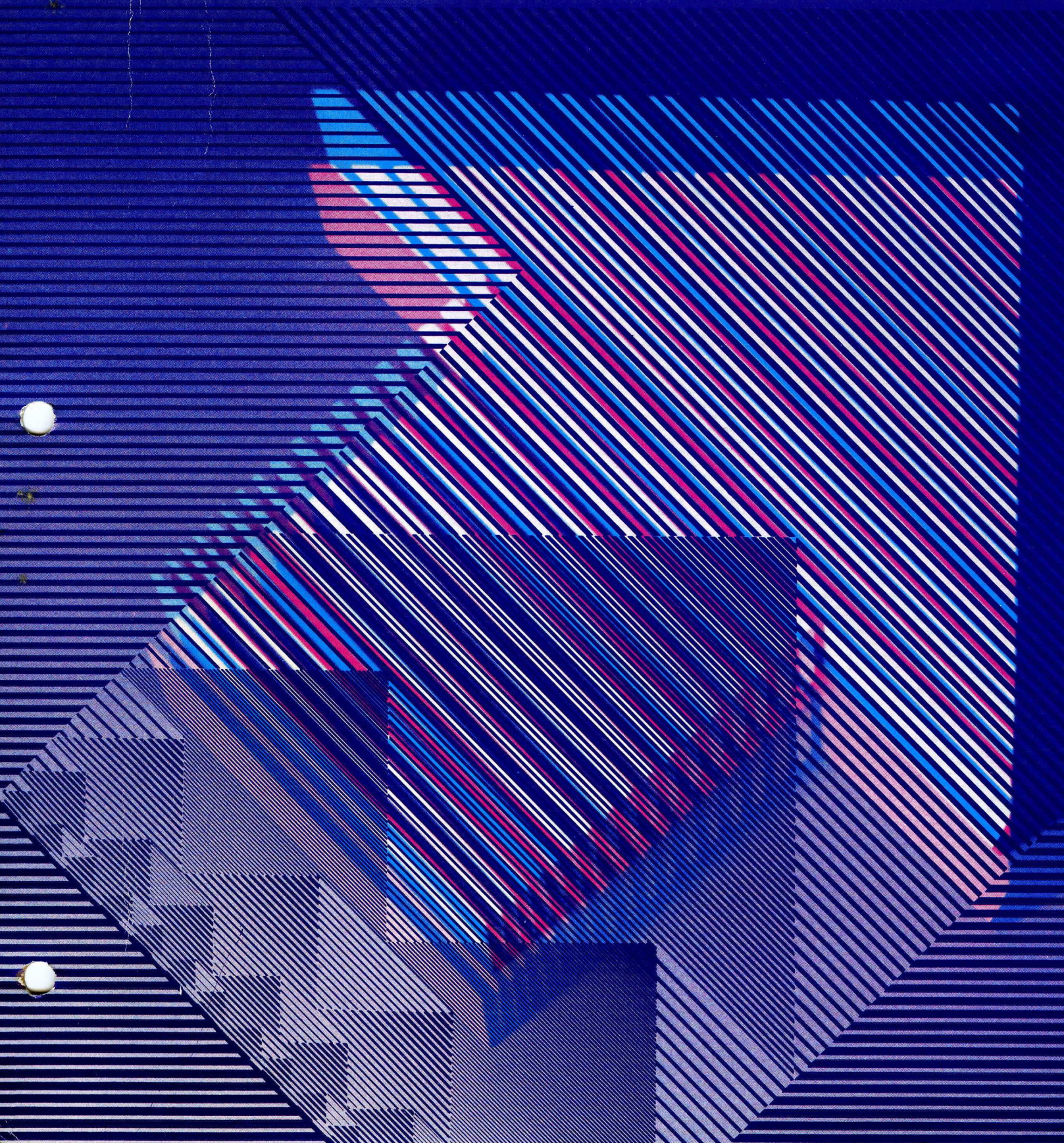
Release 6.0
Release Notes

symbolics



Release 6.0 Bulletin

symbolics

Cambridge, Massachusetts

Release 6.0 Bulletin

March 1985

This document corresponds to Release 6.0 and later releases.

Text written and produced on Symbolics 3600-family computers by the Home Office Software Support Group of Symbolics, Inc.

Text typography: Century Schoolbook and Helvetica produced on Symbolics 3600-family computers from Bitstream, Inc., outlines; text masters printed on Symbolics LGP-1 Laser Graphics Printers.
Cover design: Schafer/LaCasse
Cover printer: W.E. Andrews Co., Inc.
Text printer: ZBR Publications, Inc.

Printed in the USA.

Printing year and number: 87 86 85 9 8 7 6 5 4 3 2 1

NOTE: To add the included patches to your world, enter them into an editor buffer, making sure that you specify the correct package for each,save out the file, compile it, boot a clean world, and load the file before disk-saving.

Problem Description:

Reducing the size of your LMFS

Solution:

Several users have tried to reduce the the size of their LMFS by deleting one or more file partitions and editing the FSPT to remove references to these partitions. Other users have tried to simply remove the names of unwanted partition files from their FSPT. Both of these methods will leave you with an unuseable LMFS.

Do not delete file partitions from your LMFS or remove entries from your FSPT. Each LMFS partition contains pointers to all other file partitions in the LMFS. Deleting a file partition leaves the LMFS in an inconsistent state.

If you want to reduce the size of your LMFS, you must completely backup your LMFS, delete the entire existing LMFS including the current FSPT and initialize a new LMFS using the file system maintenance window. User files may be restored into this new LMFS from the backup tapes.

Problem Description:

Serial stream handling of XON - XOFF characters

Solution:

A common problem encountered with serial streams is the handling of the XON/XOFF protocol. The problem arises because the FEP reads all eight bits of the XON or XOFF character even though you may have specified a different number of data bits for that stream. You must determine what eight bit characters are being sent to the LISP machine as the XON and XOFF characters.

For example, assume that the printer connected to the LISP machine's serial port receives seven data bits with no parity. One might assume that it would send a Control-S (#O23) as the XOFF character and a Control-Q (#O21) as the XON character. The FEP, however, may be receiving #O221 as the XON character and #O223 as the XOFF character. The difference here is that the in both cases the parity bit

of each character is set.

The :OUTPUT-XON-CHARACTER and :OUTPUT-XOFF-CHARACTER options of SI:MAKE-SERIAL-STREAM are used to change to the character that the FEP will recognize as the XON or XOFF character. Similarly, add the OUTPUT-XON-CHARACTER and OUTPUT-XOFF-CHARACTER options to the Interface Options of the printer's namespace object when connecting a serial ASCII printer.

Problem Description:

Fep returns the error: "Request for xxx longs failed"

Solution:

This error occurs because the Fep runs out of memory to build data structures which tell lisp where to find the rest of the world on the disk.

There are two possible solutions:

- Use the FEP's "Reset Fep" command. This allows the Fep to start over with allocation of its memory used for parsing and building these data structures.
- Type in the contents of the boot command file by hand. This frees up memory which would otherwise be used for parsing the boot file.

Problem Description:

Disk-saving the same world twice before cold-booting.

Solution:

Some users have attempted to disk-save the same world twice without cold-booting between the two disk-saves. The world load produced by the second disk save cannot be booted. If you want to disk save two copies of a world, you should disk-save the first copy, cold-boot and disk save the second copy.

Problem Description:

Bus turn timeout error while using the local tape drive.

Solution:

When reading or writing tapes on the local tape drive, you may encounter the error:

>>Hard tape error: Fep/Command error:1101-16.(Bus turn timeout)

The error is the result of a timing problem between the FEP and the tape drive.

A way around this bug that usually works is to c-N down one frame to the caller of the function that blew out. Then c-m-R to re-evaluate that call. The operation should succeed this time.

Problem Description:

How BREAK and DBG interact with lexical scoping

Solution:

A major difference in the LISP interpreter between Release 5.2 and 6.0 is that the interpreter is now lexically scoped. In prior releases, the interpreter was dynamically scoped. To understand the full ramifications of lexical scoping consult the discussion of lexical scoping in the documentation.

A common practice used in earlier releases was to evaluate in the interpreter a form similar to:

```
(with-open-file (stream "ABT:>Gelsey.Kirkland") (BREAK Got-It))
```

In Release 5.2, the variable 'stream' was dynamically scoped, but in Release 6.0 the variable is lexically scoped. If you like using dynamic variables in this way, keep around some 'meta-variables' that are declared 'Special' for this purpose.

Problem Description:

FEP file locked (cannot be expunged)

Occasionally, you may have trouble deleting and expunging FEP files.

If their filenames appear twice when using DIRED or **(print-disk-label)**, and can be marked for deletion but not expunged, do the following:

Solution:

Run the function **(si:verify-fep-filesystem)**

It will take at least a minute to run, maybe longer. You should now be able to expunge the FEP directory.

Problem Description:

Sending the :finish message to a serial stream sometimes hangs in a 'Serial Finish' state.

Solution:

The following patch solves the problem.

```
-*- Package: SYSTEM-INTERNALS -*-

(DEFUN FEP-CHANNEL-NOT-EMPTY
       (CHANNEL &OPTIONAL INCLUDE-CACHE)

  (COMMENT
    (OR (> (FEP-CHANNEL-N-USED-BYTES CHANNEL) 0)
        (AND INCLUDE-CACHE
             (> (FEP-CHANNEL-FEP-INTERNAL-CACHE CHANNEL) 0))))

  (WHEN (MINUSP (FEP-CHANNEL-FEP-INTERNAL-CACHE CHANNEL))
    (SETF (FEP-CHANNEL-FEP-INTERNAL-CACHE CHANNEL) 0))

  (NOT
    (LOOP WITH START-MICROSECOND-TIME =
            (TIME:FIXNUM-MICROSECOND-TIME)
          ALWAYS (ZEROP (FEP-CHANNEL-N-USED-BYTES CHANNEL))
          ALWAYS (OR (NOT INCLUDE-CACHE)
                     (ZEROP (FEP-CHANNEL-FEP-INTERNAL-CACHE CHANNEL)))
          UNTIL (TIME-ELAPSED-P 10000.
                                START-MICROSECOND-TIME
                                (TIME:FIXNUM-MICROSECOND-TIME)))))
```

Problem Description:

(QSEND "foo@bar") works, but **(QSEND foo@bar)** sends you into the debugger.

Solution:

The following patch corrects the problem.

```
-*- Package: ZWEI -*-

(DEFMACRO QSEND (&OPTIONAL DESTINATION MESSAGE)
  `(QSEND-MSG ',DESTINATION ',MESSAGE))
```

Problem Description:

UNIX ASCII print server does not print carriage returns.

Solution:

Users that installed the ASCII print server on their UNIX systems found that files from their LISP machines were printed without any carriage returns. The problem is that neither the LISP machine nor the UNIX machine is translating from the the LISP machine character set into ASCII.

The following patch allows users to choose whether their hardcopy streams should translate into ASCII.

```
-*- Package: LGP -*-
(DEFVAR *ENABLE-ASCII-TRANSLATION-FOR-HARDCOPY* T "Controls whether
or not we use ASCII translation")

(DEFUN INVOKE-SERVICE-HARDCOPY-WITH-LGP (NETI:.SERVICE.)
  (NET:GET-CONNECTION-FOR-SERVICE NETI:.SERVICE.
    :ASCII-TRANSLATION *ENABLE-ASCII-TRANSLATION-FOR-HARDCOPY*))
```

Problem Description:

IP/TCP spawns too many "UDP Service" processes.

Solution:

Some versions of UNIX generate UDP packets with incorrect checksums. LISP machines with IP/TCP that are on the same network as these UNIX machines will eventually crash because their IP/TCP software does not correctly handle this error and generates too many "UDP Service" processes.

The following patch corrects this problem.

```
(net:define-server :unix-rwho (:medium :udp :connection icp-conn)
  (let ((host (send icp-conn :foreign-host)))
    (unless (send host :uninterned-p)
      (multiple-value-bind (pkt start end)
          (send icp-conn :read-input-buffer t t)
        (when pkt
          (let ((copy (make-array (- end start) :type 'art-string)))
            (copy-array-portion pkt start end copy 0 (- end start))
            (setf (get (locf *rwho-messages*) host) copy)))))))
```

Problem Description:

Local host name is dropped when completing pathnames with wildcard directory specifications.

Solution:

If you try to complete a pathname that contains a accordian wildcard directory specification, "**>", and the name of the local host, the host name will be dropped from the resulting pathname. For example,

Show File my-host:>foo>*.*.*[Complete]

correctly completes the pathname, but

Show File my-host:>foo>**>*.*.*[Complete]

drops the host name from the resulting pathname

The following patch corrects this problem.

```
(defmethod (local-lmfs-access-mixin :complete-string)
           (pathname string options)
  (block complete-string
    (let* ((host-name (send fs:host ':name-as-file-computer))
           (newparse
             (condition-case (val)
                 (fs:parse-pathname string net:*local-host*)
               (fs:pathname-error
                 (return-from complete-string
                   (values
                     (if (string-search ":" string)
                         string
                         (string-append host-name ":" string))
                     nil))))))
      (setq newparse
            (send newparse
                  ':new-raw-directory
                  (send (fs:merge-pathnames newparse pathname)
                        ':raw-directory)))
      (condition-case (result flag)
          (with-path-vars pathname
                          (nil default-name
                               default-type default-version)
            (send self
                  ':complete-pathname newparse string options
                  default-name default-type default-version))
        (error (values (if (string-search ":" string)
                           string
                           (string-append host-name ":" string))
                       nil))
        (:no-error
           (values
             (if (string-equal host-name "UNKNOWN")
                 result
                 (string-append host-name ":" result))
             flag))))))

)
```

Problem Description:

SETQing symbol macros sends you into the debugger.

Solution:

Many things that were variables in Release 5.2 are now symbol macros and require that you use **SETF** instead of **SETQ** to set their value.
For example, many of the parameters in the ZWEI package have been changed from variables to symbol macros. If you wanted to set the value of zwei:*default-major-mode* you must evaluate the form:

```
(setf zwei:*default-major-mode* :fundamental)
```

Table of Contents

1. Release 6.0: Introduction and Highlights

These notes accompany the release of Release 6.0. They describe changes made since Release 5.2. The release notes contain brief descriptions of the changes and pointers to the appropriate sections of the documentation. These notes contain some recent information that is not reflected in the other documentation. They are the authoritative source in cases where the documents disagree.

As in previous releases, many minor bugs have been fixed and performance in some areas has been improved. Only the more important or visible changes are mentioned here.

Two new features, the Command Processor and the Document Examiner, the online documentation system, make it possible to read the documentation online. For help in using these new facilities:

See the section "Communicating with the Lisp Machine" in *User's Guide to Symbolics Computers*.
See the section "Using the Online Documentation System" in *User's Guide to Symbolics Computers*.

Within each section of these release notes, the material is organized into incompatible changes, new features, and improvements. You can find all the incompatible changes by reading the first part of each section. A complete list of changes appears in the Table of Contents. The notes cover the following topics:

Changes to the Lisp Language in Release 6.0
This section describes changes relevant to the Lisp language. The major changes include the following:

- The default value of **base** and **ibase** has been changed from 8 to 10.
- The compiler and the interpreter have both been modified to use lexical scoping.
- Additions have been made to Symbolics-Lisp to support Common Lisp character objects in the future.
- Files that use **defwrapper** forms must be recompiled to work in Release 6.0.
- Heaps are now supported. Heaps are data structures in which each item is ordered by some predicate on its associated key.

New Feature in Release 6.0: Symbolics Common Lisp
Symbolics Common Lisp is available in Release 6.0.

Changes to Zmacs in Release 6.0
This section describes changes in the Zmacs editor.

Changes to Utilities in Release 6.0
This section describes changes in what any other computer would call the operating system and utilities. This includes the Debugger, the Inspector, the garbage collector, and various system keyboard features. The most important changes are the following:

- The Ephemeral-Object Garbage Collector has been added.
- A new online documentation lookup facility, the Document Examiner, is available.

Changes to the User Interface in Release 6.0
This section describes changes to the user interface, including the window system. The most important changes are:

- A new utility program, the Command Processor, has been added.
- A new notification system has been installed.
- Window selection has been improved.

Changes to Zmail in Release 6.0
This section describes changes in Zmail, the program for reading and sending mail.

Changes to the File System in Release 6.0
This section describes changes in the Lisp Machine File System. The most important change is logical pathname translation.

Changes to Networks in Release 6.0
This section describes changes in network implementation, interface, and protocols. The most important change is the addition of the remote login facility.

