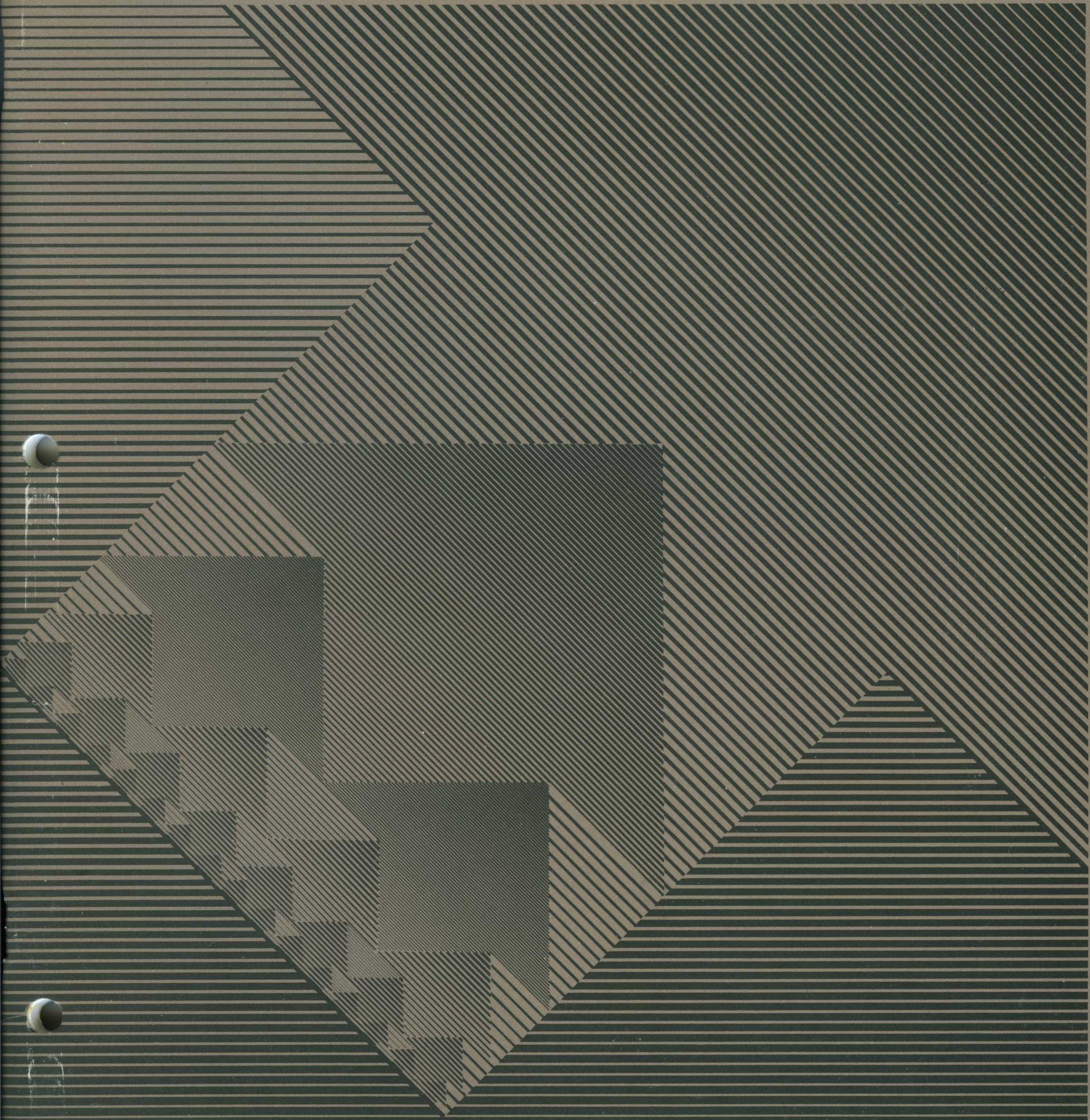
symbolics

System 210 Release Notes



symbolics inc.

System 210 Release Notes

June 1982

Prepared by Symbolics, Inc.
Written by Daniel L. Weinreb and Judith S. Hoer

symbolics inc.

System 210 Release Notes

June 1982

Cambridge, Massachusetts

TABLE OF CONTENTS

1. INTRODUCTION AND HIGHLIGHTS

These Release Notes accompany the release of System 210. They describe changes made since System 78. The changes are organized into chapters as follows:

Important changes
These changes include incompatible changes that might stop your programs from working in System 210, and commands to various programs that have been changed incompatibly or removed. Please read this section before trying to use System 210.

Changes to less frequently used features
The changes in this section might break programs that work with advanced subsystems and use advanced features.

New commands
New commands include TERMINAL commands, Zmacs commands, and Debugger commands. This section discusses changes to the user-interface of the system.

Major new facilities
New facilities include Lambda macros, the world-load compressor, VAX/VMS support, the Converse system, and a new system for handling compiler warnings.

New functions
This chapter describes new functions, special forms, flavors, and streams.

Improvements to existing functions
Several existing functions in the system have been enhanced with new features and capabilities.

Notes This section does not discuss changes. It answers some commonly asked questions and provides various news items.

The most important change since the System 78 release is that Flavors have been reimplemented to increase substantially the speed of message sending. The new implementation is mostly compatible with the old one, but all programs that use Flavors must be recompiled, and programs that use some of the more advanced features might need some minor source changes.

As with previous software releases, numerous minor bugs have been fixed and performance has been improved. Only a few of the most important bug fixes are mentioned here.

For a complete list of the changes, see the Table of Contents.

2. IMPORTANT CHANGES

These include incompatible changes that might stop your programs from working in System 210, and commands to various programs that have been changed incompatibly or removed. Please read this section before trying to use System 210.

2.1 System 210 comes with microcode 896

System 210 world loads must be run with microcode version 896. The old world loads do not work with the new microcode, and the new world loads do not work with the old microcode. If you want to switch from a System 78 band to a System 210 band, you must do a **set-current-band** and a **set-current-microload,** and cold boot. Note: since **disk-restore** does not reload the microcode, you can't use **disk-restore** to get a System 78 band if you are running System 210, or vice versa. Do not leave the current band and current microload set inconsistently; it is impossible to cold-boot! The cross-machine Debugger (CC) works when the master machine is running System 210 and the slave machine is running System 78 (but not the other way around).

2.2 Flavors changed incompatibly

Flavors have been completely reimplemented in System 210 to increase performance; message passing is now much faster. Unfortunately, certain incompatible changes were made. If your program uses Flavors, you must recompile it in System 210 to make it work in System 210. In addition, if you use some of the more advanced features of Flavors, your program might need some changes before you recompile it. The most important reason for this is that instance variables are no longer always implemented as special variables.

If you have a program that sends messages to existing objects but doesn't define any flavors of its own, you won't be affected by the changes; you don't have to recompile anything. If your program defines any flavors, it must be recompiled in System 210. The resulting compiled file cannot be loaded into System 78; you have to recompile the file in System 78 again if you want to load it into System 78.

In addition, you have to change your program before you recompile it if your program does any of the following:

- uses the **declare-flavor-instance-variables** special form

- treats instance variables as symbols, by using functions such as **set, symeval,** or **boundp** on them, or in other ways

- uses functions returned by the **get-handler-for** function or the **:get-handler-for** message

A program that does not do any of these should only need recompilation. If you have a program that does any of these, read on.

In the previous implementation of Flavors, instance variables were implemented as special variables using the **declare-flavor-instance-variables** special form. This special form is no longer available. Now use the new special form, **defun-method,** to define a function that can access instance variables.

defun-method *function-spec flavor argument-list body...* *Special Form*

Sometimes you write a function which is not itself a method, but which is to be called by methods and wants to be able to access the instance variables of the object **self**. **defun-method** is like **defun,** but the function is able to access the instance variables of *flavor.* It is valid to call the function only while executing inside a method or a **defun-method** for an object of the specified flavor, or of some flavor built upon it.

defun-method has the same effect that surrounding a **defun** by a **declare-flavor-instance-variables** used to have: the function that is defined can be called any time that **self** is an instance of a flavor, one of whose components is the specified flavor. (A flavor is considered to be one of its own components.) For now, *function-spec* must be a symbol; this restriction might be lifted in the future.

If your program has a **defselect** inside of a **declare-flavor-instance-variables,** you can convert it into the following special form:

defselect-method *function-spec flavor body...* *Special Form*

This special form is like **defselect,** but the forms of the *body* are able to access the instance variables of *flavor.* See **defun-method**.

If your program has methods with internal functions (that is, uses of **#'(lambda ...)** or **(function (lambda ...))),** and the internal function expects to be called in the environment of an instance (that is, uses **self** or an instance variable), such functions work as long as they are called from a method of the same flavor as the method in which they are used. Otherwise, the internal function should be turned into a **defun-method** or, preferably, made into a method itself.

A program that uses functions that treat instance variables as special variables (symbols) must be changed. Calls to **value-cell-location, boundp,** and **makunbound** for instance variables should be changed to uses of the special forms **variable-location, variable-boundp,** and **variable-makunbound,** respectively (see section 6.22). Calls to **set** and **symeval** should be changed to calls to **set-in-instance** and **symeval-in-instance,** respectively.

Use the new **:special-instance-variables** option of **defflavor** if you need instance variables to be bound when an instance is entered. Its format is like that of **:gettable-instance-variables**; that is, the option can be **:special-instance-variables** to declare all of the instance variables to be special variables, or it can be of the format **(:special-instance-variables a b c)** to declare only the instance variables **a, b,** and **c** to be special variables. The named variables work the old way: when any method is called, these special variables are bound to the values in the instance, and references to these variables from methods are compiled as special variable references. This detracts from the performance improvement of the the Flavors implementation, and should be avoided.