Changes to the FEP in Release 6.0
This section describes changes in the FEP. Release 6.0 requires one of: FEP version number 17, 18, 22, or 24.

Notes and Clarifications for Release 6.0
This section contains explanations and clarifications of items that people found confusing in previous releases and documentation.

1.1 Release 6.0 Documentation Changes

The documentation for Release 6.0 includes previously published Symbolics Lisp Machine documentation as well as new documents. The material has been reorganized by topic and intended use of information. The most obvious changes to the Release 6.0 documentation are the following:

- A new *User's Guide to Symbolics* computers is included. This book contains information that is useful for both new and experienced users of Symbolics computers.

- Information has been reorganized to place as much related material as possible within the same book.

- The documentation is packaged in perfect-bound books rather than loose-leaf binders.

The following table summarizes the new documentation set:

Installation and Site Operations - Book 0
: Book 0 includes the *Software Installation Guide* and information for managing site operations.

User's Guide to Symbolics Computers - Book 1
: Book 1 presents an introduction to using Symbolics computers and provides the most commonly needed information about the system.

Reference Guide to Symbolics-Lisp - Book 2
: Book 2 provides conceptual and reference material for the Symbolics-Lisp language, as well as Symbolics Common Lisp.

Text Editing and Processing - Book 3
: Book 3 includes the Zmacs, font editor, and hardcopy system documentation.

Program Development Utilities - Book 4
: Book 4 presents information that is useful for developing programs on computers. This includes the *Program Development Tools and Techniques*, *Maintaining Large Programs*, *Debugger*, *Compiler*, *Inspector*, and *Peek* documents.

Reference Guide to Streams, Files, and I/O - Book 5
: Book 5 provides information about streams, I/O, the FEP file system, the Lisp Machine File System (LMFS), and the generic file system.

Communicating with Other Users - Book 6
: Book 6 provides documentation for the Zmail and Converse utilities.

Programming the User Interface - Book 7
: Book 7 describes the window system, scrolling, menus, the digital audio facility, and the command processor program interface.

Internals, Processes, and Storage Management - Book 8
: Book 8 describes the internals of the Symbolics Lisp Machine system and includes initializations, storage management and garbage collection, and processes.

Networks - Book 9
: Book 9 includes information on networks and peripherals, network protocols, the namespace system, and the front-end processor (FEP).

System Index - Book 10
Book 10 contains index entries from all books in the set.

Notes and Bulletins - RN
The last book includes these Release Notes, as well as newsletters and bulletins.

Release 5.0 to Release 6.0 Documentation Map

The Documentation Map Table below shows how the Release 5.0 documentation has been reorganized for Release 6.0. Information has been consolidated to place as much related material as possible within the same book. In many cases, the Release 5.0 document is now a chapter of a Release 6.0 book. In other cases, information from a Release 5.0 document has been merged into more than one document.

Documentation Map Table

Release 5.0 Title	Release 6.0 Title	
3600 Serial I/O Facility	*Reference Guide to Streams, Files, and I/O*	5
Arrays and Strings	*Reference Guide to Symbolics-Lisp*	2
Compiler	*Program Development Utilities*	4
Conditions	*Reference Guide to Symbolics-Lisp*	2
Converse	*Communicating With Other Users*	6
Debugger	*Program Development Utilities*	4
Defstruct	*Reference Guide to Symbolics-Lisp*	2
Evaluation	*Reference Guide to Symbolics-Lisp*	2
Files	*Reference Guide to Streams, Files, and I/O*	5
Flow of Control	*Reference Guide to Symbolics-Lisp*	2
Font Editor	*Text Editing and Processing*	3
FSedit	*Reference Guide to Streams, Files, and I/O*	5
Functions	*Reference Guide to Symbolics-Lisp*	2

Hardcopy System	*Text Editing and Processing*	*3*
Initializations	*Internals, Processes, and Storage Management*	*8*
Internals	*Internals, Processes, and Storage Management*	*8*
Lisp Language	*Reference Guide to Symbolics-Lisp*	*2*
Lisp Machine Summary	*User's Guide to Symbolics Computers*	*1*
Macros	*Reference Guide to Symbolics-Lisp*	*2*
Maintaining Large Systems	*Program Development Utilities*	*4*
Miscellaneous Functions	[Merged into related documents.]	
Miscellaneous Useful Functions	[Merged into related documents.]	
Networks and Peripherals	*Networks*	*9*
Networks and Protocols	*Networks*	*9*
Notation Conventions	*User's Guide to Symbolics Computers*	*1*
Notes on the 3600 for LM-2 Users	[Merged into related documents.]	
Objects, Message Passing, and Flavors	*Reference Guide to Symbolics-Lisp*	*2*
Other Tools	[Merged into related documents.]	
Other Utilities and Applications	[Merged into related documents.]	
Packages	*Reference Guide to Symbolics-Lisp*	*2*
Primitive Object Types	*Reference Guide to Symbolics-Lisp*	*2*
Processes	*Internals, Processes, and Storage Management*	*8*
Program Development Help Facilities	*User's Guide to Symbolics Computers*	*1*

Program Development Tools and Techniques	*Program Development Utilities*	*4*
Release 5.0 Release Notes	[Merged into related documents.]	
Release 5.1 Release Notes	[Merged into related documents.]	
Release 5.2 Release Notes	[Merged into related documents.]	
Scroll Windows	*Programming the User Interface*	*7*
Site Operations	*Installation and Site Operations*	*0*
Software Installation Guide	*Installation and Site Operations*	*0*
Storage Management	*Internals, Processes, and Storage Management*	*8*
Streams	*Reference Guide to Streams, Files, and I/O*	*5*
Tape	*Installation and Site Operations*	*0*
Using the Input Editor	*Programming the User Interface*	*7*
Using the Window System	*Programming the User Interface*	*7*
Window System Choice Facilities	*Programming the User Interface*	*7*
Window System Program Examples	*Programming the User Interface*	*7*
Zmacs	*Text Editing and Processing*	*3*
Zmail Concepts and Techniques	*Communicating With Other Users*	*6*
Zmail Tutorial and Reference Manual	*Communicating With Other Users*	*6*

1.2 New Microcode in Release 6.0: 319.

Release 6.0 requires microcode 319.

1.3 Release 6.0 is Supported Only on 3600-family Machines

Release 6.0 is supported only on the 3600-family machines. It is not supported on LM-2s.

2. Changes to the Lisp Language in Release 6.0

2.1 Recompiling Source Files is Recommended

In general, code compiled in Release 5.0 will work in Release 6.0. However, you are advised to recompile source files to take advantage of compiler improvements and bug fixes.

Any files containing **defwrapper** forms or forms defined with **&quote** must be recompiled in Release 6.0.

You can maintain separate versions of compiled code in separate systems (a Release 5.0 system and a Release 6.0 system) by using **make-system**. For further information: See the section "Making a System" in *Program Development Utilities*.

2.2 Incompatible Changes to Lisp in Release 6.0

2.2.1 Change of Default Base to Decimal

The default value of **base** and **ibase** has been changed from 8 to 10 for Release 6.0.

2.2.2 Syntax and Base Attributes in Source Files

The editor and compiler now recognize a Syntax attribute in conjunction with the Base attribute which existed in previous releases. The syntax of a program can be either Zetalisp or Common-Lisp. The mode line (the -*- line in Lisp source files) indicates which syntax the source file has.

- If there is a Base attribute, but no Syntax attribute, the syntax is assumed to be Zetalisp.
- If there is a Syntax: Common-Lisp attribute, and no Base attribute, the base is assumed to be 10.
- If there is neither a Base nor a Syntax attribute, Base is assumed to be the default base (10) and the syntax is assumed to be Zetalisp. Furthermore, a warning is issued (upon beginning an editing session on the file) to the effect that there is neither a Syntax nor a Base attribute. You should edit your program accordingly. With most programs, the Zmacs command Update Attribute List (m-X) adds the appropriate attributes to the mode line, following the above defaults.

2.2.3 Setting Variables in Init Files Has Changed

Previously you used **setq-globally** to set certain variables in your init file, for instance to change the input and output base. In Release 6.0 the new function **setq-standard-value** should be used for setting such interactive variables. For example:

```
(login-forms
        (setq-standard-value base 8)
        (setq-standard-value ibase 8)
        (defun bar (x y) (+ x y))
        (quux 3))
```

See the section "Standard Variable Bindings Now Guarantee Consistent Behavior in Break and Debugging Loops", page 42.

2.2.4 Lexical Scoping in Release 6.0

Lexical scoping has been implemented for both the compiler and the interpreter in Release 6.0. This means that certain new ways of using variables are supported. Also, certain usages that formerly worked only in the interpreter now work in the compiler as well, and vice versa. Zetalisp and Symbolics Common Lisp both use the same lexically scoped compiler and interpreter.

Both the compiler and the interpreter support the accessing of lexical variables. The compiler and interpreter also support, in Zetalisp as well as Symbolics Common Lisp, the Common Lisp lexical function and macro definition special forms, **flet**, **labels**, and **macrolet**.

For a detailed description of the Release 6.0 implementation of lexical scoping: See the section "Lexical Scoping" in *Reference Guide to Symbolics-Lisp*.

This section provides an overview of the changes in Release 6.0 that are related to the implementation of lexical scoping.

2.2.4.1 Funargs Supported in Release 6.0

Release 6.0 supports the use of *funargs*; the term funarg is an acronym for *functional argument*. A funarg is a function that is passed as an argument, stored into a data structure, or otherwise manipulated as data. Normally, functions are simply called, not manipulated as data. The major feature of the lexical compiler and interpreter can be described as the support of funargs that refer to free lexical variables. Funargs that do not refer to free lexical variables also work.

See the section "Funargs and Lexical Closure Allocation" in *Reference Guide to Symbolics-Lisp*.

2.2.4.2 New Special Forms for Lexical Scoping

Three new special forms have been added to support lexical scoping: **flet**, **labels**, and **macrolet**. **flet** and **labels** are used to define, within their scope, a function. **macrolet** is used to define, within its scope, a macro. See the section "**flet**, **labels**, and **macrolet** Special Forms" in *Reference Guide to Symbolics-Lisp*.

2.2.4.3 Changes to Macro Expansion

macroexpand, **macroexpand-1**, and **si:*macroexpand-hook*** now behave as documented in the Digital Press *Common Lisp* manual (*CLM*).

The meaning of the optional second argument to **macroexpand** and **macroexpand-1** has changed; the second argument is now a lexical environment, which can be supplied to specify the lexical environment of the expansions, as discussed. **macroexpand** and **macroexpand-1** now also accept an optional third argument, **dont-expand-special-forms**, which prevents macro expansion of forms that are both special forms and macros.

Functions used as the value of **si:*macroexpand-hook*** now require a second argument, the lexical environment, except when expanding a lambda macro. Lambda macros do not have a lexically scoped environment. This should cause problems only if you are using Symbolics Common Lisp and the Interlisp Compatibility Package at the same time.

Programs should never need to create a lexical environment, however the lexical environment is usually available as an option (for example, **&environment** in **defmacro**).

2.2.4.4 Changes to evalhook and applyhook

evalhook and **applyhook** now also take an optional environment argument, as specified in the *CLM*.

2.2.4.5 Changes to Special Forms

Old special forms (forms defined with **&quote** and compiled in previous releases) must be recompiled. If these forms are recompiled, they are automatically translated into appropriate calls to **si:define-special-form**. The compiler inserts calls to **eval**, with the appropriate environment, for any arguments that are not **&quote**d. In many cases, the special form will not work after recompilation; these cases are no longer supported.

A special form defined this way works only when called from the interpreter. You cannot use it in code that is to be compiled. It is better to use a macro, unless your special form is intended to be used only at top level (for example, **defvar**), not inside a function.

Note: You can no longer call a special form from compiled code, no matter whether it is defined with **si:define-special-form** or with **&quote**, unless you tell the compiler how to compile it. The old code in the compiler that used to allow you to call special forms in certain cases has been removed, since it is no longer supported. In all cases, user-defined special forms should be implemented by macros.

Support for dynamic closures of special forms has been removed.

2.2.4.6 Changes to Conditions

The **sys:funcall-macro** condition and its proceed type have been removed.

The **sys:invalid-form** condition has been removed. All atoms other than symbols now evaluate to themselves.

The **sys:invalid-lambda-list** condition has been removed. Invalid lambda lists now produce **ferrors**.

The **sys:invalid-function** condition is signalled but is not proceedable.

2.2.4.7 Miscellaneous Lexical Scoping Changes

si:lexical-closure and **global:functional-alist** have been removed.

The **defmacro** lambda-list keyword **&list-of** is no longer supported.

lambda now is a special form, which evaluates to the lexical closure of the lambda-expression it represents. Thus,

```
(sort list (lambda (x y) (fun x y)))
```

is equivalent to

```
(sort list #'(lambda (x y) (fun x y)))
```

2.2.5 Common Lisp Character Switchover in Release 6.0

This section describes the changes to Zetalisp for characters and strings in a future major release and provides information to aid in converting before then.

2.2.5.1 Character Objects in Common Lisp

Zetalisp has always used positive integers to represent characters. This makes it possible to use arithmetic operations such as =, <, +, and **ldb** to perform various operations on characters. Common Lisp, on the other hand, has a separate data type for characters and specialized functions for operations on characters. In Common Lisp, it is possible to distinguish unambiguously between an integer and a character; characters print out in #\ notation.

This incompatibility between Zetalisp and Common Lisp affects strings as well as characters. A string is defined to be a one-dimensional array of characters, so in Zetalisp **aref** of a string returns an integer, but in Common Lisp **aref** of a string returns a character.

Eventually Zetalisp will be replaced with Symbolics Common Lisp, an extension of the standard Common Lisp language that also contains all of the advanced features of Zetalisp. System programs and most user programs will be written in Symbolics Common Lisp; old programs will continue to be supported by a Zetalisp compatibility package. In Release 6.0, Symbolics Common Lisp is present, but the default language dialect is still Zetalisp. To make it practical for Zetalisp and Symbolics

Common Lisp programs to coexist in the same world and call each other freely, it is necessary for them to use compatible data types. If the two languages used different representations for characters, they could not coexist conveniently, as characters and strings are ubiquitous throughout the Lisp Machine system.

For this reason, in a future major release Zetalisp will be changed incompatibly to use a separate data type for characters, as Common Lisp does. In the remainder of this discussion the term *Old Zetalisp* will be used to refer to all Zetalisp releases before this incompatible change, which use integers to represent characters, and the term *New Zetalisp* will be used to refer to all Zetalisp releases following the switch to character objects. Old Zetalisp includes Release 6. This discussion presents facilities and techniques that enable you to write programs that will work in both Old Zetalisp and New Zetalisp, requiring only recompilation to convert between the two systems. These facilities and techniques resemble Common Lisp, but do not necessarily enable programs to work without change in Symbolics Common Lisp or any other Common Lisp implementation.

Some examples of places in the system that will be affected by the transition from Old Zetalisp to New Zetalisp include:

- Label strings in windows will be arrays of characters.
- Lines in Zwei will be arrays of characters.
- Print-names of symbols will be arrays of characters.
- The **:tyi** message to a stream will return a character if the stream is a character stream, or an integer if the stream is a binary stream.
- The **:tyo** message to a stream must be given a character if the stream is a character stream, or an integer if the stream is a binary stream.
- The **:line-in** message to a stream will return an array of characters rather than an array of integers.
- The **:string-out** message to a stream must be given a string (an array of characters) if the stream is a character stream, but must be given a one-dimensional array of integers if the stream is a binary stream.
- Some old Maclisp programs that depend on the representation of characters as integers will stop working; New Zetalisp is less compatible with Maclisp than Old Zetalisp.

New Zetalisp will never be supported on the LM-2.

Details of Character Objects

A character object is a structured object containing several fields. Accessor functions, described later, are provided to extract and modify the fields. In an abstract sense the fields of a character object are:

code	The actual character, such as "upper case A".
style	A modification of the character such as "italic" or "large".
bits	Control, Meta, Super, and Hyper.

In addition there are some derived fields, whose values depend on the values of the three fields listed above. For information about derived fields: See the section "Additional Character Object Enhancements", page 15. See the section "Device Fonts", page 16.

Common Lisp calls the *style* field the *font* field. Within Symbolics-Lisp, the word "font" is not used because it has misleading prior associations in Zetalisp.

The precise meaning of the *code* and *style* fields may not be clear. Characters that are recognizably distinct always have different character codes. For example, the Roman a and the Greek α have two different character codes. The character code, which specifies the fundamental identity of a character, is modified by a style specification and by modifier bits from the keyboard. A modification of a character that leaves it recognizably the same is expressed in the style field and does not change the character code. For example, the Roman a, the bold **a**, and the italic *a* all have the same character code. The style field also expresses such attributes of a character as its displayed size and the typeface used, for example, whether it has serifs.

An operational definition of the difference between the *code* and *style* fields is provided by the **char-equal** function, which compares character codes but ignores the style and the bits. **char-equal** also ignores distinctions of alphabetic case. Because user-visible character comparisons, such as the Search and Replace commands in the editor, compare characters with **char-equal** these commands ignore differences in character style. In Old-Zetalisp (that is, the previous releases of Zetalisp before this incompatible change, which use integers to represent characters) the set/style distinction is not fully implemented, therefore, a and α might be treated as the same character.

eq is not well defined on character objects. Changing a field of a character object gives you a "new copy" of the object; it never modifies somebody else's "copy" of "the same" character object. In this way character objects are just like integers with fields accessed by **ldb** and changed by **dpb**. Because **eq** is not well defined on character objects, they should be compared for identity with the **eql** function, not the **eq** function. This statement is true of integers as well. Integers can also be compared with =, but = is only for numbers and does not work for character objects. Currently on the 3600 family of machines, **eq** and **eql** are equivalent for characters,

just as they are equivalent for fixnums, but programs should not be written to depend on this, for two reasons:

- "Extended" character objects could be introduced in the future, standing in the same relationship to "basic" character objects as bignums do to fixnums.
- **eq** might not work for characters in other implementations of Common-Lisp-compatible Lisp dialects.

In Old-Zetalisp characters are represented with integers rather than character objects, but functions are provided to manipulate integers as if they were characters. In New-Zetalisp, the same functions will manipulate characters, providing compatibility.

Additional Character Object Enhancements

New-Zetalisp will contain additional enhancements beyond the reimplementation of the Old-Zetalisp concept of character with its own data type. (The term Old-Zetalisp refers to all Zetalisp releases before this incompatible change, which use integers to represent characters, and the term New-Zetalisp refers to all Zetalisp releases following the switch to character objects.)

The detailed design of these enhancements has not yet been finalized. The enhancements include:

- The **:character** data type name, usable with **typep**, **typecase**, and related functions
- The **characterp** predicate
- The new concept of character set
- Full implementation of character styles

The *code* field of a character can be broken down into a character set and an index into that character set. These are *derived fields* of a character.

A character set is a set of related characters that are recognizably different from other characters. Character sets have names and are represented inside the machine by objects that are instances of a character-set flavor. Examples of character sets are the standard Symbolics Lisp Machine character set including the Roman alphabet and other characters, Cyrillic (the Cyrillic alphabet), and Japanese (comprising a large set of Kanji characters plus two syllabaries or alphabets). Not all character sets need contain the same number of characters. The indices in the standard character set range from 0 to 255, whereas the indices in the Kanji character set range from 0 to about 8000.

The standard Lisp Machine character set is an upward-compatible extension of the

96 Common Lisp standard characters and the 6 Common Lisp semi-standard characters. It is almost an upward-compatible extension of ASCII; it uses a single Newline character and omits the ASCII control characters.

Character styles also have names and are represented inside the machine by objects (instances of a character-style flavor). Among our existing fonts, Times Roman, Centuryschoolbook, Jess, and CPTFONT are not different character sets; they are different styles of the Roman character set. Some styles can be applied to more than one character set; for example, most character sets can be made boldface. It is possible to mix styles together; for example, a character can simultaneously be bold, italic, and 24 points high.

Format-effector characters such as Return, Tab, and Space exist only in the standard character set, but can be modified by styles that make them geometrically compatible with other character sets.

When comparing characters, there is no intrinsic ordering between characters in different character sets. Two characters of different character sets are never equal. Less-than is not well defined between them. Within a single character set, less-than is defined so that characters (and strings) can be sorted alphabetically.

In Old-Zetalisp, the concepts of character set and character style are merged into a single concept: "font". Consequently there are no formal character-set and character-style objects in Old-Zetalisp, just informal "font numbers". Furthermore, the exact meaning of these numbers depends on whether the Japanese system is loaded. Effectively, there is only one character set in Old-Zetalisp, and the "font" number is a style. However, when the optional Japanese system is loaded there are two character sets — standard and Kanji — and the "font" number specifies both set and style. See the section "Support for Nonstandard Character Sets" in *Reference Guide to Symbolics-Lisp*. The Old-Zetalisp/New-Zetalisp compatibility facility does not contain any functions for dealing with character sets and styles. Programs that depend on this cannot be compatible between the two releases without source changes.

Several of the functions in the compatibility facility are based on Common Lisp functions that take an optional argument named "font". In Old-Zetalisp these functions do not take such an argument, since its meaning would be unclear and in any case it would change incompatibly in New-Zetalisp. In New-Zetalisp these functions will probably permit either a character set or a style, or both, to be specified by optional arguments.

Device Fonts

In Old-Zetalisp, there are two additional *derived fields* of a character: the *device-font* number and the *subindex*. These two fields are derived from the *code* and *style* fields. Together they describe how to portray the character on an output device. Note: Programs that do output are not normally concerned with these fields; only programs that implement output devices need to know about them. Device-fonts will

not exist in New-Zetalisp. At that time, any program that uses them will have to be changed.

The *device-font* number is an integer that selects a device-dependent font; the *subindex* then selects a particular character image from that font. There is potentially a different device-font for each combination of character set, style, and output device. Each output device (such as a window or an LGP) has a table that maps *device-font* numbers into actual device-fonts.

The *subindex* can be an integer between 0 and 255. A character set can contain any number of characters; most character sets contain 256 or fewer characters, but the Kanji character set contains about 8000. A *device-font* always contains 256 characters, thus a large character set requires several device-fonts to portray all of the characters in that character set.

char-device-font accesses the *device-font* field and **char-subindex** accesses the *subindex* field of the specified character.

Two Kinds of Characters in Old-zetalisp

A problem with Old-Zetalisp that you need to be aware of when writing code to deal with characters is that Old-Zetalisp has two incompatible kinds of characters. (The term Old-Zetalisp refers to all Zetalisp releases before this incompatible change, which use integers to represent characters, and the term New-Zetalisp refers to all Zetalisp releases following the switch to character objects.)

One kind, associated with the **%%kbd-** byte specifiers, is used for characters from the keyboard. The other kind, associated with the **%%ch-** byte specifiers, is used for characters in files, editor buffers, and strings. The keyboard characters may contain modifier bits, such as Control and Meta; the file characters may contain a device-font. The same bits in the number are used for both purposes, so in Old-Zetalisp you cannot have a character with both bits and a font, and furthermore each character-processing function assumes that it was given a particular type of character as its argument; it has no way to tell which kind of character the caller intended, since all characters are just represented as numbers. Most functions assume file characters. Some functions work on either kind of character, as long as all arguments are of the same kind, because they treat the bits and device-font attributes identically. If you are guaranteed to be dealing with characters in the common intersection of the two kinds, that is, characters whose bits and device-font attributes are both zero, you do not need to be concerned with these issues.

An example of the way you can get into trouble now is that **(alpha-char-p #\c-A)** returns **t**. Common Lisp specifies that it is supposed to return **nil**. In Old-Zetalisp, **alpha-char-p** expects a file character, so it regards the "control" bit as being a font number and ignores it. All problems of this type will be fixed by New-Zetalisp, but in the meantime you need to be aware of them. The function descriptions in another section say which type of character each function operates on. See the section "Character and String Functions for Old-zetalisp/New-zetalisp Compatibility", page 24.

2.2.5.2 Old-zetalisp and New-zetalisp String and Character Compatibility

This section provides compatibility information for strings and characters in Old-Zetalisp and New-Zetalisp.

Summary of Character and String Compatibility Functions in Release 6.0

The following functions are provided to make it possible to write compatible code that works in both Old-Zetalisp and New-Zetalisp. (The term Old-Zetalisp refers to all Zetalisp releases before this incompatible change, which use integers to represent characters, and the term New-Zetalisp refers to all Zetalisp releases following the switch to character objects.) Some of these functions have existed in Zetalisp for a long time, while others have been newly introduced to aid in the conversion. The names of these functions were chosen to be compatible with Common Lisp, but these functions are not identical to the Common Lisp functions with similar names.

This section simply summarizes the functions. For more information about these functions: See the section "Character and String Functions for Old-zetalisp/New-zetalisp Compatibility", page 24.

Accessing and modifying fields of characters

char-bits	char-bit	set-char-bit
char-code	code-char	make-char
char-device-font	char-subindex	

Character names

char-name	name-char

Predicates on characters

char-standard	graphic-char-p	
alpha-char-p	digit-char-p	alphanumericp
upper-case-p	lower-case-p	both-case-p

Character Conversions

character	char-int	int-char
char-downcase	char-upcase	char-flipcase

Digits of Numbers

digit-char-p	digit-char

Mouse characters

mouse-char-p	char-mouse-button
char-mouse-n-clicks	make-mouse-char

ASCII characters

ascii-code	char-to-ascii	ascii-to-char

Comparison of characters affected by case, style, and bits

```
char=                  char≠                  char<
char>                  char≤                  char≥
```

Comparison of characters ignoring case, style, and bits

```
char-equal             char-not-equal         char-lessp
char-greaterp          char-not-greaterp      char-not-lessp
```

Comparison of strings affected by case, style, and bits

```
string=                string≠                string<
string>                string≤                string≥
%string=               string-exact-compare
sys:%string-exact-compare
```

Comparison of strings ignoring case, style, and bits

```
string-equal           string-not-equal       string-lessp
string-greaterp        string-not-greaterp    string-not-lessp
%string-equal          string-compare         sys:%string-compare
```

String searching affected by case, style, and bits

```
string-search-exact-char           string-search-not-exact-char
string-reverse-search-exact-char   %string-search-exact-char
string-search-exact                string-reverse-search-exact
string-reverse-search-not-exact-char
```

String searching ignoring case, style, and bits

```
string-search-char                 string-search-not-char
string-reverse-search-char         %string-search-char
string-search                      string-reverse-search
string-reverse-search-not-char
```

Obsolete Programming Practices Using Characters and Strings

This section documents programming practices that have been used in Zetalisp in the past and their compatible replacements. The replacements work with integers in Old-Zetalisp and with character objects in New-Zetalisp. The replacements are as efficient as the present practices, in both releases, in compiled code.

Use of Integers in Source Code Where Characters Are Desired

Replace 101 with **#/A**.

Use of Characters in Source Code Where Integers Are Desired

Replace **#/A** with **#.(char-code #/A)**.

The **#.** is necessary only in contexts where the **#/A** was data rather than a form; **char-code** will be constant-folded by the compiler when its argument is constant.

You could also use **char-int**, but it never makes sense to get an integer that

represents more than one field of a character, except when computing hash keys. The positions of the various fields within the word should never be meaningful to the outside world.

Use of Symbolics Lisp Machine Characters in Source Code Where ASCII Characters Are Desired

Replace **#/A** with **#.(ascii-code #/A)**.

Unfortunately this function cannot be called **ascii**, because that name is already taken for a Maclisp-compatible function that returns a symbol. **ascii-code** returns an integer, for example **(ascii-code #\cr) => #o15**. The **ascii-code** function also recognizes strings and looks up the names of the ASCII "control" characters. Thus **(ascii-code "SOH") = (ascii-code #\↓) = 1**. **(ascii-code #\c-A) = #o101**, not 1; there is no mapping between Lisp Machine control characters and ASCII control characters.

The functions **char-to-ascii** and **ascii-to-char** provide the primitive conversions needed by ASCII-translating streams. They do not deal with the translation of the Return character into a CR-LF pair; the caller must handle that. They just translate **#\Return** into CR and **#\Line** into LF. They do not deal with Symbolics Lisp Machine control characters; the translation of **#\c-G** is the ASCII code for G, not the ASCII code to ring the bell also known as "control G." **(ascii-to-char (ascii-code "BEL"))** is **#/π**, not **#\c-G**. The translation from ASCII to character never produces a Lisp Machine control character; this is necessary so that these functions can be used to translate file data (as opposed to keyboard data). Except for CR-LF, **char-to-ascii** and **ascii-to-char** are 100 percent compatible with the ASCII-translating streams.

Use of Numerical Comparisons on Characters

Use of numerical comparisons on characters is the major incompatibility. Rather than using = to compare characters, you should use one of the following specialized predicates. If none of them does what you need, use **char-code** to extract the field you want to compare, then compare it arithmetically; this should be rare.

char=	char≠	char<
char>	char≤	char≥
char-equal	char-not-equal	char-lessp
char-greaterp	char-not-greaterp	char-not-lessp
graphic-char-p	alpha-char-p	upper-case-p
lower-case-p	digit-char-p	alphanumericp

The predicates listed on the first two lines (**char=**, **char≠**, **char<**, **char>**, **char≤**, and **char≥**) are "exact". The predicates on the third and fourth lines (**char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp**) ignore bits, style, and alphabetic case. The ones that already existed in Zetalisp in Release 5.0 are compatible.

You can also use **eql** to compare characters. **char=** and **eql** are equivalent except for possible error-checking; **char=** might complain about arguments that are not characters. You are allowed to use **selectq** on characters; currently it uses **eq** to compare the characters, which works but is discouraged as poor practice. By the time **eql** needs to be used **selectq** will be fixed to use it.

You can also use **selector** with **char-equal** to select on a character, ignoring bits, style, and case.

Example:

```
(selector char char-equal
        (#/a ...))
```

Use of Arithmetic Operations on Characters

Use **char-code** and **code-char** to convert between characters and integers. There are also the specialized functions:

```
digit-char-p      char-code         code-char
char-bits         character         char-upcase
char-downcase     char-flipcase     digit-char
char-bit          set-char-bit      char-int
int-char
```

Use of ldb, ldb-test, logand, and bit-test on Characters

For characters, instead of performing logical operations — **ldb, ldb-test, logand, bit-test**, and others — on byte fields, you should now use the functions listed below. If you use these functions, your code will work in both Old-Zetalisp and New-Zetalisp. To extract the byte field you want to operate on, you should use one of these specialized functions and predicates:

```
alphanumericp     char-downcase     digit-char-p
alpha-char-p      char-flipcase     graphic-char-p
both-case-p       char-int          int-char
character         char-name         lower-case-p
char-bit          char-subindex     name-char
char-bits         char-upcase       set-char-bit
char-code         code-char         upper-case-p
char-device-font  digit-char
```

For example, you should use **(char-bit char :meta)** rather than **(ldb-test %%kbd-meta char)**; likewise, you should use **(setf (char-bit char :meta) t)**, instead of **(setq char (dpb 1 %%kbd-meta char))**.

The functions that already existed in Zetalisp in Release 5.0 (**character, char-downcase, char-flipcase,** and **char-upcase**) are compatible.

The following variables will not exist in New-Zetalisp; therefore, you should avoid using them.

```
%%ch-char              %%ch-font           %%kbd-char
%%kbd-control          %%kbd-control-meta  %%kbd-hyper
%%kbd-meta             %%kbd-mouse         %%kbd-mouse-button
%%kbd-mouse-n-clicks   %%kbd-super
```

Use of Mouse Characters

The syntax for mouse characters, such as **#\mouse-l-1**, will continue to work in New-Zetalisp.

Old coding practice	*New coding practice*
`(ldb-test %%kbd-mouse` *char*`)`	`(mouse-char-p` *char*`)`
`(ldb %%kbd-mouse-button` *char*`)`	`(char-mouse-button` *char*`)`
`(ldb %%kbd-mouse-n-clicks` *char*`)`	`(char-mouse-n-clicks` *char*`)`

You can use **setf** on these fields; there is also a function **(make-mouse-char** *button n-clicks* &optional *bits*).

It is important to note that *n-clicks* is 0 if the button was clicked once, 1 if the button was clicked twice. The button number is 0, 1, or 2 (Left, Middle, Right).

Use of Characters as Array Subscripts

Call **char-code** first. Note that this may give a large number in New-Zetalisp, or if the Japanese system is in use in Old-Zetalisp (because of character sets). Characters in the standard character set have codes less than 256, so things should work compatibly if only the standard character set is used. In New-Zetalisp, programs that use characters to index into command dispatch tables will have to be careful about nonstandard character sets causing out-of-bounds array references. Most programs with single-character commands, such as the editor, treat all characters in non-standard character sets as self-inserting.