The handlers returned by the **get-handler-for** function and the **:get-handler-for** message can no longer be used as functions with **funcall.** The most common use of **get-handler-for** had been in code sequences such as:

```
(let ((h (get-handler-for self ':mess)))
  (if (not (null h)) (funcall h ':mess args...)))
```

This should be replaced by:

```
(funcall self ':send-if-handles ':mess args...)
```

(funcall-self ...) now expands into **(funcall self ...)**. Note: your code won't work if you were depending on **funcall-self** not to redo special instance variable bindings. Use **let-globally**, which is probably what you wanted, or use the **bind** subprimitive.

The macro **si:flavor-select-method** has been renamed to **si:flavor-message-handler**. User code should not be using this anyway.

The rest of this section describes how **defun-method** works and presents performance considerations.

defun-method works by defining two functions: one is a function named *function-spec*, and the other is a function named **(:defun-method** *function-spec***)**. An optimizer is also added to *function-spec* (since optimizers currently can be added only to symbols, *function-spec* is constrained to be a symbol for now). The function named *function-spec* can be called from anywhere, as long as **self** is bound to an appropriate instance. The environment is correctly set up, and the internal **:defun-method** is called. This requires calling into the Flavor system and has some performance penalty over sending a message. However, if *function-spec* is called from a context where the compiler can know the current flavor (in other words, some constraints on what **self** can be), the optimizer on *function-spec* turns into a call to the **:defun-method** internal function, generating inline code to pass the correct environment.

Also, because of the optimizer, **defun-method** acts like a subst in that better code is generated if the **defun-method** is defined in a file earlier than where it is used. This is contrary to most of the uses of **declare-flavor-instance-variables**, which often define the function after it is used. You should reorder such functions if you want to optimize performance. However, **defun-methods** are not much faster than message-passing, even when the optimized version of the call is being used.

Note that it is now faster to send a computed message than it is to call a computed function that is a **defun-method**! It is slower to use **funcall** to call the function being defined with a **defun-method** than it is to send a message in which you have to have a form that computes the name of the message at run time.

2.3 -*- Line now called the attribute line

The name of the -*- line, which appears at the beginning of files, has been changed from *file property list* (see the *Lisp Machine Manual*, section 21.9.2) to *attribute list* (to avoid confusing it with the *property lists* that the Lisp Machine file system provides for files).

The term attribute list applies not only to the -*- line in character files, but also to an analogous data structure in compiled files. For example, in both cases the attribute list tells **load** what package to load the file into.

The most important result of this change is that two Zmacs commands have new names:

Meta-X Update Mode Line	is now called	Meta-X Update Attribute List
Meta-X Reparse Mode Line	is now called	Meta-X Reparse Attribute List

For consistency, the following two functions have been renamed:

fs:file-read-property-list	is now called	**fs:read-attribute-list**
fs:file-property-bindings	is now called	**fs:file-attribute-bindings**

The old function names continue to work, but the compiler warns you that they are obsolete.

2.4 Changes in Zmacs commands

In addition to the Meta-X Update Mode Line and Meta-X Reparse Mode Line commands mentioned above, a few other incompatible changes to Zmacs have been made.

The command Meta-X List Multiple Callers has been renamed to Meta-X Multiple List Callers, for consistency with other commands. The command Meta-X List Functions has been renamed to Meta-X List Definitions, which is a more accurate description of what it does.

Meta-X Copy Text File and Meta-X Copy Binary File have been replaced by Meta-X Copy File, which determines whether the source file is a character file or a binary file and copies the file appropriately. Different file systems sometimes use different character sets, and if the file is a character file, character translations have to be done. For example, return characters have to be converted into a carriage return and a line feed. (See the *Lisp Machine Manual,* section 21.1 for details.) If it makes a mistake about what kind of copying to use (sometimes there is no way to figure it out), you can control the kind of copying by giving a numeric argument to Meta-X Copy File. The built-in documentation (type HELP D) gives the values of the meaningful numeric arguments.

The command Meta-X Set Package has been changed. Before System 210, Meta-X Set Package changed the current package and the package associated with the file, but did not affect the attribute list of the buffer. As a result, if you did a Meta-X Set Package to a new package and subsequently read the file into the editor, it would still be associated with the old package because the attribute list had not been updated.

As of System 210, Meta-X Set Package always changes the current package, and then it asks you whether you want the file's associated package to be changed as well. If you type NO, nothing else is changed. If you type YES, the package associated with the file is updated, and the attribute list of the file is updated.

For those users who feel unnecessarily burdened by this question and are not worried about the possibility of the confusion mentioned above, a new variable is provided to control the new behavior. You can set this variable in your INIT file if you want.

zwei:*set-package-update-attribute-list* *Variable*

Controls the behavior of the Meta-X Set Package command. If its value is **:ask** (the default), Meta-X Set Package asks you whether to update the package of the file as well. If its value is **nil**, Meta-X Set Package never updates the package of the file. If its value is **t**, Meta-X Set Package always updates the package of the file.

2.5 "Class" feature and <-

The old "class" feature, which was never documented and is considered to have been made obsolete by Flavors, has been removed. Many functions and special forms associated with classes have also been removed, and editor commands associated with message-passing now act only on flavors. Unless you have old software that uses classes, this change should not affect your programs.

One important incompatibility for some users is that the function **<-** has been eliminated and should be replaced by the new function **send** (see section 6.4).

See section 3.3 for a list of symbols that have been removed from the global package.

2.6 Compiler warnings work differently

A new facility to handle compiler warnings is fully documented in section 5.5. The main difference from System 78 is that the Compiler Warnings buffer in Zmacs no longer exists by default. If you want to deal with compiler warnings as you used to in System 78, use the Meta-X Compiler Warnings command in Zmacs; this creates the Compiler Warnings buffer if it doesn't exist, and puts all of the outstanding warnings into it. If you want to use the new commands, read section 5.5.

A minor change in the textual format of comments has also been made.

2.7 Only one process can use the compiler

Only one process can run the compiler at a time. If a second process tries to enter the compiler while it is busy, a notification is printed and the second process waits with status-line state "Compiler". If the first process is not going to get out of the compiler (perhaps it is stopped with an error), you must abort it for the second process to proceed. This was announced in System 78, but now it works. (In earlier systems, if two processes tried to run the compiler at the same time, strange errors would result.)

2.8 Abort hostat with c-Abort

In earlier systems, the host status report produced by **hostat** or `TERMINAL H` would be aborted if you typed any character while it was printing. It has been changed to be like everything else: if you want to abort it while it is running, you must use `c-ABORT`.

2.9 Changed Zmail option

The "Require Subjects on Outgoing Messages" option to Zmail works somewhat differently now. This option appears as one of the lines in the big Choose Variable Values window in the Zmail Profile mode. It has four possible values:

No
: Zmail does not do anything special. This is the default.

Yes
: Zmail insists that you attach a subject line to some or any of the messages that you send. It encourages you to give a subject, by putting an empty "Subject:" line in the "Headers" window. If the message is a reply, the subject line is derived from the subject of the original message, as usual. If you try to send the message without providing a subject, Zmail prompts you for a subject in the minibuffer.

On bug reports
: This is the same as "Yes" except that it applies only to messages that are bug reports (that is, messages that you send by clicking middle on Mail, or clicking right on Mail and then clicking on Bug). Other mail is not affected.

Initial but not required
: Zmail reminds you to put a subject line in your message, by putting a "Subject:" line in the "Headers" window, but it does not prompt if you try to send the message without a subject.

2.10 Interactive messages

When an interactive message arrives for you from the network, a small window pops up near the top of the screen to show you the message, just as in System 78. However, the allowed responses to the "Reply?" prompt have changed slightly. You can type R to reply or N to do nothing; these are like the old Y and N answers. You can also type C to enter the Converse window. These changes are part of the new Converse facility, described in section 5.4. If you don't want to use Converse, type R and N to reply to messages.

2.11 union and intersection

The functions **union** and **intersection** are now nondestructive - they do not clobber their arguments. New functions **nunion** and **nintersection** now exist; these are destructive versions and can reuse cons cells from any list passed to them.

union &rest *lists*
: Takes any number of lists which represent sets and creates and returns a new list that represents the union of all the sets it is given. **(union)** returns **nil**.

intersection &rest *lists*
: Takes any number of lists which represent sets and creates and returns a new list that represents the intersection of all the sets it is given. **(intersection)** returns **nil**.

nunion &rest *lists*

Takes any number of lists which represent sets and returns a new list that represents the union of all the sets it is given, by destroying any of the lists passed as arguments and reusing the conses. **(nunion)** returns **nil**.

nintersection &rest *lists*

Takes any number of lists which represent sets and returns a new list that represents the intersection of all the sets it is given, by destroying any of the lists passed as arguments and reusing the conses. **(nintersection)** returns **nil**.

2.12 process-run-function

The function **process-run-function** has been changed slightly to do what the function **process-run-temporary-function** used to do. (The latter still exists, but is obsolete and will go away eventually.) This means that the process created by **process-run-function** is killed if the machine is warm-booted. (**process-run-function** has some new features, described in section 7.4.)

2.13 load-patches loads the site files

When you do **(load-patches)**, any changed site-specific information for your site is loaded as well. For example, if your host table has been updated, **(load-patches)** loads the new host table. If you call **load-patches** to load the patches for a specific system (rather than for all systems), **load-patches** doesn't update the site information; if you want to load patches for all systems but not update the site information, you can use the new **:nosite** option to **load-patches**.

2.14 Tree Edit Any

The Tree Edit Any command in the File System Editor now prompts you for a wildcard pathname instead of a directory pathname. This means that if you want to list the directory >foo, instead of typing `>foo` you should type `>foo>*.*.*`, a wildcard pathname that matches all of the files in the >foo directory. (In fact, you can generally just type `>foo>`, because the name, type, and version usually default to *.) When you are asked for the wildcard pathname, the defaults are now displayed for you.

This makes Tree Edit Any compatible with all the other directory listers, such as the Display Directory and Dired commands in Zmacs. It also allows you to list part of a directory by using a wildcard pathname to specify which files should be listed. Wildcard pathnames are discussed in section 7.11.

3. CHANGES TO LESS FREQUENTLY USED FEATURES

The changes in this section might break programs that work with advanced subsystems and use advanced features.

3.1 Incompatible compile driver change

The compile driver has been changed incompatibly in several ways. The affected software is mostly internal and should not affect most users. However, an advanced tool that interfaces to the compiler, or uses the functions in Zmacs that do compilation or evaluation of sections of buffers (in particular the function **zwei:eval-print**), might need to be changed. None of the software that was changed is currently documented, so for details of the change, see the source code.

3.2 Temporary si:reset-temporary-area

Some programs use the dangerous **si:reset-temporary-area** feature to deallocate all Lisp objects stored in a given area. As before, use of this technique is not recommended, since gross system failure can result if any outstanding references to objects in the area exist. If you are not now using temporary areas, you can ignore the following change.

Those programs that use the feature must now declare any areas that are to be mistreated this way. When you create a temporary area with **make-area,** you must give the **:gc** keyword and supply the value **:temporary**. (This also marks the area as **:static**; all temporary areas are considered static by the garbage collector.) **sys:reset-temporary-area** now signals an error if its argument has not been declared temporary.

3.3 Symbols removed from global

The following symbols have been removed from the global package:

```
%method-class          defclass                    process-error-stop-process
<-                     defmethod-instance          process-plist
<-as                   defstructclass              put-on-alternating-list
<<--                   endf                        remmethod
area-number            fixnum-class                subclass-of-class-symbol-p
array-class            flonum-class                subclass-of-classp
art-q-list-array       get-from-alternating-list   subinstance-of-class-symbol-p
begf                   locative-pointer            subinstance-of-classp
class                  map-class-hierarchy         symbol-class
class-class            number-class                tail
class-method-symbol    object-class                trap-enable
class-symbol           pair                        unwind-protect-tag
class-symbolp          pkg-load-map                unwind-protect-value
cons-class
```

3.4 Obsolete tv:sheet- functions

Various internal reorganizations were made within the window system. The undocumented **tv:sheet-** functions are now obsolete; most still exist, but the compiler warns you that they are obsolete if it sees any calls to them. If you were using them, use message-passing instead.

3.5 chaos:shout removed

The function **chaos:shout**, described in the System 78 Release Notes, has been removed. It was useful for sending announcements to all users at a site; but because it used the interactive message system, it elicited a reply from each recipient, which was inappropriate. The new function **chaos:notify-all-lms**, described in section 6.3, replaces it.

3.6 Reorganized system source files

Some of the files that hold the source code of the system have been reorganized, particularly those containing language primitives. This should not affect any programs or break anything, but it is mentioned here just in case.

3.7 Debugger now uses error-output

The Debugger now does its input and output using the stream which is the value of **error-output** (or what **error-output** is indirected to if it is a synonym stream) instead of using **terminal-io**. The default value of **error-output** is a synonym stream pointing to **terminal-io**; hence, this change makes a difference only if you change **error-output**. Debugger problems caused by **query-io** being bound to the wrong thing are also fixed.

3.8 %p-store-contents subprimitive changed

A bug in **%p-store-contents**, which caused it to follow invisible pointers, has been fixed. This is mentioned here because it is a low-level change that might break something if you have a program that uses it. (In fact, two functions in the system depended on the buggy behavior and have been fixed; other functions in the system did not work because of the buggy behavior.)

3.9 Microassembler not loaded by default

The microassembler is not loaded into the default system. The main reason is that it has always been necessary to do a **(make-system 'ua)** before running it, in order to get the latest **qcom** and so on. Also, it is rarely used and increases the size of world loads, which is especially inconvenient at sites with T-80 disks.

3.10 Automatic log out of idle file-server jobs

When you use some host on the Chaosnet to access files, a file-server job is created on that host. This job waits for commands from your Lisp Machine, which it processes. However, the job uses some resources on the host, even when it is idle. To avoid wasting resources, the Lisp Machine automatically kills the file-server job after it has been idle for 30 minutes.

If you try to use the host again after the server has been killed, the system automatically creates a new server. You can almost ignore the fact that the server is killed. The main way it affects you is that if the server in question requires a password, you are asked for the password again, and you have to type it in again.

Data connections are also destroyed automatically. This is invisible to the user; it just saves resources. The first one goes after ten idle minutes, and additional ones go after one idle minute.

4. NEW COMMANDS

New commands include TERMINAL commands, Zmacs commands, and Debugger commands. This section discusses changes to the user-interface of the system.

4.1 New Peek facilities

To switch Peek from one mode to another, you used to type a single character. This still works. However, Peek now has a menu at the top, with one item for each mode. The item for the currently selected mode is highlighted in reverse video. If you click on one of the items with the mouse, Peek switches to that mode. The Help message is now a Peek mode, instead of being printed specially, and Peek starts out in the Help mode.

The Z command has been changed to use units of seconds, instead of units of 60ths of a second.

Servers is a new mode in the menu and and is available on the S key. It shows a display of all network servers running on the machine and gives information about what each server is doing.

The Processes mode, available on the P key, now shows more information about processes, including the priorities, quanta, percentage of the processor being used, and idle time, for each process. The meanings of the priority numbers are discussed in section 8.4.

4.2 Name changes for process wait states

The names of process wait states are the things you see in the status line (at the bottom of the screen) such as TYI and NETO; you also see them in Peek when you look at the display of processes. They are intended to explain the state of a process: whether it is running, stopped, or waiting for something. If the process is waiting, this name tells you what it is waiting for. They've been changed to try to be more consistent in their spelling and capitalization; for example, NETO has been changed to "Net Out". Since these names are for user information only (programs don't look at them), this change should not cause programs any problems.

4.3 Package declarations in the attribute list of a file

It is now possible to put the package-declaration of a package into the attribute list of a file. In previous releases, it was necessary to construct a package declaration for a package; the declaration for FOO would reside in its own little file, and then each file to be loaded into package FOO would say **package: FOO** in its attribute list. As of System 210, the contents of the package declaration can now reside in the attribute lists of the files themselves. To do so, the **package** attribute should have the syntax of a Lisp list. Example:

```
-*- Mode: Lisp; Package: (foo global 1000); Base: 8 -*-
```

The first element of the list is the name of the package; the second is the name of the superior package (this is almost always global); the third is the initial estimate of the number of symbols in the package (using 1000 is safe; it grows by itself if necessary, so this number is not critical). Options such as **shadow** and **extern** can be included in the list as subsequent items; the syntax of this list is analogous to that of the **package-declare** special form, except that the obsolete **file alist** is omitted.

If several files get loaded into the same package, they should have the same **package** attribute. If the declaration of the package makes much use of special options, such as **shadow** and **extern,** it might be more convenient to do things the old way, putting the package declaration into its own file, to prevent accidents in which changes to the package declaration aren't faithfully duplicated in all of the source files. Furthermore, if you already have a little file to contain a system declaration, then that's a perfectly good place to put the package declaration as well; whether you want to use the new syntax is a matter of personal preference. The new syntax is primarily useful when you have a program small enough that it fits in one file, thereby eliminating a system declaration.

This new feature is *not* backward-compatible. Files using this new syntax cannot be loaded into systems that predate System 210. Furthermore, when a file using the new syntax is compiled, the attribute list of the compiled file also uses the new syntax; such compiled files cannot be loaded into systems that predate System 210.

4.4 Faster disk-restore

There is a new experimental feature that makes **disk-restore** faster. It has a drawback: paging performance might be degraded for a while as you start to use the machine after it has been **disk-restore**d in this way. The effects of the tradeoff are unclear; the feature is being offered experimentally to determine whether people like it.

When you use **disk-restore** in the normal way, the entire contents of the **lod***n* partition on the disk are copied into the **page** partition in large blocks. This copying takes most of the time in **disk-restore**. **disk-restore** now takes a second argument, called **full-restore-p,** whose value is **t** by default; if you give **nil** explicitly as the second argument, the band is restored in the new way. Instead of copying the **lod***n* partition, the virtual memory system gets set up in such a way that the first time a page is referred to, it is read in from the saved **lod***n* partition directly; when the page is paged out, it is put into the **page** partition as usual, and any new pages are created in the **page** partition. The reason that this can hurt performance is that page faults satisfied from the **lod***n* partition cause longer disk seeks, which take more time. As a result, the first time you try to use any particular body of software such as the editor or the compiler, the paging activity needed to page in that body takes longer.

If you use **disk-restore,** try giving it a second argument of **nil,** and see whether you think that the overall change in performance is an improvement or a drawback. If you have any comments, please let us know.