Distinguishing Characters From Blips

There is no **characterp** function in Old-Zetalisp, because the chance for confusion between characters and numbers is too great. Rather than using **numberp** to test for characters, as many system and example programs do, use **listp** to test for blips, and assume any input that is not a list must be a character. For now, at least, all blips in system programs are lists. It is possible that instances might sometimes be used as blips in some future system.

Also, anything that wants to check the data type of a character argument cannot be source-compatible. In Old-Zetalisp you use **(check-arg-type** *char* **:fixnum)** and in New-Zetalisp you use **(check-arg-type** *char* **:character)**. In Common Lisp you would use **(check-type** *char* **character)**.

String Comparison and String Searching Functions

The complete set of case-independent string comparison functions is:

```
string-equal          string-lessp       string-greaterp
string-not-greaterp   string-not-lessp   string-not-equal
string-compare        %string-equal      sys:%string-compare
```

The complete set of case-dependent string comparison functions is:

```
string=               string<            string>
string≤               string≥            string≠
string-exact-compare  %string=           sys:%string-exact-compare
```

For fat strings, the dependency on character style is the same as the dependency on alphabetic case.

For string searching, the new set of case-dependent search functions is as follows. Note that none of these exist in Common Lisp, which does string searching with sequence operations instead.

```
string-search-exact                string-reverse-search-exact
string-search-exact-char           string-search-not-exact-char
string-reverse-search-exact-char   %string-search-exact-char
string-reverse-search-not-exact-char
```

Case-dependent versions of the **string-search-set** and **string-trim** families are not provided, because the set normally does not contain alphabetics. Common Lisp handles this better, using the **:test** argument to its sequence functions.

Fat Strings

It is impossible to provide complete compatibility here, because of the impending introduction of character sets and styles. Any program that uses fat strings in Old-Zetalisp will need some source changes to work in New-Zetalisp. Conversion instructions will be provided in the future.

Making Strings

art-string arrays will continue to be strings. In Old-Zetalisp they are arrays of 8-bit bytes; in New-Zetalisp they are arrays of characters restricted to have zero bits and style fields, and a code field in the Standard character set.

Common Lisp Functions Not Included

standard-char-p is not included because the Common Lisp standard characters are not directly relevant to Zetalisp.

string-char-p is not included because it is part of the Common Lisp type system.

char-font is not included because it is superseded in New-Zetalisp by **char-style**. The New-Zetalisp/Symbolics Common Lisp functions **char-set**, **char-style**, and

char-index are not included because character sets and styles are not implemented in Old-Zetalisp.

2.2.5.3 Character and String Functions for Old-zetalisp/New-zetalisp Compatibility

The functions listed below have been documented in earlier releases. These functions are compatible with both Old-Zetalisp and New-Zetalisp.

```
char-code                       string-search-char
char-standard                   string-search-not-char
char-equal                      string-reverse-search-char
char-lessp                      string-reverse-search-not-char
char-upcase                     string-search
char-downcase                   string-reverse-search
char-flipcase                   %string-search-char
string-equal                    string-search-set
string-lessp                    string-search-not-set
string-compare                  string-reverse-search-set
%string-equal                   string-reverse-search-not-set
alphabetic-case-affects-string-comparison
```

2.2.6 Symbols Added to or Removed From global in Release 6.0

The following symbols have been added to the **global** package in Release 6.0:

%32-bit-difference
%32-bit-plus
%find-structure-extent
%string-search-exact-char
%string=
&environment
&whole
read-form-completion-alist
read-form-completion-delimiters
read-form-edit-trivial-errors-p
alpha-char-p
alphanumericp
array-column-major-index
array-push-portion-extend
art-boolean
art-fixnum
ascii-code
ascii-to-char
ascii-to-string
both-case-p
ceiling
change-instance-flavor
char≠
char≤

char≥
char-bit
char-bits
char-device-font
char-greaterp
char-int
char-mouse-button
char-mouse-n-clicks
char-name
char-not-equal
char-not-greaterp
char-not-lessp
char-subindex
char-to-ascii
char<
char=
char>
choose-gc-parameters
cis
code-char
complex
complexp
conjugate
copytree-share
cosh
cp-off
cp-on
define-cp-command
defvar-resettable
defvar-standard
defwhopper-subst
denominator
desetq
digit-char
digit-char-p
display-notifications
dlet
dlet*
flet
floor
get-flavor-handler-for
graphic-char-p
imagpart
int-char
labels
letf

letf*
location-contents
lower-case-p
macrolet
make-char
make-heap
make-mouse-char
mouse-char-p
name-char
note-private-patch
numerator
parse-ferror
phase
process-wait-forever
push-in-area
rational
rationalp
read-and-eval
read-command
read-command-or-form
read-expression
read-for-eval
read-form
read-interactive
read-or-character
readline-no-echo
realpart
round
set-char-bit
setq-standard-value
sinh
stack-let
stack-let*
standard-value-let
standard-value-let*
standard-value-progv
string≠
string≤
string≥
string-exact-compare
string-greaterp
string-nconc-portion
string-not-equal
string-not-greaterp
string-not-lessp
string-reverse-search-exact

string-reverse-search-exact-char
string-reverse-search-not-exact-char
string-search-exact
string-search-exact-char
string-search-not-exact-char
string-to-ascii
string<
string=
string>
tand
tanh
time-elapsed-p
truncate
unwind-protect-case
upper-case-p
with-input-editing-options
with-input-editing-options-if
with-notification-mode
without-floating-underflow-traps

The following symbols have been removed from the **global** package in Release 6.0:

&dt-atom
&dt-dontcare
&dt-fixnum
&dt-frame
&dt-list
&dt-number
&dt-symbol
&function-cell
functional-alist
zdt
zed

2.2.7 apply and funcall No Longer Work for Special Forms

You can no longer **apply** or **funcall** a special form or a macro. In Release 5, you could **apply** or **funcall** a special form, but the results were unpredictable. In Release 6, doing this signals an error.

2.2.8 Interpreter Caches Global Variable Declarations

The interpreter caches lexical, dynamic, and special information like the compiler does. If you change the meaning of a variable (for example, declare it special), you must reinterpret and recompile the **defun** form.

2.2.9 Files Using defwrapper Forms Must Be Recompiled in Release 6.0

Compiled files that contain **defwrapper** forms do not work in both Release 5 and Release 6. You should therefore snapshot (using **make-system** tools, if necessary) your existing Release 5 version and recompile a separate Release 6 version.

2.2.10 :fixnum-array Option for defstruct is Obsolete

The **:fixnum-array** option for **defstruct** is not supported on the 3600-family machines.

2.2.11 :flonum-array Option for defstruct is Obsolete

The **:flonum-array** option for **defstruct** is not supported on the 3600-family machines. The option is not needed on the 3600, as its purpose was to enhance efficiency on the Symbolics LM-2 machine.

2.2.12 make-array No Longer Accepts Obsolete Form

When **make-array** was originally implemented, it took its arguments in the following fixed pattern:

```
(make-array area type dimensions
            &optional displaced-to leader
                      displaced-index-offset
                      named-structure-symbol)
```

leader was a combination of the **:leader-length** and **:leader-list** options, and the list was in reverse order.

This form of **make-array** is obsolete and no longer supported.

2.2.13 Forms in a Top-level progn Are Top-level to the Compiler

Forms within a top-level **progn** are treated by the compiler as if they had appeared at top-level, regardless of whether or not **'compile** is specified.

If your code depends on forms not being seen by the compiler, hide the forms by wrapping **(eval (quote...))** around them.

```
(eval (quote
  (progn
     (forms)
     ...)))
```

2.2.14 Lambda-list Keyword Changes

&functional has never worked on the 3600-family machines.

The use of **&quote** and **&eval** is not recommended. Macros should be used instead to define special functions.

&list-of has been removed from Symbolics-Lisp. Use **loop** or **mapcar** instead of **&list-of**.

Example 1, using **&list-of**:

```
(defmacro send-commands (object
                         &body &list-of (command . arguments))
  `(let ((o ,object))
     . ,(mapcar #'(lambda (com args) `(send o ',com . ,args))
                command arguments)))
```

Using **mapcar**:

```
(defmacro send-commands (object &body command.arguments)
  (let ((command (mapcar #'car command.arguments))
        (arguments (mapcar #'cdr command.arguments)))   ;simulate &list-of
    `(let ((o ,object))
       . ,(mapcar #'(lambda (com args) `(send o ',com . ,args))
                  command arguments))))
```

Using **loop**:

```
(defmacro send-commands (object &body command.arguments)
  `(let ((o ,object))
     ,@(loop for (command . arguments) in command.arguments
             collect `(send o ',command ,@arguments))))
```

Example 2, using **&list-of**:

```
(defmacro print-let (x &optional &list-of ((vars vals)
                                           '((base 10.)
                                             (*nopoint t))))
  `((lambda (,@vars) (print ,x))
    ,@vals))
```

Using **mapcar**:

```
(defmacro print-let (x &optional (let-vars '((base 10.)
                                             (*nopoint t))))
  `((lambda (,@(mapcar #'car let-vars))
      (print ,x))
    ,@(mapcar 'cadr let-vars)))
```

Using **let**:

```
(defmacro print-let (x &optional (let-vars '((base 10.)
                                             (*nopoint t))))
  `(let ,let-vars
     (print ,x)))
```

See the section "New &-Keywords for **defmacro**", page 39.

2.2.15 defmacro Patterns Are Now Made Consistent

defmacro now destructures all levels of patterns in a consistent way. In the past, &-keywords were allowed only at the top level of **defmacro**'s argument pattern. &-keywords were not allowed inside nested lists, and all arguments in nested lists were effectively optional. Also, error checking was not done on the matching of lengths of the pattern and the subform. See the section "New &-Keywords for **defmacro**", page 39.

All levels of the argument pattern now behave uniformly. As a result, **&optional**, for example, needs to be inserted into some macros to ensure that they work the same way that they used to work.

```
(defmacro foo (x y z) ...)
```
Example 1 has not changed.

```
(defmacro foo (x y &optional z) ...)
```
Example 2 has not changed.

```
(defmacro foo ((x y z) &body w) ...)
```
Example 3 is now written as Example 4 if z was supposed to be optional.

```
(defmacro foo ((x y &optional z) &body w) ...)
```

The following is a more complicated example.

```
(defmacro hairy ((&whole first-form w &key x y z &allow-other-keys)
                 (&optional a (b 'c) &aux (a-and-b (and a b)))
                 &body body)
  ;; print things during macro expansion
  (format t "~&First form is ~S~%W = ~S, X = ~S, Y = ~S, Z = ~S~%"
    first-form w x y z)
  (format t "A = ~S, B = ~S, A-and-B = ~S~%" a b a-and-b)
  (format t "BODY = ~S~%" body)
  ;; and expand into nil
  nil)
```

When

```
(hairy (this-is-w :x this-is-x :z this-is-z :y this-is-y
                  :something-else ignored)
       ()
       body-form-1
       body-form-2)
```

is expanded, it prints

```
First form is (THIS-IS-W :X THIS-IS-X :Z THIS-IS-Z  :Y THIS-IS-Y
:SOMETHING-ELSE IGNORED)
W = THIS-IS-W, X = THIS-IS-X, Y = THIS-IS-Y, Z = THIS-IS-Z
A = NIL, B = C, A-and-B = NIL
BODY = (BODY-FORM-1 BODY-FORM-2)
```

during expansion and expands into

```
NIL
```

When

```
(hairy (this-is-w :z this-is-z)
       (this-is-a explicit-b)
  body-form-1
  body-form-2)
```

is expanded, it prints

```
First form is (THIS-IS-W :Z THIS-IS-Z)
W = THIS-IS-W, X = NIL, Y = NIL, Z = THIS-IS-Z
A = THIS-IS-A, B = EXPLICIT-B, A-and-B = EXPLICIT-B
BODY = (BODY-FORM-1 BODY-FORM-2)
```

during expansion and expands into

```
NIL
```

This behavior exists for all of **defmacro**'s keywords, except for **&environment**.

2.2.16 alphabetic-case-affects-string-comparison is Now Obsolete

The variable **alphabetic-case-affects-string-comparison** is now obsolete. See the section "Common Lisp Character Switchover in Release 6.0", page 12.

2.3 New Features in Lisp in Release 6.0

2.3.1 New Function: change-instance-flavor

change-instance-flavor changes the flavor of an instance to another flavor that has compatible instance variables.

See the function **change-instance-flavor** in *Reference Guide to Symbolics-Lisp*.

2.3.2 New Option to defflavor: :export-instance-variables

The **:export-instance-variables** option has been added to **defflavor**.

:export-instance-variables exports the symbols from the package in which the flavor is defined. The following example shows the use of **:export-instance-variables**.

```
(defflavor box
        (x-dim y-dim z-dim)
        ()
  :gettable-instance-variables
  ;; export all the instance variables
  :export-instance-variables)
```

See the section "Importing and Exporting Symbols" in *Reference Guide to Symbolics-Lisp*.

2.3.3 New Macro: defwhopper-subst

The macro **defwhopper-subst** has been added. **defwhopper-subst** defines a wrapper for the specified message to the specified flavor by combining the use of **defwhopper** with the efficiency of **defwrapper**. The body is expanded in-line in the combined method, providing improved time efficiency but decreased space efficiency unless the body is small.

See the macro **defwhopper-subst** in *Reference Guide to Symbolics-Lisp*.

2.3.4 New Macro: si:define-simple-method-combination

The macro **si:define-simple-method-combination** provides a simple means of defining a method combination with the name *combination-type*, which must be a symbol and is usually a keyword, such as **:progn** or **:list**.

See the macro **si:define-simple-method-combination** in *Reference Guide to Symbolics-Lisp*.

2.3.5 New Option to defstruct: :export

The **:export** option has been added to **defstruct**.

The **:export** option exports the specified symbols from the package in which the structure is defined. This option accepts the following as arguments: the names of slots and the following options: **:alterant**, **:constructor**, **:copier**, **:predicate**, **:size-macro**, and **:size-symbol**.

The following example shows the use of **:export**.

```
(defstruct (2d-moving-object
             (:type :array)
             :conc-name
             ;; export all accessors and make-2d-moving-object
             (:export :accessors :constructor))
  mass
  x-pos
  y-pos
  x-velocity
  y-velocity)
```

See the section "Importing and Exporting Symbols" in *Reference Guide to Symbolics-Lisp*.

2.3.6 New Special Form: without-floating-underflow-traps

without-floating-underflow-traps replaces the variable **zunderflow**, which was the only way to turn off underflow traps previously. **zunderflow** worked on the LM-2, but is not quite correct for the 3600 family. You should use **without-floating-underflow-traps** on the 3600 family because it is more mathematically correct and (when there is an underflow) it is faster.

See the special form **without-floating-underflow-traps** in *Reference Guide to Symbolics-Lisp*.

2.3.7 New Function: tand

tand *x* *Function*

Returns the tangent of *x*, where *x* is expressed in degrees.

For example:

```
(tand 45) => 1.0
(tand -45.0) => -1.0
(tand 180.0d0) => 0.0d0
```

2.3.8 New Functions: %32-bit-plus and %32-bit-difference

These two functions are the 32-bit versions of the **%24-bit-** functions that existed on the LM-2.

%32-bit-plus *x y* *Function*

Returns the sum of *x* and *y* in 32-bit wraparound arithmetic. Both arguments must be fixnums. The result is a fixnum.

%32-bit-difference *x y* *Function*

Returns the difference of *x* and *y* in 32-bit wraparound arithmetic. Both arguments must be fixnums. The result is a fixnum.

Example:

```
(+ si:*largest-fixnum* 1) => 20000000000  ;;a bignum
(%32-bit-plus si:*largest-fixnum* 1) => -20000000000  ;;a fixnum
```

2.3.9 New Keywords to typep

There are four new keywords to **typep**:

:complex `(typep #c(1.3 7.0) :complex) => t`

:rational `(typep 5\3 :rational) => t`

:non-complex-number
`(typep 4 :non-complex-number) => t`

:list-or-nil `(typep '(a b c) :list-or-nil) => t`

2.3.10 Rational and Complex Numbers

2.3.10.1 Rational Numbers

Rational numbers include both ratios and integers. Ratios are represented in terms of an integer numerator and denominator. The ratio is always "in lowest terms", meaning that the denominator is as small as possible. If the denominator is 1, the rational number is represented as an integer. The denominator is always positive; the sign of the number is carried by the numerator. See the section "Numeric Type Conversions" in *Reference Guide to Symbolics-Lisp*.

rational *x* *Function*

Converts any noncomplex number to an equivalent rational number. If *x* is a floating-point number, **rational** returns the rational number of least denominator, which when converted back to the same floating-point precision, is equal to *x*.

numerator *x* *Function*

If *x* is a ratio, **numerator** returns the numerator of *x*. If *x* is an integer, **numerator** returns *x*.

denominator *x* *Function*

If *x* is a ratio, **denominator** returns the denominator of *x*. If *x* is an integer, **denominator** returns 1.

rationalp *x* *Function*

Returns **t** if *x* is a ratio. Returns **nil** if *x* is an integer. Note that in Common Lisp, **rationalp** of an integer returns **t**.

2.3.10.2 Complex Numbers

A complex number is a pair of noncomplex numbers, representing the real and imaginary parts of the number. The types of the real and imaginary parts are always the same. No Symbolics-Lisp complex number has a rational real part and an imaginary part of integer zero. Such a number is always represented simply by the rational real part. See the section "Numeric Type Conversions" in *Reference Guide to Symbolics-Lisp*.

complex *real* &optional *imag* *Function*

Constructs a complex number from real and imaginary noncomplex parts. If the types of the real and imaginary parts are different, the coercion rules are applied to make them the same. If *imag* is not specified, a zero of the same type as *real* is used. If *real* is an integer or a ratio, and *imag* is 0, the result is *real*.

realpart *x* *Function*

If *x* is a complex number, **realpart** returns the real part of *x*. If *x* is a noncomplex number, **realpart** returns *x*.

imagpart *x* *Function*

If *x* is a complex number, **imagpart** returns the imaginary part of *x*. If *x* is a noncomplex number, **imagpart** returns a zero of the same type as *x*.

complexp *x* *Function*

Returns **t** if *x* is a complex number, otherwise **nil**.

2.3.11 New Transcendental Functions

The following new transcendental functions have added in Release 6.0.

cis *x* *Function*

x must be a noncomplex number. **cis** could have been defined by:

```
(defun cis (x)
  (complex (cos x) (sin x)))
```

Mathematically, this is equivalent to e^{ix}.

phase *x* *Function*

The phase of a number is the angle part of its polar representation as a complex number. The phase of zero is arbitrarily defined to be zero. **phase** could have been defined as:

```
(defun phase (x)
  (atan2 (imagpart x) (realpart x)))
```

sinh *x* *Function*

Returns the hyperbolic sine of *x*, where *x* is expressed in radians.

cosh *x* *Function*

Returns the hyperbolic cosine of *x*, where *x* is expressed in radians.

tanh *x* *Function*

Returns the hyperbolic tangent of *x*, where *x* is expressed in radians.

2.3.12 New Function: conjugate

conjugate *x* *Function*

Returns the complex conjugate of *x*. The conjugate of a noncomplex number is itself. **conjugate** could have been defined by:

```
(defun conjugate (x)
  (complex (realpart x) (- (imagpart x))))
```

2.3.13 New Functions for Converting Non-integral Numbers to Integers

Four new functions for converting non-integral numbers to integers have been added to Symbolics-Lisp. These functions are specified by Common Lisp, but are also added to Symbolics-Lisp so that programs written in Symbolics-Lisp can use them. For information about these new functions:

See the function **floor** in *Reference Guide to Symbolics-Lisp.*
See the function **ceiling** in *Reference Guide to Symbolics-Lisp.*
See the function **round** in *Reference Guide to Symbolics-Lisp.*
See the function **truncate** in *Reference Guide to Symbolics-Lisp.*

2.3.14 New Functions for Converting Numbers to Floating-point Numbers

Four new functions for converting numbers to floating-point numbers have been added to Symbolics-Lisp. These functions are specified by Common Lisp, but are also added to Symbolics-Lisp so that programs written in Symbolics-Lisp can use them. For information about these new functions:

See the function **sys:ffloor** in *Reference Guide to Symbolics-Lisp.*
See the function **sys:fceiling** in *Reference Guide to Symbolics-Lisp.*
See the function **sys:fround** in *Reference Guide to Symbolics-Lisp.*
See the function **sys:ftruncate** in *Reference Guide to Symbolics-Lisp.*

2.3.15 New Function: location-contents

location-contents replaces the use of **car** and **cdr** on locatives. Similarly, although either of the functions **rplaca** and **rplacd** can be used to store an object into the cell at which a locative points, you should use **(setf (location-contents x) y)** instead.

See the function **location-contents** in *Reference Guide to Symbolics-Lisp.*

2.3.16 New Facility: Heaps

Heaps have been implemented in Release 6.0. A heap is a data structure in which each item is ordered by some predicate (for example, less-than) on its associated key. You can add an item to the heap, delete an item from it, or look at the top item. The "top" operation is guaranteed to return the first (that is, smallest) item in the heap. Heaps are useful in maintaining priority queues.

For additional information about heaps:

See the function **make-heap** in *Reference Guide to Symbolics-Lisp.*
See the section "Messages to Heaps" in *Reference Guide to Symbolics-Lisp.*
See the section "Heaps and Loop Iteration" in *Reference Guide to Symbolics-Lisp.*

2.3.17 Previously Undocumented Feature: Array Registers

Array registers are now documented. The array register feature makes optimization possible and convenient. Array registers are documented in the following topics:

See the section "Array Registers" in *Reference Guide to Symbolics-Lisp.*
See the section "Accessing Multidimensional Arrays as One-dimensional" in *Reference Guide to Symbolics-Lisp.*

2.3.18 New Function: array-column-major-index

The function **array-column-major-index** takes an array and valid subscripts for the array and returns a single non-negative integer less than the total size of the array that identifies the accessed element in the column-major ordering of the elements.

See the function **array-column-major-index** in *Reference Guide to Symbolics-Lisp.*

2.3.19 New Special Forms: letf and letf*

Two new special forms have been added in Release 6.0: **letf** and **letf***. **letf** is just like **let**, except that it can bind any storage cells rather than just variables. **letf*** is just like **let***, except that it can bind any storage cells rather than just variables. For more information:

See the special form **letf** in *Reference Guide to Symbolics-Lisp.*
See the special form **letf*** in *Reference Guide to Symbolics-Lisp.*

2.3.20 New Function: copytree-share

copytree-share is similar to **copytree**, except that it also assures that all lists or tails of lists are optimally shared when **equal**.

See the function **copytree-share** in *Reference Guide to Symbolics-Lisp.*

2.3.21 New Macro: unwind-protect-case

The macro **unwind-protect-case** has been added.

See the macro **unwind-protect-case** in *Reference Guide to Symbolics-Lisp.*

2.3.22 New Function: array-push-portion-extend

The function **array-push-portion-extend** copies a portion of one array to the end of another, updating the fill pointer of the other to reflect the new contents.

See the function **array-push-portion-extend** in *Reference Guide to Symbolics-Lisp.*

2.3.23 New Function: string-nconc-portion

The function **string-nconc-portion** adds information onto a string without consing intermediate substrings. It is like **string-nconc** except that it takes parts of strings without consing substrings.

See the function **string-nconc-portion** in *Reference Guide to Symbolics-Lisp*.

2.3.24 New Flavor: sys:float-invalid-compare-operation

sys:float-invalid-compare-operation is built on and identical to **sys:float-invalid-operation**, except that it does not expect a numeric result.

See the flavor **sys:float-invalid-compare-operation** in *Reference Guide to Symbolics-Lisp*.

2.3.25 New Error Flavor: sys:read-premature-end-of-symbol

sys:read-premature-end-of-symbol *Flavor*

This is a new error flavor based on **sys:read-error**. It can be used for signalling when some read function finishes reading in the middle of a string that was supposed to contain a single expression.

Message	*Value returned*
:short-symbol	the symbol that was read
:original-string	the string that it was reading from when it finished in the middle

An example of the use of **sys:read-premature-end-of-symbol** is in **zwei:symbol-from-string**.

2.3.26 New Array Error Flavor: sys:array-wrong-number-of-subscripts

sys:array-wrong-number-of-subscripts assumes that the array is correct and that the user/application caused the error by providing the incorrect number of subscripts.

See the flavor **sys:array-wrong-number-of-subscripts** in *Reference Guide to Symbolics-Lisp*.

2.3.27 New Stream Handling Error Flavors: sys:stream-closed and sys:network-stream-closed

sys:stream-closed is used when an operation that required a stream to be open was attempted on a closed stream. See the flavor **sys:stream-closed** in *Reference Guide to Symbolics-Lisp*.

sys:network-stream-closed is a combination of **sys:network-error** and **sys:stream-closed** and is usually used as a base flavor by network implementations (for example, Chaos and TCP). See the flavor **sys:network-stream-closed** in *Reference Guide to Symbolics-Lisp*.

2.3.28 New Message to Error Flavor fs:directory-not-found

Errors of flavor **fs:directory-not-found** support the **:directory-pathname** message. This message, which can be sent to any such error, returns (when possible) a "pathname as directory" for the actual directory which was not found.

Example:

Assume the directory x:>a>b exists, but has no inferiors. The following produces an error instance to which **:pathname** produces #<LMFS-PATHNAME x:>a>b>c>d>thing.lisp> and **:directory-pathname** produces #<LMFS-PATHNAME x:>a>b>c> >.

```
(open "x:>a>b>c>d>thing.lisp")
```

Note: Not all hosts and access media can provide this information, although LMFS can. When a host does not return this information, **:directory-pathname** returns the same as **:pathname**, whose value is a pathname as directory for the best approximation known to the identity of the missing directory.

2.3.29 New &-Keywords for defmacro

defmacro has two new &-keywords: **&whole** and **&environment**.

&whole **&whole** is followed by *variable*, which is bound to the entire macro-call form or subform. *variable* is the value that the macro-expander function receives as its first argument. **&whole** is allowed only in the top-level pattern, not in inside patterns.

&environment **&environment** is followed by *variable*, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment. It should be used only with the **macroexpand** function for any local macro definitions that the **macrolet** construct might have established within that lexical environment. **&environment** is allowed only in the top-level pattern, not in inside patterns. See the section "Lexical Environment Objects and Arguments" in *Reference Guide to Symbolics-Lisp*.

defmacro now accepts these lambda-list keywords, formerly accepted only by **defun**: **&key** and **&allow-other-keys**.

&key Separates the positional arguments and rest argument from the keyword arguments.

&allow-other-keys
In a lambda-list that accepts keyword arguments, says that keywords that are not specifically listed after **&key** are allowed. They and the corresponding values are ignored, as far as keyword

arguments are concerned, but they do become part of the rest argument, if there is one.

See the section "Lambda-list Keyword Changes", page 28. See the section "**defmacro** Patterns Are Now Made Consistent", page 30.

2.3.30 New Special Forms for Destructuring

Three new special forms for destructuring have been added: **desetq**, **dlet**, and **dlet***.

desetq lets you assign values to variables through destructuring patterns. **dlet** binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings. **dlet*** binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings.

For more information about these special forms:

See the special form **desetq** in *Reference Guide to Symbolics-Lisp*.
See the special form **dlet** in *Reference Guide to Symbolics-Lisp*.
See the special form **dlet*** in *Reference Guide to Symbolics-Lisp*.

For more information about the concept of destructuring in general:

See the section "Destructuring" in *Reference Guide to Symbolics-Lisp*.

2.4 Improvements to Lisp in Release 6.0

2.4.1 defflavor Now Accepts the Option :required-init-keywords

defflavor now accepts the option **:required-init-keywords**.

:required-init-keywords Option for defflavor

Specifies keywords that must be supplied. The arguments are keywords. It is an error to try to make an instance of this flavor or any incorporating it without specifying these keywords as arguments to **make-instance** (or to **instantiate-flavor**) or as a **:default-init-plist** option in a component flavor. This error can often be detected at compile time.

2.4.2 set-syntax-macro-char Takes an Optional Fourth Argument

set-syntax-macro-char takes an optional fourth argument, *non-terminating-p*. If *non-terminating-p* is **nil** (the default), **set-syntax-macro-char** makes a normal macro character. If it is **t**, **set-syntax-macro-char** makes a nonterminating macro character. A nonterminating macro character is a character that acts as a reader macro if seen between tokens, but if seen inside a token it acts as an ordinary letter and does not terminate the token.

Example:

```
(set-syntax-macro-char #/π '(lambda (&rest ignore) 'pi) readtable nil)
'(πa) is a list of two elements, '(aπb) is a list of three elements.

(set-syntax-macro-char #/π '(lambda (&rest ignore) 'pi) readtable t)
'(πa) is a list of two elements, '(aπb) is a list of one element.
```

2.4.3 The Reader Now Accepts Floating-point Infinity

The reader recognizes IEEE floating-point infinity. The syntax for infinity is as follows:

- A required plus or minus sign
- The digit "1"
- Any of the Common Lisp exponent mark characters
- The exponent character, which must be an infinity sign: ∞

For example, +1e∞.

2.4.4 si:install-microcode Takes a Second Optional Argument

si:install-microcode takes a second optional argument: *boot-file-to-update.* If *boot-file-to-update* is not given, the default prompts the user for the pathname of a boot file to update. If a pathname is given, that pathname is used as the boot file to update without a question. If the keyword **:no-boot-file-update** is given, no update is done and no question is asked.

2.4.5 lambda is Now a Special Form

lambda *lambda-list body...* *Special Form*

Provided, as a convenience, to obviate the need for using the **function** special form when the latter is used to name an anonymous (lambda) function. When **lambda** is used as a special form, it is treated by the evaluator and compiler identically to the way it would have been treated if it appeared as the operand of a **function** special form. For example, the following two forms are equivalent:

```
(my-mapping-function (lambda (x) (+ x 2)) list)

(my-mapping-function (function (lambda (x) (+ x 2))) list)
```

Note that the form immediately above is usually written as:

```
(my-mapping-function #'(lambda (x) (+ x 2)) list)
```

The first form uses **lambda** as a special form; the latter two do not use the **lambda** special form, but rather, use **lambda** to name an anonymous function.

Using **lambda** as a special form is incompatible with Common Lisp.

2.4.6 The Interpreter Understands Declarations

Declarations are understood by the interpreter as well as the compiler. Formerly, declarations were meaningful only to the compiler. See the section "Declarations" in *Reference Guide to Symbolics-Lisp*.

2.4.7 loop Now Supports Iteration Over Hash Tables or Heaps

loop now has iteration paths that support iterating over each entry in a hash table or a heap.

See the section "**loop** Iteration Over Hash Tables or Heaps" in *Reference Guide to Symbolics-Lisp*.

2.4.8 Zero-dimensional Arrays Are Now Supported

Zero-dimensional arrays are now supported. To create one, supply **nil** as the *dimensions* argument to **make-array**.

2.4.9 array-pop Takes an Optional Second Argument

array-pop takes an optional second argument:

array-pop *array* &optional (*default* **nil**)

The optional second argument, if supplied, is the value to be returned if the array is empty. If **array-pop** is called with one argument and the array is empty, it signals an error.

See the function **array-pop** in *Reference Guide to Symbolics-Lisp*.

2.4.10 Standard Variable Bindings Now Guarantee Consistent Behavior in Break and Debugging Loops

There are two new **defvar** types to define variables that have standard values to which they revert at warm boot time or in breakpoint loops. For documentation on them:

See the special form **defvar-resettable** in *User's Guide to Symbolics Computers*.
See the special form **defvar-standard** in *User's Guide to Symbolics Computers*.

There are several new functions for dealing with standard variables. For more information on these functions:

See the special form **setq-standard-value** in *User's Guide to Symbolics Computers.*
See the macro **standard-value-let** in *User's Guide to Symbolics Computers.*
See the macro **standard-value-let*** in *User's Guide to Symbolics Computers.*
See the macro **standard-value-progv** in *User's Guide to Symbolics Computers.*

For a list of the currently defined standard variables, their standard values, and their valid values: See the section "Standard Variables" in *User's Guide to Symbolics Computers.*

2.4.11 Break and the Debugger Now Bind readtable to si:standard-readtable

The variable **readtable** is now bound to the value of the variable **si:standard-readtable** in break loops and the Debugger.

See the variable **si:standard-readtable** in *User's Guide to Symbolics Computers.*

3. New Feature in Release 6.0: Symbolics Common Lisp

Symbolics Common Lisp (SCL) is available in Release 6.0. Symbolics Common Lisp (SCL) is an enhanced version of Common Lisp that contains all of the useful features of Zetalisp.