4.5 New Zmacs commands: Start Patch, Abort Patch, Print Buffer

A new Zmacs command is useful if you change your mind in the middle of creating a patch to a patchable system. If you have given the Meta-X Add Patch command (once or many times) but have not yet done Meta-X Finish Patch to install the patch, and you decide that you don't want to make the patch after all, use the new Meta-X Abort Patch command. This deallocates the minor version number that Meta-X Add Patch allocated, and it tells Zmacs that you are no longer "in the middle of making a patch" (so that the next time you do Meta-X Add Patch, Zmacs starts a new patch instead of appending to the one in progress).

Another new command, Meta-X Start Patch, starts a new patch but does not move any Lisp forms into the patch file. Usually, you don't have to use this command, since Meta-X Add Patch starts up a new patch if there isn't one in progress. However, sometimes you want to create a patch file that does not contain any Lisp code from another file; the patch file might contain only an ad-hoc form to be evaluated, rather than any redefinitions. If you want to create a fresh patch file and type some forms into it, you can use Meta-X Start Patch, select the patch buffer, and start editing it. You can do Meta-X Add Patch later for the same patch, if you want to.

The command Meta-X Print Buffer has been added. It is just like Meta-X Print File except that it prints out the contents of the buffer instead of the contents of a file.

4.6 Shift keys replace double-clicking

When you click a mouse button while holding down any of the keys labelled SHIFT, CONTROL, or HYPER, your click is interpreted as a double-click. Furthermore, if you set **tv:mouse-double-click-time** to **nil**, double-clicking in the usual way is disabled, and the system no longer waits after you click to see whether you are going to click again. This can improve mouse-click response time.

This feature is under control of the following variable. You can set this variable to **nil** if you want to turn off the new behavior (use **login-setq** to set it in your INIT file).

tv:*mouse-incrementing-keystates* *Variable*

The value is a list of the names of those shifting keys which, if held down, make a single-click of the mouse into a double-click. Its default value is **(:shift :control :hyper)**.

4.7 New Debugger command and functions

Meta-I (for "Instance") is a new Debugger command which helps you examine the values of instance variables in the stack group being debugged. The command prompts you for the name of an instance variable, and it prints out the value of that instance variable, inside the instance that is the value of **self** in the environment of the current frame.

Several new functions can be used inside the Debugger. The Debugger's command loop lets you type in Lisp forms, which it reads, evaluates, and prints. When you are typing these forms, you can use the following functions to examine or modify the arguments, locals, function object, and values being returned in the current frame.

eh-arg *arg*

Returns the value of argument *arg* in the current stack frame. **(setf (eh-arg** *n***) x)** sets the value of the argument *arg* in the current frame to the value of **x**. *arg* can be the number of the argument (for example, 0 to specify argument number zero) or the name of the argument. This function can be called only from the read-eval-print loop of the Debugger.

eh-loc *loc*

Returns the value of the local variable *loc* in the current stack frame. **(setf (eh-loc** *n***) x)** sets the value of the local variable *loc* in the current frame to the value of **x**. *loc* can be the number of the local variable (for example, 0 to specify local variable number zero) or the name of the local variable. This function can be called only from the read-eval-print loop of the Debugger.

eh-fun

Returns the function object of the current stack frame. **(setf (eh-fun) x)** sets the function object of the current frame to the value of **x**. This function can be called only from the read-eval-print loop of the Debugger.

eh-val &optional *n*

Returns the value of the *n*th value to be returned from the current stack frame. **(setf (eh-val** *n***) x)** sets the value of the *n*th value to be returned from the current frame to the value of **x**. *n* must be a fixnum, since values don't have names. **(eh-val)** returns a list of all the values to be returned by this frame; it is invalid to **setf** this form. This function can be called only from the read-eval-print loop of the Debugger.

The Debugger sets the following two variables:

eh-sg *Variable*

Inside the read-eval-print loop of the Debugger, the value of **eh-sg** is the stack group currently being debugged by the Debugger.

eh-frame *Variable*

Inside the read-eval-print loop of the Debugger, the value of **eh-frame** is the location of the current frame (expressed as a fixnum offset, into the control stack of the stack group being debugged, of the base word of the current frame).

4.8 Two new Dired features

The first new feature is a command called Describe Attribute List. It is assigned to the comma key (c-,) and is available on the pop-up menu that you get when you click right in Dired. This command prints out the contents of the attribute list of the current file (the one where the cursor is). It works for character files and compiled files.

The second new feature occurs when you type Q to finish up Dired. At this point, you are shown the files that will be deleted and are asked for confirmation. You can now type E, which will delete *and* expunge files. This is meaningful for file systems in which there is "undeletion", such as TOPS-20, TENEX, and the Lisp Machine file system. The new command is useful if you use Dired to free up disk space, since the disk space is not deallocated until the directory is expunged.

4.9 New File System Maintenance program commands

The new Server Shutdown command lets you shut down a file server cleanly. You would run this command only on a Lisp Machine that was acting as a file server for other users. Clicking left on Server Shutdown means that you plan to shut down the file system soon. You are asked for a short message to be sent to people using the file server, which you can use to explain why it is being shut down and when it will return. You are also asked when you want the shutdown to take place; the default is five minutes. All of the users of the file system are sent periodic messages warning them that the server is going to be shut down. Finally, when the time comes, all Chaosnet servers on the machine are closed, and creation of new servers is disabled. At this point servers are shut down, and the machine can be cold-booted or whatever. While the shutdown is "in progress" (the messages are being sent), you can cancel the shutdown by clicking right or reschedule it by clicking middle on Server Shutdown.

This command shuts down the network server; it does not affect the local operation of the file system itself. It shuts down all servers, not just file servers, since anything that requires the file servers to be shut down also requires that all servers be shut down.

The new Server Errors command prints out all of the error messages associated with errors encountered by the file server. When such errors happen, you get the message:

    [File Server got an error]

The error message is recorded; this command lets you see it.

The Print Loaded Band command has been changed to Print Herald and now prints out the herald message.

The Reload command has been removed from the Maintenance menu. The Retrieve command is now called Reload/Retrieve, and this new command handles both reloading (restoring the entire file system from tape) and retrieving (getting back particular files from backup tape).

Consolidated dumps have been implemented. The backup menu offers a *consolidation date*, which must be a date in the past (defaulting to one week ago). A consolidated dump dumps all files that have been created or modified since the consolidation date. It is fast, like an incremental dump, and allows recycling any incremental dump tapes that were made in between the consolidation date and the present. To do a consolidated dump, click on either Incremental Dump or Complete Dump, which pops up a Choose Variable Values window; select Consolidated in the first line of the window, and set the Consolidate from: item to the consolidation date. The Consolidate from: item is not used for incremental or complete dumps.

4.10 New user option controls pathname defaulting

fs:*always-merge-type-and-version* *Variable*

The default algorithm for merging does *not* take the type and version from the default pathname if a name is specified. For example, typing `foo` while the default pathname is bar.lisp results in a pathname of foo, *not* foo.lisp. If you set this variable to **t**, the type and version in the resulting pathname are taken from the defaults; in the above example, foo.lisp would be the result. The default is **nil**, which preserves the old behavior. Because different people seem to like it different ways, it is an option that you can use **login-setq** to set in your INIT file.

4.11 TERMINAL B

The new command `TERMINAL B` buries the currently selected window, if any - that is, it moves it underneath all other windows. This usually brings some other window up to the top, which is automatically selected.

4.12 TERMINAL 0 H

The new `TERMINAL 0 H` command lets you ask for the status of specific hosts. `TERMINAL H` queries every host in the host table, whereas `TERMINAL 0 H` queries only those hosts that you specify.

5. MAJOR NEW FACILITIES

New facilities include Lambda macros, the world-load compressor, VAX/VMS support, the Converse system, and a new system for handling compiler warnings.

5.1 Lambda macros

Lambda macros are similar to regular Lisp macros, except that regular Lisp macros replace, and expand into, Lisp forms, whereas lambda macros replace, and expand into, Lisp functions. They are an advanced feature, used only for certain special language extensions or embedded programming systems.

To understand what lambda macros do, consider how regular Lisp macros work. When the evaluator is given a Lisp form to evaluate, it inspects the car of the form to figure out what to do. If the car is the name of a function, the function is called. But if the car is the name of a macro, the macro is expanded, and the result of the expansion is considered to be a Lisp form and is evaluated. Lambda macros work analogously, but in a different situation. When the evaluator finds that the car of a form is a list, it looks at the car of this list to figure out what to do. If this car is the symbol **lambda,** the list is an ordinary function, and it is applied to its arguments. But if this car is the name of a lambda macro, the lambda macro is expanded, and the result of the expansion is considered to be a Lisp function and is applied to the arguments.

Like regular macros, lambda macros are named by symbols and have a body, which is a function of one argument. To expand the lambda macro, the evaluator applies this body to the entire lambda macro function (the list whose car is the name of the lambda macro), and expects the body to return another function as its value.