SCL is built on top of the normal Symbolics Lisp Machine system, known as Zetalisp. SCL enables you to write programs that can be transported between the 3600-family machines and other machines that run Common Lisp implementations. In a future release Symbolics Common Lisp will become the standard language and Zetalisp will continue to be supported by means of a compatibility package.

Source files can contain a new Syntax attribute, indicating either Zetalisp or Symbolics Common Lisp. For further information: See the section "Syntax and Base Attributes in Source Files", page 9.

See the section "Symbolics Common Lisp" in *Reference Guide to Symbolics-Lisp*.

4. Changes to Zmacs in Release 6.0

4.1 Incompatible Changes to Zmacs in Release 6.0

4.1.1 New loop Indentor

Zwei now indents code within a **loop** macro in a more attractive way than it did in the past. The TAB key indents the code while recognizing and dealing appropriately with **loop** keyword clauses. This new indentation style is a change in the Zmacs user interface for writing Lisp code. You might want to know how to turn it off because it indents new code in a style that is inconsistent with existing code.

To turn off the new **loop** indentor, include the following flag in your init file:

```
(SETF ZWEI:*INHIBIT-FANCY-LOOP-INDENTATION* T)
```

The initial value for this flag is **nil**; **t** reverts to the old-style indentor.
See the section "Indentation in **loop** Macros" in *Text Editing and Processing*.

4.1.2 Ztop Mode No Longer Supported

Ztop Mode in Zmacs is no longer supported and has been removed from the system.

4.1.3 Save All Files (m-X) Renamed to Save File Buffers (m-X)

The new name Save File Buffers (m-X) more accurately reflects this command's action, since it is a request to save the Zmacs buffers that are associated with a file, and only those buffers. See the section "Saving Buffers" in *Text Editing and Processing*.

4.2 Improvements to Zmacs in Release 6.0

4.2.1 Macro Expand Expression All Now Bound to m-sh-M

m-sh-M is like c-sh-M, Macro Expand Expression, except that it expands as far down as it can, rather than just expanding the outermost form. See the section "Macro Expand Expression All" in *Text Editing and Processing*.

4.2.2 SCROLL and m-SCROLL Now Display Next Screen and Previous Screen

The SCROLL key displays the next screenful of text, the same as c-V. m-SCROLL displays the previous screenful of text, the same as m-V.

4.3 New Features in Zmacs in Release 6.0

4.3.1 New Zwei Command: Copy Mouse (C-(m))

Copy mouse inserts the object on which you click at the cursor position. This command allows you to build a program or document by selecting things already appearing on your screen. Position the cursor where you want the object to appear; hold down the CTRL key and click middle on the object you want to copy: it is inserted as though you had just typed it. If you change your mind, and want to remove what you have just inserted, press c-W, and it is removed.

The object to be copied can be a word, a printed representation of a Lisp symbol, a parenthesized or quoted group of words, a printed representation of a lisp list or string, or a line. What object is picked up by clicking c-(M) on it is determined by the same rules as **Mouse Mark Thing**, or (M) in Lisp Mode. That is:

- Clicking after the last visible character of a line or before the first visible character of a line copies the whole line.
- Clicking on a word picks up that whole word, including any punctuation. The following examples illustrate the meaning of "whole word" in this context, and the convenience of using Copy mouse on the printed representation of LISP objects:
  - "ACME-VMS:SYMBOLICS:[REL6...]*.*;*"
  - fs:set-logical-pathname-host
- Clicking on an open or close parenthesis copies the text between that parenthesis and its matching parenthesis, including both parentheses. For example, clicking on the first open parenthesis of a LISP form yields the entire form:

```
(fs:set-logical-pathname-host "SYS"
   :translations
      '(("SYS:**;*.*.*" "ACME-LISPM:>Rel-6>**>*.*.*")))
```

- Similarly, clicking on an open or close square bracket, or angle bracket (that is, any of the following: [ ] < >), picks up the text between the delimiters, including the delimiters. For example, clicking c-(M) on the opening square bracket yields:
  - [REL6...]
- Clicking on an open or close quotation mark (\") copies the whole quoted string.
- Clicking between words copies all text up to the end of the next word (or possible symbol printname).

Appropriate spaces are put before the inserted object, if needed. See the section "Mouse Documentation Line in Zmacs" in *Text Editing and Processing*.

4.3.2 Two New Zmacs Commands for Recompiling and Reloading Patches

Two new Zmacs commands have been added — Recompile Patch (M-X) and Reload Patch (M-X).

4.3.2.1 Recompile Patch (M-X)

Recompile Patch (M-X) recompiles an existing patch file. This command is useful when, for example, an existing patch needs to be edited or a compiled patch file becomes damaged in some way. Never recompile a patch manually or in any way other than using the Recompile Patch command. This command ensures that source and object files are stored where the patch system can find them.

Use Recompile Patch with caution! Recompiling a patch that has already been loaded by other users can cause divergent world loads.

4.3.2.2 Reload Patch (M-X)

Reload Patch (M-X) reloads an existing patch file. This command makes it easy to reload a patch file without having to know its pathname.

4.3.3 Three New Zmacs Commands for Formatting Text

The new extended commands Format Region (M-X), Format Buffer (M-X), and Format File (M-X) display text in a formatted style using *environments* and *commands* that you embed in the text. You can send the formatted text to a Symbolics LGP-1 printer (no other printer is supported) by giving the command a numeric argument.

See the section "Zmacs Commands for Formatting Text" in *Text Editing and Processing*.

4.3.4 Two New Zmacs Commands for Reverting Buffers

Two new Zmacs commands have been added which are particularly useful when more than one person works on the same code — Refind File (M-X) and Refind All Files (M-X). These commands enable you to make sure that you work on the most up-to-date version, regardless of who updates it, at all times.

For more information about these commands, see the sections:

See the section "Refind File" in *Text Editing and Processing*.
See the section "Refind All Files" in *Text Editing and Processing*.

5. Changes to Utilities in Release 6.0

5.1 New Features in Utilities in Release 6.0

5.1.1 Ephemeral-object Garbage Collection in Release 6.0

Ephemeral-object garbage collection has been implemented in Release 6.0 *Ephemeral-object garbage collection* is a method by which the scavenger agents can pay special attention to short-lived, or ephemeral, objects. It is effective on any area having the **:gc :ephemeral** characteristic as specified by **make-area**. The **working-storage-area** has the ephemeral characteristic by default; since it is the initial value of **default-cons-area**, objects created with no area specification are subject to ephemeral-object garbage collection while it is turned on.

The overall effects are as follows:

- All objects created in ephemeral areas while the ephemeral collector is operating are considered ephemeral objects.
- The ephemeral-object garbage collector has means of tracking ephemeral objects, to avoid having to scan all of virtual memory for possible references to them.
- Garbage collection tends to increase the locality of objects and their references, so that ephemeral objects and their references are likely to be concentrated on relatively few pages.
- The above factors combine to dramatically reduce the amount of paging the garbage collector must do to find and process garbage, compared with the "dynamic" method, which operates on all of dynamic space rather than just the ephemeral portion of it. They also mean that when the dynamic (nonephemeral) objects are eventually garbage-collected, dynamic space contains less garbage than would otherwise be the case.

Before turning on the ephemeral-object garbage collector for the first time, it is necessary to run a hardware diagnostic test on your Symbolics Lisp Machine. The ephemeral-object garbage collector exercises some hardware in the machine that has not been used in the past. The diagnostic test provided with Release 6.0 indicates whether or not a problem will occur when the ephemeral-object garbage collector is turned on. For instructions on running the diagnostic test: See the document *Installation and Site Operations.*

If the diagnostic test succeeds, you can turn on the ephemeral garbage collector. By default, **gc-on** or the Start GC command enables the ephemeral collector along with dynamic-object garbage collection.

For the newly corrected documentation on **gc-on**: See the section "Clarification of **gc-on** Printed Documentation", page 80.

For a more detailed description of ephemeral-object garbage collection: See the section "Ephemeral-object Garbage Collection" in *Internals, Processes, and Storage Management*.

5.1.2 New Feature in Release 6.0: the Document Examiner

The Document Examiner is a utility for finding and reading documentation on line. It is available via SELECT D, the System menu, and the command Select Activity Document Examiner. Some Document Examiner commands are also available in the editor and in Lisp Listeners and **break** loops.

For brief information, in the Document Examiner type Help or click left on [Help]. For complete documentation, click middle on [Help] or: See the section "Using the Online Documentation System" in *User's Guide to Symbolics Computers*.

5.2 Improvements to Utilities in Release 6.0

5.2.1 compiler:make-obsolete Now Makes a Flavor or Structure Obsolete

compiler:make-obsolete now takes an optional third argument, the definition-type of the definition to make obsolete. Formerly, **compiler:make-obsolete** worked only on functions. Now it knows how to make flavors and structures obsolete, as well as functions.

See the special form **compiler:make-obsolete** in *Program Development Utilities*.

5.2.2 Changes to Patch Files

In Release 6 patch files are organized differently. Individual patches for each major version of a system reside in their own subdirectory. The patch directory file resides in the same directory as the patch files for that version; the system version-directory file resides in the immediately superior directory. Formerly, all patch files (the system version-directory file, the patch directory file, and all individual patch files) for all versions of a system were stored together.

For more information about the organization of patch files in Release 6: See the section "Organization of Patch Files" in *Program Development Utilities*.

The file types of the system version-directory file have changed for some hosts since Release 5. File types in parentheses are supported for compatibility.

Host	*File types of the system version-directory file*	
	Release 6	*Release 5*
TOPS-20	SYSTEM-DIR	PATCH-DIR
UNIX 4.1	sd	sd
UNIX 4.2	system-dir (sd)	patch-dir (sd)
VMS 3.0	SPD	SPD
VMS 4.0	SPD	SPD
ITS	(SDIR)	(PDIR)
LMFS	system-dir (patch-dir, directory)	patch-dir or directory
Multics	system-dir	patch-dir

To determine the names of your patch files, use **si:patch-system-pathname**. Although it has existed for some time, this function had never been documented. See the function **si:patch-system-pathname** in *Program Development Utilities*.

You need not take any action to accommodate these changes to patch files. Release 6 tries to be compatible with Release 5; the patch facility reverts to the old naming scheme if it cannot find the patch file by the new scheme. This means that you should be able to load systems made on Release 5 under Release 6, but not vice versa.

5.2.3 :selective Option for load-patches Has Changed

The **:selective** option for **load-patches** offers a new choice — highest.

For each patch **:selective** displays the patch comment and then asks you whether or not to load the patches. The choices are Y, N, P, or H: yes, no, proceed, or highest. Answering P turns off selective mode for any remaining patches to the current system. H means highest patch number to load. If you do not specify a limit, it loads all patches from the present level for a given system.

5.2.4 New Function note-private-patch Adds Private Patch to Your World

note-private-patch, a new function, adds a private patch to the database in your world and includes the name of the patch in your herald.

See the function **note-private-patch** in *Program Development Utilities*.

5.2.5 New Function si:map-system-files Operates on a Declared System

The function **si:map-system-files** maps a function over each file in the specified version of the system. See the function **si:map-system-files** in *Program Development Utilities*.

5.2.6 New Function si:set-system-file-properties Operates on a Declared System

The function **si:set-system-file-properties** sets the properties of each file in the specified version of the system. See the function **si:set-system-file-properties** in *Program Development Utilities*.

6. Changes to the User Interface in Release 6.0

6.1 Incompatible Changes to the User Interface in Release 6.0

6.1.1 Input Editor Options Now Specified Dynamically

In the past, input editor options were accumulated as arguments to reading functions and eventually passed to the input editor as the first argument to the **:rubout-handler** message. Now you specify input editor options dynamically, using the special forms **with-input-editing-options** and **with-input-editing-options-if**. You can use these special forms to supply input editor options for high-level functions like **prompt-and-read** and **fquery**.

The optional *input-editor-options* argument to **read, read-or-end, read-for-top-level, readline, readline-trim,** and **readline-or-nil** is obsolete. The argument is supported in this release for compatibility.

See the special form **with-input-editing-options** in *Reference Guide to Streams, Files, and I/O*. See the special form **with-input-editing-options-if** in *Reference Guide to Streams, Files, and I/O*.

6.1.2 Change to Subforms of with-input-editing

The *input-editor-options* and *parameters* "arguments" to **with-input-editing** are obsolete. Input editor options are now specified dynamically: See the special form **with-input-editing-options** in *Reference Guide to Streams, Files, and I/O*. See the special form **with-input-editing-options-if** in *Reference Guide to Streams, Files, and I/O*. It is no longer necessary to supply *parameters*, a list of lexically external variables referred to in the body of the **with-input-editing** form, because **with-input-editing** now converts its body to a lexical closure.

The two remaining "arguments", both optional, are the stream from which characters are read and a keyword specifying the activation characters for the input editor. See the special form **with-input-editing** in *Reference Guide to Streams, Files, and I/O*.

The *input-editor-options* and *parameters* "arguments" are supported in this release for compatibility.

6.1.3 :input-editor Message to Interactive Streams Replaces :rubout-handler

The **:rubout-handler** message to interactive streams is obsolete; use the **:input-editor** message instead. The **:input-editor** message does not take an *input-editor-options* argument, as the **:rubout-handler** message did. Input editor options are now specified dynamically: See the special form **with-input-editing-options** in *Reference Guide to Streams, Files, and I/O*. See the

special form **with-input-editing-options-if** in *Reference Guide to Streams, Files, and I/O.*

The **:rubout-handler** message is still supported in this release for compatibility.

:input-editor *function* &rest *arguments* *Message*

This is supported by interactive streams such as windows. It is described in its own section: See the section "The Input Editor Program Interface" in *Reference Guide to Streams, Files, and I/O.*

Most programs should not send this message directly. See the special form **with-input-editing** in *Reference Guide to Streams, Files, and I/O.*

6.1.4 Variable si:*typeout-default* Replaces tv:rh-typeout-default

The variable **tv:rh-typeout-default** has been renamed to **si:*typeout-default***, and its default value has been changed from **:insert** to **:overwrite**. This variable controls the style of typeout (for example, warnings and help messages) performed by the input editor.

6.1.5 tv:*escape-keys* and tv:*system-keys* Renamed to tv:*function-keys* and tv:*select-keys*

The variable **tv:*escape-keys*** has been renamed to **tv:*function-keys***, and the variable **tv:*system-keys*** has been renamed to **tv:*select-keys***. Instead of modifying these variables directly, use the function **tv:add-function-key** to add a new FUNCTION key, and use the function **tv:add-select-key** to add a new SELECT key.

6.1.6 Replacing io-buffer-output-function and Binding tv:kbd-tyi-hook Are Obsolete

In previous releases, you could change the way the system intercepts special characters on input to a window by replacing the **io-buffer-output-function** of the window's I/O buffer or by binding the variable **tv:kbd-tyi-hook**. These techniques are now obsolete. To change the way the system intercepts special characters, bind the variable **sys:kbd-intercepted-characters**.

For more information: See the section "Intercepted Characters" in *Programming the User Interface.*

6.1.7 :mouse-or-kbd-tyi and :mouse-or-kbd-tyi-no-hang Messages Obsolete

The **:mouse-or-kbd-tyi** and **:mouse-or-kbd-tyi-no-hang** messages to windows are obsolete. The **:mouse-or-kbd-tyi** and **:mouse-or-kbd-tyi-no-hang** methods of **tv:stream-mixin** have been removed. Use **:any-tyi** and **:any-tyi-no-hang** instead.

6.1.8 :item-list Message to Windows is Obsolete

The **:item-list** message to windows, used to create and display a list of mouse-sensitive items, is obsolete. It has been replaced by the new function **si:display-item-list**.

All interactive streams now support the **:item** message, whether or not they support mouse sensitivity. See the section "Interactive Streams and Mouse-sensitive Items" in *Programming the User Interface*.

6.1.9 New Language for Specifying Frame Constraints

The language used to specify constraints in constraint frames has been changed. The new language is more straightforward than the old. A new init option for **tv:basic-constraint-frame**, **:configurations**, replaces the **:constraints** option.

The **:constraints** option and the old language are supported in this release for compatibility. To convert a list that was the argument for the **:constraints** option to a list that can be used as the argument for the **:configurations** option, use the function **tv:back-convert-constraints**.

For more information: See the section "Specifying Panes and Constraints" in *Programming the User Interface*.

6.1.10 Change in Optional Argument to read-or-end

The second optional argument to **read-or-end** was previously an eof-option. It is now a reading function to be called to read input if a nonwhitespace character is encountered. The default is **read-expression**.

6.1.11 Changes to fquery Options

An **fquery** help function, specified by the **:help-function** option, now takes one argument (the stream) instead of three. This change is incompatible.

fquery has a new option, **:status**, that directs **fquery** to return the symbol **:status** if **query-io** is a window that becomes deexposed or deselected while **fquery** is waiting for single-character input.

:help-function Specifies a function to be called if the user presses the HELP key. The default help function simply lists the available choices. Specifying **nil** disables special treatment of HELP. If you specify a help function, it should take one argument, the stream on which to display the help message. The function can get the list of available choices from the value of the special variable **format:fquery-choices**.

:status This option takes effect only if **query-io** is a window and **:type** is **:tyi**. If the value is **:selected** and the window becomes deselected

while **fquery** is waiting for input, **fquery** returns **:status**. If the value is **:exposed** and the window becomes deexposed or deselected while **fquery** is waiting for input, **fquery** returns **:status**. If the value is **nil**, **fquery** continues to wait for input when the window is deexposed or deselected. The default is **nil**.

This option is intended for queries that appear in temporary windows that might become deexposed or deselected before the user responds.

See the function **fquery** in *Programming the User Interface*.

6.1.12 Changes to prompt-and-read Options

The **:number**, **:number-or-nil**, **:pathname**, **:pathname-or-nil**, and **:host-list** **prompt-and-read** options have been changed incompatibly. The addition of keyword arguments has made other options obsolete.

- **:number** and **:number-or-nil** now read input as a decimal number by default; formerly, by default they read input in the base that was the value of **ibase**. The **:input-radix** keyword for these types has been changed to **:base**, though **:input-radix** is supported in this release for compatibility.

  In other words, **(prompt-and-read :number)** is now the same as **(prompt-and-read '(:number :base 10.))**. To get the former behavior, use **(prompt-and-read `(:number :base ,ibase))**.

- The default version for **:pathname** and **:pathname-or-nil** is now **nil**; formerly it was **:newest**. To supply a default version of **:newest**, use the **:default-version** argument for these options.

- **:host-list** no longer accepts the **:chaos-only** keyword. It has been replaced by **:host-type**.

New keyword arguments have been added for the following options: **:date**, **:past-date**, **:date-or-never**, **:past-date-or-never**, **:number-or-nil**, **:decimal-number**, **delimited-string**, **:delimited-string-or-nil**, **:font-list**, **:host**, **:host-or-local**, **:pathname-host**, **:host-list**, **:keyword-list**, **:pathname**, **:pathname-or-nil**, and **:pathname-list**.

The keyword arguments to **:delimited-string**, **:number**, **:date**, **:pathname**, and **:host** make some other options obsolete. For example, instead of **:number-or-nil**, use **(:number :or-nil t)**. The obsolete options are supported in this release for compatibility. Following is a list of the obsolete options:

Option	*Makes obsolete*
:delimited-string	**:delimited-string-or-nil**

:number	**:number-or-nil, :decimal-number, :decimal-number-or-nil**
:date	**:past-date, :date-or-never, :past-date-or-never**
:pathname	**:pathname-or-nil**
:host	**:host-or-local, :pathname-host**

The following options are new: **:class, :complete-string, :flavor-name, :font, :function-spec, :integer, :keyword, :object, :object-list,** and **:symbol.**

See the function **prompt-and-read** in *Programming the User Interface.*

6.1.13 Changes to tv:choose-variable-values Variable Types

The **:number** and **:number-or-nil tv:choose-variable-values** variable types have been incompatibly changed. Formerly, values for these variables were read in the base that was the value of **ibase** and printed in the base that was the value of **base.** Now these types take a **:base** parameter that specifies the input and output base. If **:base** is not specified, the values are read and printed in decimal.

The **:number** type also takes an **:or-nil** parameter. If this is not **nil, nil** is accepted as a variable value.

The parameters for **:number** make the **:number-or-nil, :decimal-number,** and **:decimal-number-or-nil** types obsolete. Also, the new type **:expression** replaces the obsolete **:sexp.** The obsolete types are supported in this release for compatibility.

The following variable types are new: **:expression, :eval-form, :integer, :inverted-boolean, :past-date-or-never,** and **:time-interval-60ths.**

See the section "Predefined **tv:choose-variable-values** Variable Types" in *Programming the User Interface.*

6.1.14 Change to define-prompt-and-read-type Dispatch Functions

The dispatch functions defined by **define-prompt-and-read-type** are no longer called with the same arguments. Formerly the first two arguments were the stream and a list of input editor options. Now the arguments depend on the first argument to **prompt-and-read.** If the first argument to **prompt-and-read** is just *keyword,* the dispatch function is called with no arguments. If the first argument to **prompt-and-read** is (*keyword . type-args*), the arguments to the dispatch function are the elements of *type-args.* These are a series of alternating keywords and values.

The second subform of **define-prompt-and-read-type,** used to construct the parameter list of the dispatch function, has changed. Formerly this subform was just the parameter list of the dispatch function. Now it is **nil** if no *type-args* are

allowed, or else a list of **&key** elements for the dispatch function's parameter list. **define-prompt-and-read-type** inserts **&key** in the parameter list itself; **&key** should not appear in the second subform.

The third subform of **define-prompt-and-read-type** can now be **nil**, a **format** control string, a list of a **format** control string and **format** control args, or a form to be evaluated. This subform is used to generate *input-type* in the default prompt, "Enter *input-type*: ":

subform	*input-type*
nil	"a " followed by the print name of the type keyword.
format control string	Generated by calling **format** with arguments of **t** and the control string when it is time to display the prompt.
list of **format** control string and args	Generated by calling **format** with arguments of **t**, the control string, and the control args when it is time to display the prompt. The control args can examine any of the parameters in the second subform.
form	Generated by evaluating the form when it is time to display the prompt. The form can examine any of the parameters in the second subform. It should send output to **standard-output**.

See the special form **define-prompt-and-read-type** in *Programming the User Interface.*

6.1.15 Audio Wavetable Size Increased From 256 to 1024 Words

The size of wavetables has been changed from 256 words to 1024 words for greater audio fidelity. For details on the use of wavetables: See the section "The Polyphony Feature" in *Programming the User Interface.*

6.1.16 :clear-eof Message to Windows is Obsolete

In Release 5.0, the **:clear-eof** message to windows was renamed to **:clear-rest-of-line**, but windows continued to accept the **:clear-eof** message for compatibility. In Release 6.0 windows no longer accept **:clear-eof**.

The **:clear-eof** message was renamed because it had two different meanings. For windows, it meant to clear the window from the cursor position to the bottom. For noninteractive streams, it means to read the EOF indicator, so that data past the EOF could be read.

6.2 New Features in the User Interface in Release 6.0

6.2.1 New Feature in Release 6.0: the Command Processor

The command processor is a utility program that collects arguments on behalf of a command and then runs that command for you. By default, the command processor is on in all Lisp Listeners and **break** loops.

For a description of the command processor and its user interface: See the section "Communicating with the Lisp Machine" in *User's Guide to Symbolics Computers*.

For information on the command processor programming interface, including the command processor reader and the facility for defining commands: See the section "The Command Processor Program Interface" in *Programming the User Interface*.

6.2.2 New Feature: Window Graying

Screens and frames can now *gray* areas that contain no windows or that contain windows that are not fully exposed. To gray an area of the screen is to cover it with a semitransparent texture pattern.

By default, the main screen now covers deexposed inferiors with a stipple pattern and background areas with a white pattern. You can change these by using the functions **tv:set-screen-deexposed-gray** and **tv:set-screen-background-gray**. Call these functions with an argument of **nil** to disable graying entirely. The value of the variable **tv:*gray-arrays*** is a list of variables bound to other graying specifications that can be used as arguments to these functions.

For more information: See the section "Window Graying" in *Programming the User Interface*.

6.2.3 New Input Editor Commands: PAGE, COMPLETE, c-?

Pressing PAGE while in the input editor erases input editor typeout, such as typeout from the HELP or c-sh-A commands.

In Lisp Listeners and **break** loops, COMPLETE attempts to complete the current symbol over the set of possibilities specified by definitions in Zmacs buffers. c-? displays the possible completions of the current symbol.

6.2.4 New Reading Functions

The following reading functions are new:

sys:read-character Reads and returns a single character. This function displays notifications and help messages and reprompts at appropriate times. It is used by **fquery** and the **:character** option for **prompt-and-read**.

readline-no-echo Reads a line of input without echoing the input, and returns the input as a string, without the terminating character. This function is used to read passwords and encryption keys.

read-expression Like **read-for-top-level**, except that if it encounters a top-level end-of-file it just beeps and waits for more input. This function is used by the **:expression** option for **prompt-and-read**.

read-or-character This function is like **read-expression**, except that if it is reading from an interactive stream and the user types a delimiter as the first character or the first character after only whitespace characters, it returns four values: **nil**, **:character**, the character code of the delimiter, and any numeric argument to the delimiter.

read-and-eval Calls **read-expression** to read a form, without completion. It then evaluates the form and returns the result.

read-form Like **read-expression**, except that it assumes that the returned value will be given immediately to **eval**. This function is used by the Lisp command loop and by the **:eval-form** and **:eval-form-or-end** options for **prompt-and-read**. By default, it offers completion over definitions in Zmacs buffers, and it catches simple unbound-variable and undefined-function errors.

For more information, see the description of each function.

6.2.5 New Message to Input Streams: :input-wait

Use **:input-wait** to wait until input is available from an interactive stream or some other condition, such as the arrival of a notification, is met. Any stream that can become the value of **terminal-io** must support **:input-wait**.

See the message **:input-wait** in *Reference Guide to Streams, Files, and I/O*.

6.2.6 New Message to Streams: :interactive

:interactive *Message*

The **:interactive** message to a stream returns **t** if the stream is interactive and **nil** if it is not. Interactive streams, built on **si:interactive-stream**, are streams designed for interaction with human users. They support input editing. Use the **:interactive** message to find out whether a stream supports the **:input-editor** message.

6.2.7 New Message to Interactive Streams: :noise-string-out

While inside the input editor, a read function can send an interactive stream a **:noise-string-out** message to display a string that is not to be treated as input. See the method **(:method si:interactive-stream :noise-string-out)** in *Reference Guide to Streams, Files, and I/O.*

6.2.8 New Input Editor Help Options

The input editor has four new help options. For details on each of them:

See the option **:complete-help** in *Reference Guide to Streams, Files, and I/O.*
See the option **:partial-help** in *Reference Guide to Streams, Files, and I/O.*
See the option **:merged-help** in *Reference Guide to Streams, Files, and I/O.*
See the option **:brief-help** in *Reference Guide to Streams, Files, and I/O.*

For information on all input editor options: See the section "Input Editor Options" in *Reference Guide to Streams, Files, and I/O.*

6.2.9 New Input Editor Options

The input editor has several new options. For information on them:

See the option **:input-history-default** in *Reference Guide to Streams, Files, and I/O.*
See the option **:blip-handler** in *Reference Guide to Streams, Files, and I/O.*
See the option **:editor-command** in *Reference Guide to Streams, Files, and I/O.*
See the option **:input-wait** in *Reference Guide to Streams, Files, and I/O.*
See the option **:input-wait-handler** in *Reference Guide to Streams, Files, and I/O.*
See the option **:suppress-notifications** in *Reference Guide to Streams, Files, and I/O.*
See the option **:notification-handler** in *Reference Guide to Streams, Files, and I/O.*

For information on all input editor options: See the section "Input Editor Options" in *Reference Guide to Streams, Files, and I/O.*

6.2.10 New Special Forms: tv:with-mouse-and-buttons-grabbed, tv:with-mouse-and-buttons-grabbed-on-sheet

tv:with-mouse-and-buttons-grabbed &body *body* *Special Form*

The forms in *body* are evaluated with the mouse and buttons grabbed. When the buttons are grabbed, the mouse process does not maintain the

value of **tv:mouse-last-buttons**. Instead, the user process can read directly from the mouse buttons, without losing clicks that the mouse process might fail to notice. Within the body of this form, you can call the functions **tv:mouse-wait**, **tv:wait-for-mouse-button-down**, **tv:wait-for-mouse-button-up**, and **tv:mouse-buttons**.

tv:with-mouse-and-buttons-grabbed-on-sheet (&optional (*sheet* **'self**)) &body *body* *Special Form*

Like **tv:with-mouse-and-buttons-grabbed**, except that the mouse is confined to *sheet*. During execution the variables **tv:mouse-x** and **tv:mouse-y** are relative to the window's outside coordinates. The default value of *sheet* is **self**, so if *sheet* is not supplied, this form needs to appear inside a method or defun-method of a window flavor.

6.2.11 New Message to Windows: :set-font-map-and-vsp

:set-font-map-and-vsp *new-map new-vsp* of **tv:sheet** *Method*

Changes the font map and vsp of the window.

new-map can be an array of font descriptors or a list of font descriptors, as with the argument to the **:set-font-map** message. However, if the *new-map* argument to **:set-font-map-and-vsp** is **nil**, the font map is not changed.

new-vsp is an integer representing the new vsp, or **nil**, meaning not to change the vsp.

6.2.12 New Notification System

A new notification system has been installed. The undocumented flavors **tv:notification-mixin** and **tv:pop-up-notification-mixin** no longer exist.

A process uses **tv:notify** to notify the user. A central notification delivery process tries to give the process associated with the selected window a chance to accept the notification. The user process can wait for a notification by examining the locative returned by the **:notification-cell** message to the selected window. It can receive a notification by sending the window a **:receive-notification** message. It can use **sys:display-notification** to display a notification. If the user process does not accept a notification, the notification delivery process usually tries to display the notification itself, in either a pop-up window or the selected-window. The user process can use the **with-notification-mode** special form to determine what the delivery process does with notifications the user process doesn't accept.

All notifications since cold booting are displayed in a scroll window obtained by pressing SELECT N or calling **display-notifications**. They are also available to the Show Notifications command.

See the section "Notifications" in *Programming the User Interface.*

6.2.13 New Time Functions

time-elapsed-p *increment initial-time* &optional (*final-time* **(time)**) *Function*

Returns **t** if at least *increment* 60ths of a second have elapsed between *initial-time* and *final-time*. Otherwise, returns **nil**.

initial-time and *final-time* should be time values as returned by the **time** function. *final-time* defaults to the result of **(time)**.

Example:

```
(defun process-sleep (interval &optional (whostate "Sleep"))
  (process-wait whostate #'time-elapsed-p interval (time)))
```

time:parse-universal-time-relative *date-spec reference-date-spec* *Function*
&optional (*future-p* **t**)

Like **time:parse-universal-time**, except that *date-spec* is parsed relative to *reference-date-spec*. The returned values are the same as those of **time:parse-universal-time**.

date-spec is a string suitable as the first argument to **time:parse-universal-time**. *reference-date-spec* is a universal-time integer or a string that can be parsed as an unambiguous time. If *future-p* is **nil**, an ambiguous *date-spec* is interpreted as being in the past relative to *reference-date-spec*; otherwise, it is interpreted as being in the future. The default for *future-p* is **t**.

For example:

```
(time:parse-universal-time-relative "5 pm" "today")
```

returns the same value when evaluated anytime today, whether or not the current time is before or after 5 pm.

time:parse-present-based-universal-time *time-being-parsed* *Function*

Like **time:parse-universal-time**, except that missing components in *time-being-parsed* are defaulted to the beginning of the smallest unsupplied unit of time. The returned values are the same as those of **time:parse-universal-time**. *time-being-parsed* is a string suitable as the first argument to **time:parse-universal-time**.

For example, "5 pm" is parsed as 5 pm on the current day, whether the current time is before or after 5 pm. "Thursday" is parsed as Thursday of the current week, whether today is Wednesday or Friday. "1 June" is parsed as June 1 of the current year, whether the date is before or after June 1.

6.3 Improvements to the User Interface in Release 6.0

6.3.1 Improvements to Activity and Window Selection

A number of bugs in selecting windows and activities have been fixed. These bugs often manifested themselves in improper or unexpected selection of windows and in incorrect blinker states.

An activity is a group of windows that the user regards as a single unit. An activity is designated by a representative window from that activity. Often an activity consists of a frame and its panes. Typeout windows and their parents also have some characteristics of activities.

A new flavor, **tv:select-relative-mixin**, allows a window to participate in its superior's activity. **tv:pane-mixin** also does this, among other things. The **:alias-for-selected-windows** message returns the representative window of the receiver's activity.

Selecting an activity has been more clearly distinguished from selecting a window relative to its activity. Selecting a window relative to its activity designates that window to become the selected window when the activity is selected, but it does not change the selected activity. Such a window, if a pane, is called the selected-pane of its frame. Use the **:select** and **:mouse-select** messages to a selectable window or frame to switch activities. Use the **:select-relative** message to a selectable window (or the **:select-pane** message to a frame) to select that window relative to its activity without changing activities. Use the **tv:window-call** or **tv:window-mouse-call** special form to select a window temporarily, selecting a new activity if necessary. Use the **tv:window-call-relative** special form to select a window temporarily relative to its activity without changing activities.

When a selectable window receives a **:select-relative** message and its activity is not currently selected, it informs its superior by sending the superior an **:inferior-select** message. A window that participates in its superior's activity also sends its superior an **:inferior-select** message when it receives a **:select** message.

For details on selecting activities and windows: See the section "Activities and Window Selection" in *Programming the User Interface*.

6.3.2 Lisp Listeners and break Loops Catch Trivial Errors in the Input Editor

In a Lisp Listener or **break** loop, if you try to evaluate an unbound symbol or a list whose car is a symbol that is not defined as a function, the Lisp Listener or **break** loop now catches the error in the input editor. It offers to use a lookalike symbol in another package or lets you edit your input to correct it. Formerly such errors caused entry to the Debugger.

This feature is implemented using the new reading function **read-form**. To disable it, you can set the variable ***read-form-edit-trivial-errors-p*** to **nil**. See the function **read-form** in *Programming the User Interface*.

6.3.3 New Type for :start-typeout Message to Interactive Streams: :clear-window

The *type* argument to the **:start-typeout** message to interactive streams has a new permissible value: **:clear-window**. This informs the input editor that typeout to the window will follow, and that the window should be cleared and the typeout should appear at the top. See the method **(:method si:interactive-stream :start-typeout)** in *Reference Guide to Streams, Files, and I/O*. See the variable **si:*typeout-default*** in *Reference Guide to Streams, Files, and I/O*.

6.3.4 New Optional Argument to :replace-input Message to Interactive Streams

The **:replace-input** message to interactive streams now takes a third optional argument, which specifies what action to take when the message is sent while the input editor buffer is being rescanned. See the method **(:method si:interactive-stream :replace-input)** in *Reference Guide to Streams, Files, and I/O*.

6.3.5 New Optional Arguments to :initial-input Input Editor Option

The **:initial-input** input editor option now takes three optional arguments:

- An index into the string at which to start copying the string into the input buffer
- An index into the string at which to stop copying the string into the input buffer
- An index into the string at which to place the initial cursor position

See the option **:initial-input** in *Reference Guide to Streams, Files, and I/O*.

6.3.6 Improvements to Typeout Windows

A new flavor of typeout window, **tv:temporary-typeout-window**, saves and restores the bits of its superior window. A new special form, **tv:with-terminal-io-on-typeout-window**, executes its body with **terminal-io** bound to the typeout window of a window. When this special form is used with a window that has a temporary typeout window, the program does not have to take any action to restore the display when the typeout window goes away.

See the flavor **tv:temporary-typeout-window** in *Programming the User Interface*. See the function **tv:with-terminal-io-on-typeout-window** in *Programming the User Interface*.

6.3.7 Improvements to tv:add-function-key

Characters added to the FUNCTION key via **tv:add-select-key** are now converted to uppercase. **tv:add-function-key** has a new option, **:process**. The value is a list to be passed as the first argument to **process-run-function** when a process is created in which the function should be applied or the form evaluated. See the function **tv:add-function-key** in *Programming the User Interface.*

6.3.8 New Option for defwindow-resource: :superior

defwindow-resource now takes a **:superior** option. This is a form to be evaluated when the resource is allocated to return the superior window of the desired window. If this is not supplied, the superior is the value of **tv:mouse-sheet.** See the special form **defwindow-resource** in *Programming the User Interface.*

6.3.9 Mouse Scaling Now Works on 3600-family Computers

You can now scale mouse motion on 3600-family computers, using the variables **tv:mouse-x-scale-array** and **tv:mouse-y-scale-array**. Formerly mouse scaling worked only on the LM-2. See the section "Scaling Mouse Motion" in *Programming the User Interface.*

6.3.10 sys:%beep Now Works on 3600-family Consoles That Support Digital Audio

sys:%beep now works on 3600-family consoles that support the digital audio facilities. **sys:%beep** generates tones. The arguments, *half-wavelength* (in microseconds) and *duration,* are compatible with the version of beep that ran on the Symbolics LM-2 computer. In the following example, a 440 Hz tone is generated for 50 milliseconds.

```
(sys:%beep (// 1000000. 440. 2) 50000.)
```

6.3.11 Optional Argument :ask Added to zwei:save-all-files

zwei:save-all-files now accepts the optional argument **:ask**, which specifies that the function ask before saving each modified buffer. The default is **t**, which asks about each modified buffer; this was the previous behavior.

7. Changes to Zmail in Release 6.0

7.1 Incompatible Changes to Zmail in Release 6.0

7.1.1 c-m-Y Has Been Changed to c-X c-Y in Release 6.0

The command to yank the message being replied to into a reply has been changed from c-m-Y to c-X c-Y. A side effect of this change is that now in editing windows in mail c-m-Y yanks minibuffer commands as it does in Zmacs.

For more information about this commmand:

See the section "c-X c-Y Yank Current Message Zmail Command" in *Communicating with Other Users*.
See the variable **zwei:*prune-headers-after-yanking*** in *Communicating with Other Users*.

7.1.2 zwei:chaos-direct-send-it is Now Obsolete

If you have the form
(login-setq zwei:*mail-sending-mode* 'zwei:chaos-direct-send-it) in your Zmail init file, you should remove it. This value for **zwei:*mail-sending-mode*** was useful for sites that did not have a store-and-forward mailer. The presence of the Symbolics Store-and-Forward Mailer in Release 6 makes it obsolete.

8. Changes to the File System in Release 6.0

8.1 Improvements to the File System in Release 6.0

8.1.1 New Logical Pathname Translations

Logical pathname translation has been redone in Release 6. A summary of the changes follows:

- Translations are now much simpler, due to a syntax enhancement.
- One logical host can translate to multiple physical hosts.
- A powerful, general heuristic is provided for translating logical pathnames to VAX/VMS and UNIX filenames. This replaces the special mechanisms used in Releases 4 and 5 to handle UNIX and VAX/VMS file-name limitations in the source files.

8.1.1.1 Logical Pathname Wildcard Syntax

Logical pathnames support a wildcard syntax meaning "Match any directory, and any subdirectory, to any level." For example:

```
Show Directory SYS:**;*.BFD.*
```

Here, the Show Directory command lists all font files anywhere in the SYS hierarchy, to any level.

This corresponds to the >**> syntax for LMFS pathnames, and the [name...] syntax for VAX/VMS file specifications. See the section "LMFS Pathnames" in *Reference Guide to Streams, Files, and I/O*. See the section "VAX/VMS Pathnames" in *Reference Guide to Streams, Files, and I/O*.

This makes it easy to specify logical pathname translations on Lisp Machines and VAX/VMS. For example:

```
(fs:set-logical-pathname-host "SYS"
   :translations '(("SYS:**;*.*.*" "ACME-LISPM:>Rel-6>**>*.*.*")))

(fs:set-logical-pathname-host "SYS"
  :translations
  '(("SYS:**;*.*.*" "ACME-VMS:SYMBOLICS:[REL6...]*.*;*"))
  :no-translate nil)
```

For more information about the argument **:no-translate**: See the section "Translation Rules", page 72.

It is important to note that wherever a "**;" appears in the logical-host pathname, there must be a corresponding "wild-inferiors" pathname on the physical-host pathname.

UNIX and TOPS-20 do not have a syntax with this meaning. For these hosts, it is necessary to list explicitly each level of directory to be translated. For example:

```
(fs:set-logical-pathname-host "SYS"
  :translations
   '(("SYS:*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*.*.*")
     ("SYS:*;*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*//*.*.*")
     ("SYS:*;*;*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*//*//*.*.*")
     ("SYS:*;*;*;*;*.*.*"
      "ACME-UNIX://usr//symbolics//rel-6//*//*//*//*//*.*.*"))
  :no-translate nil)
```

8.1.1.2 Splitting Logical Hosts Across Physical Hosts

It is possible to have a logical host translate to more than one physical host. All that is needed is an explicit specification of the hosts involved, in the translation list given to **fs:set-logical-pathname-host**. For example:

```
(fs:set-logical-pathname-host "SYS"
  :translations '(("SYS:DOC;**;*.*.*" "ACME-LISPM:>Rel-6>doc>**>*.*.*")
                  ("SYS:**;*.*.*" "ACMEVAX:SYMBOLICS:[REL6...]*.*.*"))
  :no-translate nil)
```

Note that it is not necessary to specify the **:physical-host** argument to **fs:set-logical-pathname-host** as long as the host names are specified in the translation list. If the argument is specified, it serves as a default when parsing those pathnames.

8.1.1.3 Translation Rules

The logical system host **sys** comes preloaded with heuristics that eliminate characters illegal in VAX/VMS file specifications, such as "-".

The heuristics also deal with limitations in the lengths of file specifications on foreign hosts. For example, some file names can be shortened and contracted without changing their meanings. Thus, sys:io;pathnm-cometh.lisp may translate to acmevax:symbolics[rel6.io]pthnmcmth.lsp on a VAX/VMS physical host.

The system keeps careful track of these changes and does not allow two logical pathnames to translate to the same thing. On the attempt to translate a second logical pathname to a physical pathname already found as the result of a logical-pathname translation, an error is signalled. If the first attempt was due to a typographical error made by the user, and the second was due to the system translating a logical pathname, for example in response to the m-. command, the error is signalled. However, when **:no-translate nil** is used in the **fs:set-logical-pathname-host** form, the system translates all its logical pathnames when setting the logical system host; then, incorrect translations cannot be entered by mistake.

There also are special translation rules for microcode files, font files, and others, which retain the special characteristics of these file names.

9. Changes to Networks in Release 6.0

9.1 Incompatible Changes to Networks in Release 6.0

9.1.1 define-site-variable No Longer Used

In Release 6.0 the **defvar** and **add-initialization** functions should be used to define a site variable, rather than the **define-site-variable** function. For more information, see the following functions:

defvar
add-initialization

9.2 New Features in Networks in Release 6.0

9.2.1 Overview of Remote Login Capability

The remote login facilities allow up to three ASCII terminals to be connected directly via the Symbolics computer's serial ports. Any number of terminals can be connected via the network. If a modem is connected to the machine, it is also possible to dial up the machine from an ASCII terminal or from another Symbolics computer. Video operations are supported only on ASCII terminals that support ANSI X3.64 display codes (Ann Arbor Ambassador, Digital Equipment VT100, and so forth).

Network servers are available for the remote login protocols TELNET, SUPDUP, TTYLINK, and 3600-LOGIN. TELNET and SUPDUP are standard protocols used on the Arpanet. TTYLINK is a raw byte-stream. 3600-LOGIN is used only in communication between two Symbolics computers.

The following programs can be run from terminals connected via a network, a serial port, or a modem:

- Lisp Listener
- Input editor
- Debugger (not the Window Debugger)
- Command processor

Zmacs, Zmail, and other programs that use the window system or the mouse cannot be used.

The remote login facility is useful for applications such as the following:

- Examining the status of a physically distant machine, such as a file server.
- Monitoring the status of a long computation from home.
- Simple data-entry or query-and-answer applications.

Note that the remote login feature cannot support several programmers on the same machine, because program-development tools, such as Zmacs, cannot be used remotely.

For further information on remote login: See the section "Using the Remote Login Facilities" in *Networks*.

For information on the new functions dealing with remote login:

See the function **neti:ask-terminal-parameters** in *Networks*.
See the function **neti:set-terminal-parameters** in *Networks*.
See the function **neti:enable-serial-terminal** in *Networks*.
See the function **net:remote-login-on** in *Networks*.

10. Changes to the FEP in Release 6.0

FEP software is distributed in its own versions, which are separate from Lisp software releases. Release 6.0 requires any of: FEP version 17, version 18, version 22, or version 24. FEP version 24 is preferred.

11. Notes and Clarifications for Release 6.0

11.1 Use #||...||# Instead of #|...|# to Comment Out Lisp Code

#| begins a comment for the Lisp reader. The reader ignores everything until the next |#, which closes the comment. #| and |# can be on different lines, and #|...|# pairs can be nested.

Use of #|...|# always works for the Lisp reader. The editor, however, currently does not understand the reader's interpretation of #|...|#. Instead, the editor retains its knowledge of Lisp expressions. Symbols can be named with vertical bars, so the editor (not the reader) behaves as if #|...|# is the name of a symbol surrounded by pound signs, instead of a comment.

Now consider #||...||#. The reader views this as a comment: the comment prologue is #|, the comment body is |...|. and the comment epilogue is |#. The editor, however, interprets this as a pound sign (#), a symbol with a zero length print name (||), lisp code (...), another symbol with a zero length print name (||), and a stray pound sign (#). Therefore, inside a #||...||#, the editor commands which operate on Lisp code, such as balancing parentheses and indenting code, work correctly.

11.2 Clarification of What readline Returns

The documentation for **readline** states that it returns four values:

- The line as a character string, without the Newline character.
- An *eof* flag, if *eof-option* was **nil**. This is **t** if the line was terminated because end-of-file was encountered, or **nil** if it was terminated because of a RETURN, LINE, or END character.
- The character that delimited the string.
- Any numeric argument given the delimiter character.

The documentation is incorrect. The correct information is that **readline** returns two values, which are the same as the first two previously mentioned, except that if the line is already at end-of-file, **readline** returns **nil** as its first value.

- The line as a character string, without the Newline character, or if already at end-of-file, **nil**.
- An *eof* flag, if *eof-option* was **nil**. This is **t** if the line was terminated because

end-of-file was encountered, or **nil** if it was terminated because of a RETURN, LINE, or END character.

11.3 Clarification of gc-on Printed Documentation

The printed documentation of **gc-on** is in error regarding the default values of its options. The online version is correct and is reproduced here:

gc-on &key *ephemeral dynamic* *Function*
Turns garbage collection on. It is off by default. The keywords **:ephemeral** and **:dynamic** select the type(s) of garbage collection employed; the defaults are **:ephemeral t** and **:dynamic t** if no options are specified. If either option is specified, the other defaults to **nil**; this allows you to turn on one form of garbage collection and leave the other one off.

11.4 Warning Against Deleting LMFS File Partitions

Several users have tried to reduce the the size of their LMFS by deleting one or more file partitions and editing the FSPT to remove these partitions. Using this procedure resulted in an unusable LMFS.

Do not delete file partitions from your LMFS. Each LMFS partition contains pointers to all other file partitions in the LMFS. Deleting a file partition leaves the other partitions with pointers to a nonexistent file.

If you want to reduce the size of your LMFS, you must completely backup your LMFS, delete the entire existing LMFS and create a new one. The user files can then be restored into this new LMFS from the backup tapes.

11.5 Serial Stream Handling of Xon - Xoff Characters

A common problem encountered with serial streams is the handling of the XON/XOFF protocol. The FEP reads all eight bits of the XON or XOFF character even if you have specified a different number of data bits for that stream. You must determine what eight bit characters are being sent to the Symbolics Lisp Machine as the XON and XOFF characters.

For example, assume that the printer connected to the Symbolics Lisp Machine's serial port receives seven data bits with no parity. You might assume that it would send a Control-S (#O23) as the XOFF character and a Control-Q (#O21) as the XON character. The FEP, however, might be receiving an #O221 as the XON character and #O223 as the XOFF character. The difference here is that the in both cases the parity bit of each character is set.

The :OUTPUT-XON-CHARACTER and :OUTPUT-XOFF-CHARACTER options of SI:MAKE-SERIAL-STREAM are used to change the character that the FEP will recognize as the XON or XOFF character. Similarly, add the OUTPUT-XON-CHARACTER and OUTPUT-XOFF-CHARACTER options to the Interface Options of the printer's namespace object when connecting a serial ASCII printer.

See the section "Parameters for Serial I/O" in *Reference Guide to Streams, Files, and I/O*.

Index

B

B

B

C C C

D D D

E

E

F

F

G

H

K

K K

L

L L

M

N N N

P

P P P

Q

Q

Q

R

R

S

S

S

T

T

T

U

V

W