Several special forms are provided for dealing with lambda macros. The primitive for defining a new lambda macro is **lambda-macro**; it is analogous with the **macro** special form. For convenience, **deflambda-macro** and **deflambda-macro-displace** are defined; these work like **defmacro** to provide easy parsing of the function into its component parts. The special form **deffunction** creates a new Lisp function that uses a named lambda macro instead of **lambda** in its definition.

lambda-macro *name lambda-list* &body *body* *Special Form*

Analogously with **macro,** defines a lambda macro to be called *name. lambda-list* should be a list of one variable, which is bound to the function being expanded. The lambda macro must return a function. Example:

```
(lambda-macro ilisp (x)
  '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

This defines a lambda macro called **ilisp**. After it has been defined, the following list is a valid Lisp function:

```
(ilisp (x y z) (list x y z))
```

The above function takes three arguments and returns a list of them, but all of the arguments are optional and any extra arguments are ignored. (This shows how to make functions that imitate Interlisp functions, in which all arguments are always optional and extra arguments are always ignored.) So, for example,

```
(funcall #'(ilisp (x y z) (list x y z)) 1 2)  =>  (1 2 nil)
```

deflambda-macro *Special Form*

deflambda-macro is like **defmacro,** but defines a lambda macro instead of a normal macro.

deflambda-macro-displace *Special Form*

deflambda-macro-displace is like **defmacro-displace,** but defines a displacing lambda macro instead of a displacing normal macro.

deffunction *function-spec lambda-macro-name lambda-list* &body *body* *Special Form*

Defines a function using an arbitrary lambda macro in place of **lambda**. A **deffunction** form is like a **defun** form, except that the the function spec is immediately followed by the name of the lambda macro to be used. **deffunction** expands the lambda macro immediately, so the lambda macro must already be defined before **deffunction** is used. For example, suppose the **ilisp** lambda macro were defined, as in the example above. Then the following example would define a function called **new-list,** that would use the lambda macro called **ilisp**:

```
(deffunction new-list ilisp (x y z)
  (list x y z))
```

new-list's arguments are optional, and any extra arguments are ignored. Examples:

```
(new-list 1 2) => (1 2 nil)
(new-list 1 2 3 4) -> (1 2 3)
```

Lambda macros can be accessed with the (**:lambda-macro** *name*) function spec.

5.2 World-load compressor

There is a new program called **compress,** for compressing world loads. To *compress* a world load means to free all the *garbage* (storage occupied by objects that can no longer be referred to) and to remove the gaps left behind so that all of the Lisp objects are stored contiguously.

If you have a large software system and you want to save it away in a disk partition (band), you start with a fresh Lisp image, load in your program, and save the environment with **disk-save**. However, loading in a large system can create some garbage, particularly if a lot of load-time computation is done. All of this garbage is saved away with **disk-save**. By using the compressor, you can create a smaller saved world load, without the garbage. The main advantage of making a smaller world load is that it can fit into a smaller disk partition, which is particularly important if some machines at your site have small (T-80) disks. The compressor is useful only if there is any substantial amount of garbage to be removed from the world load. World loads supplied with new software releases have already been compressed.

To run the compressor, use a machine with a disk larger than a T-80. You need one large **lod***n* partition (at least 35,000 blocks long), and a second, normal-sized partition to hold the result. If you don't have any partitions that large, use **si:edit-disk-label** to create one. Then do the following steps.

1. Establish the environment.
   First, you must establish the environment that you want to compress. If you want to compress a world load that is already stored in a band on the disk, restore that band by cold-booting or by using **disk-restore**. If you want to create an environment to be compressed by loading some files, then load them into a fresh Lisp world.
2. Garbage collect.
   Run the garbage collector on the Lisp world, with **(si:full-gc t)**. This should take about 20 minutes.
3. Save this in the large partition.
   Use **(disk-save** *partition***)**, where *partition* is the name of the large partition, to save the resulting garbage-collected environment into the large disk partition mentioned above.
4. Load **compress**.
   Use **(make-system 'compress)** to load **compress** into the Lisp world. It is not loaded by default, since it is rarely used.
5. Compress the large partition.
   Run the compressor itself, with **(si:compress-band 0** *part1* **0** *part2***)**, where *part1* is the name of the large partition, and *part2* is the name of another, normal-sized partition that holds the resulting compressed world load. The compressor copies from *part1* to *part2*, copying only the useful storage, and relocating all pointers. This takes about an hour to run because it has to examine every object in the Lisp world in order to fix up the pointers.
6. You are done.
   Boot the compressed band (*part2*) by cold-booting or with **disk-restore**, to test it.

5.3 VAX/VMS now a supported file system

VAX/VMS is now supported as a file system. If your site has a VAX running the VMS operating system, it can be connected to the Chaosnet and its file system can be used directly from the Lisp Machine.

Since file extensions in the VAX/VMS file system are restricted to three characters, the representation of standard pathname types, such as LISP and TEXT, are stored in abbreviated forms in VAX/VMS pathnames. As was mentioned in the System 78 Release Notes, such abbreviations are also used in UNIX files, because of the 14-character limitation of filenames in UNIX. Following is a complete list of the standard pathname types and their abbreviations on both VAX/VMS and UNIX.

Type	VAX/VMS	UNIX
LISP	LSP	l
TEXT	TXT	tx
MIDAS	MID	md
QFASL	QFS	qf
PRESS	PRS	pr
(PDIR)	PDR	pd
QWABL	QWB	qw
PATCH-DIRECTORY	PDR	pd
BABYL	BAB	bb
XMAIL	XML	xm
MAIL	MAI	ma
INIT	INI	in
UNFASL	UNF	uf
OUTPUT	OUT	ot
ULOAD	ULD	ul
CWARNS	CWN	cw

If you create a pathname for a VAX/VMS host whose name is FOO and whose type is QFASL, the printed representation of the pathname is FOO.QFS, which is what the file is named in the VAX/VMS file system. This translation is performed in both directions; that is, if you parse a pathname that looks like FOO.QFS, the pathname object has name FOO and type QFASL. Fortunately, since VAX/VMS pathnames cannot have extensions longer than three characters, and all system types are longer than three characters, ambiguity such as is possible in UNIX is avoided. (This problem was mentioned in the System 78 Release Notes; it is also discussed in section 8.6.)

5.4 Converse: a new program for handling interactive messages

The Converse interactive message editor is run by a new window with its own process. Converse keeps track of all of the messages that you have received or sent. The Converse window shows all of the messages that have been sent or received since the machine was cold-booted. Messages are organized into *conversations*, based on the identity of the other party in the conversation. Conversations are separated from each other by a thick black line. Within each conversation are all messages, outgoing and incoming, arranged in chronological order, and separated by thin black lines. You can use Converse to look at conversations, send messages, and receive messages. Converse is built on the Zwei editor, so you can edit your message as you type it in, or pick up and move around text between one message and another, or between messages, files, and pieces of mail.

To send a message to someone, you can use the **qsend** function (as in System 78), or you can select the Converse window with SYSTEM C or by typing (qsend), and use Converse to send the message. To send a message as part of an existing conversation, find that conversation in Converse (you can use the regular editor commands to scroll around in the Converse window), and fill in the blank message at the end, finishing with END to send the message and exit Converse (or c-END to stay in Converse). If you want to start a new conversation, go to the top of the Converse window and fill in the blank message, starting with the "To:" line to specify the new recipient.

When a new message arrives, a small window pops up near the top of the screen. You can respond R to type in a reply, N to do nothing, or C to enter Converse. Converse has several advantages: you can look over the previous messages in the conversation, and you get full editing power to help you construct a reply. You can customize what happens when a new message arrives by using the variable **zwei:*converse-mode***, described below.

Converse remembers all messages that you send or receive, even if you didn't use the Converse window to send them or reply to them.

Converse has several commands for moving around within conversations, writing conversations out to files, and so on. Type HELP in Converse for a summary of these commands. You can enter Converse by typing SYSTEM C, by evaluating **(qsend)**, by using the Select command from the mouse, or by answering C when a message arrives in the pop-up window.

qsend &optional *destination message*
Sends interactive messages to users on other machines on the network. *message* should be a string. If it is omitted, **qsend** asks you to type in a message. You should type in the contents of your message and type END when you're done. *destination* is normally a string of the form "*name@host*", to specify the recipient's name and host. If you omit the *@host* part and just give a name, **qsend** looks at all of the Lisp Machines at your site to find any that *name* is logged into; if the user is logged into one Lisp Machine, it is used as the host; if more than one, **qsend** asks you which one you mean. If you leave out *destination* altogether, doing just **(qsend)**, Converse is selected as if you had typed SYSTEM C.

print-sends &optional *(stream* **standard-output***)*
Prints out all messages you have received (but not messages you have sent), in forward chronological order, to *stream*. Converse is more useful for looking at your messages, but this function predates Converse and is retained for compatibility.

The following three variables allow you to customize Converse's behavior. You can set them in your INIT file using **login-setq**.

zwei:*converse-mode* *Variable*
Controls what happens when an interactive message arrives over the network at your machine. It should have one of the following values:

:pop-up(This is the default.) A small window pops up at the top of the screen, displaying the message. You are asked to type R (for Reply), N (for Nothing), or C (for Converse). If you type R, you can type in a reply to the message inside the little window. When you type END, this reply is sent back to whomever sent the original message to you, and the little window disappears. If you type N, the little window disappears immediately. If you type C, the Converse window is selected, as if **zwei:*converse-mode*** were **:auto** (see below).

:auto The Converse window is selected. This is the big window that shows you all of your conversations, letting you see everything that has happened, and letting you edit your replies with the full power of the Zwei editor. With this window selected, you can reply to the message that was sent, send new messages, participate in other conversations, and so on. You can exit with END or ABORT (END sends a message and exits; ABORT just exits), or you can select a new window by any of the usual means (such as the TERMINAL or SYSTEM keys).

:notify A notification is printed, telling you that a message arrived and from whom. If you want to see the message, you can enter Converse by typing SYSTEM C or using the **print-sends** function. If you want to reply to the sender, you can enter Converse or use the **qsend** function manually.

:notify-with-message
A notification is printed, which includes the entire contents of the message and the name of the sender. If you want to reply, you can enter Converse or use the **qsend** function manually.

zwei:*converse-append-p* *Variable*
If the value is **nil** (the default), a new message is prepended to its conversation. If the value is not **nil,** a new message is appended to its conversation. **print-sends** is not affected by this variable.

zwei:*converse-beep-count* *Variable*
The value is the number of times to beep when a message arrives. The default value is two. Beeping occurs only if the Converse window is exposed or if the value of **zwei:*converse-mode*** is **:pop-up** or **:auto**. (Otherwise, notification tells you about the message and includes the usual beeping.)

The new Rubout Handler (described in section 5.6) is used while you type in a message to **qsend** and while you type in a reply in **:pop-up** mode. So if you use the default mode (**:pop-up**) and turn on the new Rubout Handler, you get some editing power while replying, although not as much as with full Converse (since the latter uses Zwei).

5.5 New compiler warnings database

In System 78, compiler warnings were directed to a special Zmacs buffer named Compiler Warnings, which was always present. In System 210, warnings are kept in an internal database, and several functions and editor commands are provided that allow you to inspect and manipulate this database in various ways.

The database of compiler warnings is organized by pathname: warnings that were generated during the compilation of a particular file are kept together, and this body of warnings is identified by the generic pathname of the file being compiled. Any warnings that were generated while compiling some function not in any file (for example, by using the **compile** function on some interpreted code) are stored under the pathname **nil**. For each pathname, the database has entries, each of which associates the name of a function (or a flavor) with the warnings generated during its compilation. The database starts out empty when you cold-boot. Whenever you compile a file, buffer, or function, the warnings generated during its compilation are entered into the database. If you recompile a function, the old warnings are removed, and any new warnings are inserted. If you get some warnings, fix the mistakes, and recompile everything, the database becomes empty again.

Warnings are printed out as well as stored in the database. If the value of the special variable **suppress-compiler-warnings** is not **nil,** warnings are not printed, although they are still stored in the database.

The database has a printed representation. **print-compiler-warnings** produces this printed representation from the database, and **compiler:load-compile-warnings** updates the database from a saved printed representation. Following are the details:

print-compiler-warnings &optional *files (stream* **standard-output***)*
Prints out the compiler warnings database. If *files* is **nil** (the default), it prints the entire database. Otherwise, *files* should be a list of generic pathnames, and only the warnings for the specified files are printed. (**nil** can be a member of the list, too, in which case warnings for functions not associated with any file are also printed.) The output is sent to *stream*; you could use this to send the results to a file.

compiler:load-compiler-warnings *file* &optional *(flush-old-warnings* **t***)*
Updates the compiler warnings database. *file* should be the pathname of a file containing the printed representation of the compiler warnings related to the compilation of one or more files. If *flush-old-warnings* is **t** (the default), any existing warnings in the database for the files in question are completely replaced by the warnings in *file*. If *flush-old-warnings* is **nil**, the warnings in *file* are added to those already in the database.

The printed representation of a set of compiler warnings is sometimes stored in a file. You can create such a file using **print-compiler-warnings**, but it is usually created with **make-system** given the **:batch** option. The default type for such files is CWARNS. (See section 5.3 for VAX/VMS and UNIX abbreviations.)

Several Zmacs commands deal with the database.

Meta-X Compiler Warnings
Prints the compiler warnings database into a buffer called Compiler Warnings, creates the buffer if it does not exist already, and switches to that buffer. You can peruse the compiler warnings by scrolling around and doing text searches through them.

Meta-X Edit Compiler Warnings
Prompts you with the name of each file mentioned in the database, allowing you to edit the warnings for that file. It then splits the Zmacs frame into two windows: the upper window displays a warning message, and the lower one displays the source code whose compilation caused the warning. After you have finished editing each function, `c-.` gets you to the next warning: the top window scrolls to show the next warning, and the bottom window displays the function associated with this warning. Successive `c-.`s take you through all of the warning messages for all of the files you specified. When you are done, the last `c-.` puts the frame back into its one-window configuration.

Meta-X Edit File Warnings
Asks you for the name of the file whose warnings you want to edit. You can give either the source file or the compiled file. Only warnings for this file are edited. If the database does not have any entries for the file you specify, the command prompts you for the name of a file that contains the warnings, in case you know that the warnings are stored in another file.

Meta-X Load Compiler Warnings
Prompts you for the name of a file containing the printed representation of some compiler warnings and loads them into the database. (This is like the **compiler:load-compiler-warnings** function.) This command always passes **t** as the *flush-old-warnings* argument; that is, it replaces the old warnings rather than merging with them. The default file type is CWARNS and the default version is **:newest** (the latest version).

The symbols **compiler-warnings-buffer** and **concatenate-compiler-warnings-p** are now obsolete and have been removed from global.

5.6 New experimental Rubout Handler

There is a new experimental Rubout Handler. The Rubout Handler is the software module that lets you type RUBOUT when you are being prompted for a Lisp form or a line of text; you most frequently encounter the Rubout Handler in Lisp Listeners. The existing Rubout Handler is simple, with only a few commands (such as RUBOUT and CLEAR INPUT).

To try out the new Rubout Handler, do **(tv:rh-on)**. It is upward-compatible with the old Rubout Handler, but has many new commands. Most of its commands are analogous to commands in the Zwei editor; for example, c-F and c-B move the cursor forward and backward, respectively, by one character. Type HELP for a complete list of commands.

The Rubout Handler has some new commands that are not present in the editor. The most interesting commands are c-C, m-C, and STATUS. These commands deal with a ring buffer of pieces of text, similar to but distinct from the kill buffer. This new ring is called the input buffer, and it contains a history of the most recent things that you have typed. c-C and m-C work analogously to c-Y and m-Y, but they use this input buffer instead of the kill buffer. Typing STATUS displays the entries of the input buffer. These commands are useful when you want to retype something that you typed before, possibly with some modifications.

The Rubout Handler does not use internal functions of Zwei to do its editing. As a result, it is not completely compatible with the analogous editor commands. The most striking difference is that the commands c-m-F, c-m-B, c-m-K, and c-m-RUBOUT in the Rubout Handler work simply on areas of text delimited by pairs of parentheses; they do not understand any of the subtle aspects of Lisp syntax, and they do not recognize Lisp symbols or numbers as objects. This problem will be fixed in a future release; it is one of the reasons that the new Rubout Handler is experimental.

You might want to put a **(tv:rh-on)** in your INIT file, so that it is turned on whenever you log in. If you think something is wrong with it and you need to turn it off and go back to the old Rubout Handler, you can do **(tv:rh-off)**. (If the Rubout Handler seems so broken that you can't even type Lisp forms, type TERMINAL CALL to get into the "cold load stream", type `(tv:rh-off)`, then type ABORT to get out of the cold load stream, and type TERMINAL CLEAR-SCREEN to fix up the display.)

6. NEW FUNCTIONS

This chapter describes new functions, special forms, flavors, and streams.

6.1 New special form: typecase

typecase *form clauses...* *Special Form*

typecase is a special form for selecting various forms to be evaluated depending on the type of some object. It is something like **select**. A **typecase** form looks like:

```
(typecase form
   (types consequent consequent ...)
   (types consequent consequent ...)
   ...
   )
```

form is evaluated, producing an object. **typecase** examines each clause in sequence. *types* in each clause is either a single type (if it is a symbol) or a list of types. If the object is of that type, or of one of those types, then the consequents are evaluated and the result of the last one is returned. Otherwise, **typecase** moves on to the next clause. As a special case, *types* can be **otherwise**; in this case, the clause is always executed, so this should be used only in the last clause. For an object to be of a given type means that if **typep** is applied to the object and the type, it returns **t**. That is, a type is something meaningful as a second argument to **typep**. Example:

```
(defun tell-about-car (x)
  (typecase (car x)
    (:fixnum "The car is a number.")
    ((:string :symbol) "The car is a name.")
    (otherwise "I don't know.")))

(tell-about-car '(1 a)) => "The car is a number."
(tell-about-car '(a 1)) => "The car is a name."
(tell-about-car '("word" "more")) => "The car is a name."
(tell-about-car '(1.0)) =>
"I don't know."
```

6.2 New function: chaos:host-up

chaos:host-up *host* &optional *timeout*

Asks a host whether or not it is up (alive, functional, responding). If it is up, this function returns t; if not, it returns two values: **nil**, and the error that occurred (usually "Host not responding."). *host* can be a host object or the name of a host; *timeout* is in 60ths of a second and defaults to three seconds. If the host does not respond after this much time, it is assumed to be down.

Note that if this function returns **nil,** it is possible that the host is up but is not connected to the Chaosnet. This function tests whether the Lisp Machine is capable of communicating with the host over the Chaosnet.

6.3 New functions: chaos:notify-all-lms and chaos:notify

chaos:notify-all-lms &optional *message*

Sends a message to all Lisp Machines at your site (based on the value of the list **si:machine-location-alist**). *message* should be a string; if it is not provided, the function prompts for a message. Each recipient receives the message as a notification, rather than as an interactive message.

chaos:notify *host* &optional *message*

Sends a message to the specified host. *host* should be a host (the host name, as a string, or a host object). *message* is a string; if it is not provided, the function prompts for a message. The recipient receives the message as a notification, rather than as an interactive message.

6.4 New functions: send and lexpr-send

send is the new official function to use to send messages to objects. It should be used in the same way that **funcall** has been used up to now.

send *object message-name* &rest *arguments*

Sends the message named *message-name* to the *object. arguments* are the arguments passed.

Currently, **send** does exactly the same thing as **funcall.** However, in a future release, it will be possible to send messages to objects of any data type, and **send** will be changed upward-compatibly to make this work.

Another new function, **lexpr-send,** is to **send** as **lexpr-funcall** is to **funcall.**

lexpr-send *object message-name* &rest *arguments*

Sends the message named *message-name* to the *object. arguments* are the arguments passed, except that the last element of *arguments* should be a list, and all the elements of that list are passed as arguments. Example:

```
(send some-window ':set-edges 10 10 40 40)
```

does the same thing as

```
(setq new-edges '(10 10 40 40))
(lexpr-send some-window ':set-edges new-edges)
```

Note: **send-self** or **lexpr-send-self** do not exist, because the new implementation of Flavors eliminates any particular performance benefit. To send a message to **self,** pass **self** as the first argument to **send.**

6.5 New function: time-increment

A new function, **time-increment,** joins **time-lessp** and **time-difference**. These three functions do a special kind of wrap-around arithmetic to manipulate the values returned by the **time** function. **time-lessp** and **time-difference** are explained in the *Lisp Machine Manual.*

time-increment *time increment*

Adds *increment* to *time* and returns the resulting time value, compensating for wrap-around. *time* should be a value of time, as returned by the **time** function, and *increment* should be an amount of time expressed as a fixnum in units of 60ths of a second.

6.6 New function: uptime

A new function, **uptime,** is a variant of **hostat.**

uptime &rest *hosts*

Queries the specified *hosts,* asking them for their "uptime"; each host responds by saying how long it has been up and running. **uptime** prints out the results. If **uptime** reports that a host is "not responding", either the host is not responding to the network, or it does not support the UPTIME protocol. Note: systems that predate System 210 do not support the UPTIME protocol.

(The Chaosnet protocol works like the TIME protocol, except that the number returned is the uptime of the server expressed in 60ths of a second, instead of the time of day expressed in seconds, and the contact name is UPTIME rather than TIME.)

6.7 New function: load-and-save-patches

load-and-save-patches is a new function that makes it easy for you to load all patches into the Lisp environment and save the result into a band on the disk.

load-and-save-patches

Loads any patches that need to be loaded and any new versions of the site files. Then, if anything has been changed, it prints out the disk label for you and asks where you would like the Lisp environment to be saved. After you tell it, it does a **disk-save**. This function should be called before you log in to avoid putting the contents of your login file into the saved world load. Since the function knows many of the common errors people make when saving bands and what to do about them, it can help you avoid many problems.

A related change is that **load-patches** now returns **t** if any patches were made, and **nil** otherwise.

6.8 New function: store-conditional

store-conditional is a new primitive for constructing interprocess communication tools.

store-conditional *locative old-value new-value*

Takes three arguments: *locative* (which addresses some cell), *old-value* (any Lisp object), and *new-value* (any Lisp object). It checks to see whether the cell contains *old-value*, and, if so, it stores *new-value* into the cell. The test and the set are done as a single atomic operation. **store-conditional** returns **t** if the test succeeded and **nil** if the test failed. You can use **store-conditional** to do arbitrary atomic operations to variables that are shared between processes. For example, to atomically add 3 into a variable **x**:

```
(do ()
    ((store-conditional (locf x) x (+ x 3))))
```

The first argument is a locative so that you can atomically affect any cell in memory; for example, you could atomically add 3 to an element of an array or structure. The first argument can also be a cons, in which case the cell that is the car of the cons is used.

(For a long time a function called **%store-conditional** has worked almost the same way; however, it was a special subprimitive that did not check its data type. The new function, without the **%** in its name, is a normal Lisp function that checks the type of its first argument appropriately.)

6.9 New function: fundefine

fundefine *function-spec*

Removes the definition of *function-spec*, if any. It undoes the effect of the **fdefine** function. For example, if the argument is a symbol, it does the same thing as **fmakunbound**.

6.10 New function: undeletef

undeletef *file* &optional *(error-p* **t***)*

Undeletes the specified file. *file* can be a pathname or a stream which is open to a file. If *error-p* is **t** and an error occurs, it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error message is returned as a string; otherwise **t** is returned. **undeletef** is like **deletef** except that it undeletes the file instead of deleting it. **undeletef** is meaningful only for files in file systems that support undeletion, such as TOPS-20 and the Lisp Machine file system.

6.11 New function: si:flavor-allowed-init-keywords

si:flavor-allowed-init-keywords *flavor-name*

Returns a list of all symbols that are valid init-options for the flavor, sorted alphabetically. *flavor-name* should be the name of a flavor (a symbol). This function is primarily useful for people, rather than programs, to call to get information. You can use this to help remember the name of an init-option or to help write documentation about a particular flavor.

6.12 New function: si:clear-band

si:clear-band *partition* &optional *subset-start subset-n-blocks*

Zeroes a partition of the disk. *partition* specifies a partition on the disk in the manner of the other band functions such as **si:receive-band** - namely, as a string or symbol being the name of the partition, or a number *n* to mean **lod***n*. It writes zeroes over the entire band on the disk. It optionally takes two more arguments; if these are given, only the specified contiguous subset of the band is cleared. The partition's comment field in the disk label is cleared as well. Like all functions that clobber disk partitions, it asks for confirmation first. This function is useful for getting rid of the remnants of an old file system in a file partition; in general it is not needed.

6.13 New function: viewf

viewf *pathname* &optional *(stream* **standard-output***) leader*

Prints the file named by *pathname* onto the *stream*. (The optional third argument is passed as the *leader* argument to **stream-copy-until-eof**.) The name **viewf** is analogous with **deletef, renamef,** and so on. Note: **viewf** should not be used for copying files; its output is not the same as the contents of the file (for example, it does a **:fresh-line** operation on the stream before printing the file).

6.14 New functions to handle intervals of time

New functions read and print time intervals. They convert between strings of the form "3 minutes 23 seconds" and fixnums representing numbers of seconds.

time:print-interval-or-never *interval* &optional *(stream* **standard-output***)*

Prints the representation of *interval* as a time interval onto *stream*. If *interval* is **nil,** it prints "Never". *interval* should be a non-negative fixnum, or **nil**.

time:parse-interval-or-never *string* &optional *start end*

string is the character-string representation of an interval of time. *start* and *end* specify a substring of *string* to be parsed; they default to the beginning and end of *string,* respectively. The function returns a fixnum if *string* represented an interval, or **nil** if *string* represented "never". If *string* is anything else, an error occurs. Examples of acceptable strings:

```
"4 seconds"          "4 secs"             "4 s"
"5 mins 23 secs"     "5 m 23 s"           "23 SECONDS 5 M"
"never"              "not ever"           "no"
""                   "3 yrs 1 week 1 hr 2 mins 1 sec"
```

Note that several abbreviations are understood, the components can be in any order, and case (upper versus lower) is ignored. Also, "months" is not acceptable, since months vary in length. This function accepts anything that **time:print-interval-or-never** produces, and it returns the same fixnum (or **nil**).

time:read-interval-or-never &optional *(stream* **standard-input***)*
Reads a line of input from *stream* (using **readline**) and calls **time:parse-interval-or-never** on the resulting string.

6.15 New function: fill-pointer

fill-pointer *array*
Returns the value of the fill pointer. *array* must have a fill pointer. **fill-pointer** is actually a subst, so it compiles inline instead of as a function call. **setf** can be used on a **fill-pointer** form to set the value of the fill pointer.

Programs access the fill pointer by explicitly asking for the zeroth element of the array leader.

6.16 New function: errorp

errorp *object*
Returns **t** if *object* is an error object, and **nil** if it is not.

In the present system, most functions return a string to indicate that an error has occurred. For example, if you use the **:noerror** keyword in **open**, or the *error-p* argument of **deletef** and related functions, a string is returned if anything went wrong. In the next release, a new facility for handling errors, and a new kind of object called an error object, will be introduced. In that system, **errorp** will discriminate between error objects and all other objects, and you will be able to use it to determine whether or not a function is reporting an error. For now, **errorp** is exactly the same as **stringp**.

6.17 New function: si:full-gc

A new function, **si:full-gc**, garbage-collects the entire Lisp environment. A new initialization list, accessed through the **full-gc** keyword to **add-initialization**, is run by **si:full-gc**.

si:full-gc &optional *gc-static-areas*

The simplest available functional interface to the garbage collector, **si:full-gc** does an immediate, complete, nonincremental garbage collection. When it is over, no further garbage collection happens. If *gc-static-areas* is not **nil**, a set of system areas that are normally static (not subject to garbage reclamation) are treated as if they were dynamic for the duration of this garbage collection. **si:full-gc** runs the forms on the **full-gc** initialization list and then does any garbage collection without multiprocessing (inside a **without-interrupts** form), so the machine essentially "freezes" and does nothing but garbage collection for the duration. This operation takes about 20 minutes; *gc-static-areas* does not seem to take significantly longer.

full-gc is a new system initialization list. You can add forms to it by passing the **full-gc** keyword in the list of keywords that is the third argument of **add-initialization**. The **full-gc** initialization list is run just before a full garbage collection is performed by **si:full-gc**. All forms are executed without multiprocessing, so the evaluation of these forms must not require any use of multiprocessing: they should not go to sleep or do input/output operations that might wait for something. Typical forms on this initialization list reset the temporary area of subsystems and make sure that what is garbage (and about to be collected) has no more pointers to it.

6.18 New functions to deal with closures

copy-closure *closure*

Creates and returns a new closure by copying *closure*, which should be a closure or an entity. **copy-closure** generates new external value cells for each variable in the closure and initializes their contents from the external value cells of *closure*.

closure-variables *closure*

Creates and returns a list of all of the variables in *closure*, which should be a closure or an entity. It returns a copy of the list that was passed as the first argument to **closure** when *closure* was created.

boundp-in-closure *closure symbol*

Returns **t** if *symbol* is bound in the environment of *closure*; that is, it does what **boundp** would do if you restored the value cells known about by *closure*. If *symbol* is not closed over by *closure*, this is just like **boundp**.

makunbound-in-closure *closure symbol*

Makes *symbol* be unbound in the environment of *closure*; that is, it does what **makunbound** would do if you restored the value cells known about by *closure*. If *symbol* is not closed over by *closure*, this is just like **makunbound**.

A note about all of the *xxx*-**in-closure** functions (**SET-**, **symeval-**, **boundp-**, and **makunbound-**): if the variable is not directly closed over, the variable's value cell from the global environment is used. That is, if closure A closes over closure B, *xxx*-**in-closure** of A does not notice any variables closed over by B.

A note about **closure-alist**: if any variable in the closure is unbound, this function signals an error. It has been changed to return the variables in forward order, rather than in reverse order as it used to. Use of this function is not recommended.

6.19 New function: lmfs:grow-file-partition

lmfs:grow-file-partition

Enlarges the size of a file partition to make room for more files. The procedure is as follows. First, use the disk-label editor (**si:edit-disk-label**) to expand the file partition. (You must not change the starting block of the file partition; you should delete the partition [or several] that follows the file partition, and then expand the size of the file partition to use up the new space.) Second, do **(lmfs:grow-file-partition)**.

This function cannot be used to shrink or move a file partition. One way to do those things is to dump the entire file system to backup tape, create and initialize a new file partition, and restore the entire file system from tape.

6.20 New functions: si:describe-partitions and si:describe-partition

Two new functions, **si:describe-partitions** and **si:describe-partition**, print out descriptions of the contents of the partitions on the disk.

si:describe-partitions &optional *(unit* **0***)*

Prints out descriptions of all of the partitions of the disk specified by *unit* to **standard-output**.

si:describe-partition *partition* &optional *(unit* **0***)*

Prints out the description of the specified *partition* of the disk specified by *unit* to **standard-output**. *partition* should be the name of a partition (as a string or symbol, or as a number *n* to indicate **lod***n*).

The description of any partition starts with the line that would be printed by **print-disk-label**, giving the partition's name, address, size, and comment. For partitions containing world loads, the description includes four other data items:

Valid Size

The size of the world load itself. (The partition is generally larger than the world load, and the rest of the partition is unused.)

System Version

The major version number of System in the world load. (For example, for the world load being distributed now, this would be 210.)

Desired Microcode

The version of the microcode with which this world load expects to be booted.

GC Generation Count

The number of times that a either a garbage collector "flip" has happened, or a band compression has happened. This number is generally 0 for uncompressed bands and 2 for compressed bands.

For partitions containing microloads, the description tells how many entries the microload contains for various internal processor memories. The most interesting piece of information that it tells you about microloads is the version number of the microload; this is normally in the disk label comment as well, but the comments can easily be changed to any text string, whereas the description is based on examination of the actual contents of the partition.

Future system releases might print out more information about these and other kinds of partitions.

6.21 New global function: print-herald

The function **print-herald** has been made global (it used to be in the **system-internals** package).

print-herald &optional *(stream* **standard-output***)*

> Prints out the herald message to *stream.* The herald message is what the machine prints when it is cold-booted; it tells you the comment on the disk label, the versions of the systems that are running, the site name, the machine's own host name, and the name of the associated machine.

Note: the function **print-loaded-band,** which printed out the message in pre-System 78 releases, is no longer global.

6.22 New special forms for dealing with variables

value-cell-location on local variables is now obsolete. In the past, the only way to generate a locative pointer to the memory cell associated with a local variable called **a** was with the form **(value-cell-location 'a)**. This is inelegant, since **value-cell-location** is a function that concerns symbols (special variables) in particular, rather than variables in general (see the *Lisp Machine Manual,* section 3.1). This form continues to work, but the compiler issues a warning telling you that it is obsolete. A new special form replaces it:

variable-location *variable* *Special Form*

> Returns a locative pointer to the memory cell that holds the value of the variable. *variable* should be any kind of variable (it is not evaluated): local, special, or instance.

If **a** is a local or instance variable and you use the obsolete **(value-cell-location 'a)** form, the compiler issues a warning and converts it into the proper **variable-location** form. So if you have programs that use this form, they will continue to work. Similarly, the compiler issues warnings for obsolete uses of **boundp** and **makunbound,** and generates code that works.

(value-cell-location 'a) is still a good form when **a** is a special variable. It behaves slightly differently from the form **(variable-location a),** in the case that **a** is a variable "closed over" by some closure (see the *Lisp Machine Manual,* Chapter 11). **value-cell-location** returns a locative pointer to the internal value cell of the symbol (the one that holds the invisible pointer, which is the real value cell of the symbol), whereas **variable-location** returns a locative pointer to the external value cell of the symbol (the one pointed to by the invisible pointer, which holds the actual value of the variable).

You can also use **locf** on variables (this has always been true). **(locf a)** now expands into **(variable-location a)**.

There are two more new special forms:

variable-boundp *variable* *Special Form*

Returns **t** if the variable is bound and **nil** if the variable is not bound. *variable* should be any kind of variable (it is not evaluated): local, special, or instance. Note: local variables are always bound; if *variable* is local, the compiler issues a warning and replaces this form with **t**.

variable-makunbound *variable* *Special Form*

Makes the variable be unbound and returns *variable*. *variable* should be any kind of variable (it is not evaluated): local, special, or instance. Note: since local variables are always bound, they cannot be made unbound; if *variable* is local, the compiler issues a warning.

If **a** is a special variable, **(boundp 'a)** is the same as **(variable-boundp a)**, and **(makunbound 'a)** is the same as **(variable-makunbound a)**.

Note: along with these changes, **let-globally** has been fixed to deal with unbound variables correctly. If **let-globally** is entered and one of the variables is unbound, it is made unbound again when the **let-globally** is exited, whether it is exited normally or abnormally.

6.23 New special forms: do* and do*-named

A new special form, **do***, is like **do** but uses serial binding.

do* *Special Form*

do* is just like **do** except that the variable clauses are evaluated sequentially rather than in parallel. When a **do** starts, all of the initialization forms are evaluated before any of the variables are set to the results; when a **do*** starts, the first initialization form is evaluated, then the first variable is set to the result, then the second initialization form is evaluated, and so on. The stepping forms work analogously.

do*-named *Special Form*

do*-named is just like **do-named** except that the variable clauses are evaluated sequentially, rather than in parallel. See above.

6.24 New flavor: tv:centered-label-mixin

tv:centered-label-mixin *Flavor*

If you mix this flavor with any of the flavors that create labels, the text of the label is centered within the edge of the window, instead of being aligned against the left edge.

Also, labels can now have more than one line of text if they contain return characters.

6.25 New stream function: si:null-stream

The function **si:null-stream** can be used as a dummy stream object. As an input stream, it immediately reports end-of-file; as an output stream, it absorbs and discards arbitrary amounts of output. Note: **si:null-stream** is not a variable; it is defined as a function. Use its definition (or the symbol itself) as a stream, not its value. Examples:

```
(stream-copy-until-eof a 'si:null-stream)
(stream-copy-until-eof a #'si:null-stream)
```

Either of the above two forms reads characters out of the stream that is the value of **a** and throws them away, until **a** reaches the end-of-file.

6.26 New messages: :create-directory and :create-link

Two new messages can be used to create directories and links.

:create-directory &optional *(error-p* **t***)* &rest *options* to **pathname** *Method*

Creates the directory specified by the directory component of the pathname. The name, type, and version of the pathname are ignored. If *error-p* is **nil** and an error occurs, the message returns a string containing the error message instead of signalling an error. The following example creates a directory named bar in the existing directory named >foo on the host named lm6.

```
(send (fs:parse-pathname "lm6:>foo>bar>") ':create-directory)
```

:create-link *target* &rest *options* to **pathname** *Method*

Creates a new link with pathname **self**. The target of the link is *target*, which should be a pathname object for the same host as **self**. The only option is **:error**; if its value is **t** and an error occurs, the message returns a string containing the error message instead of signalling an error. The following example creates a link from lm6:>foo>bar>a.b to lm6:>fred>c.d.

```
(send (fs:parse-pathname "lm6:>foo>bar>a.b")
      ':create-link
      (fs:parse-pathname "lm6:>fred>c.d"))
```

These two messages currently work only in the Lisp Machine file system, used locally or remotely.

7. IMPROVEMENTS TO EXISTING FUNCTIONS

Several existing functions in the system have been enhanced with new features and capabilities.

7.1 New features in fquery

A new value for the **:type** keyword in **fquery**, **:mini-buffer-or-readline,** is like the **:readline** value. The exception is that if **fquery** is called from inside Zwei or Zmail, the line of text is read from the minibuffer instead of from the **query-io** stream. The idea of this feature is to let you write things that work equally well inside Zwei or on their own; if you use this value, you make it easier for your code to be integrated into a Zwei extension.

In the list that is the value of the **:choices** option, a new choice is available. If the last element in the list of choices is the symbol **:any** (instead of being a list, like all other choices), and the user gives some response that is not one of the other choices, instead of complaining and reprompting the user, **fquery** returns what the user typed (a single character or a string, depending on the **:type** option).

A new keyword, **:stream,** has as its value the stream to use for both input and output. The default value is the value of the global variable **query-io**.

7.2 New option to make-array: :initial-value

A new option to **make-array**, **:initial-value,** makes its value the initial value of every element of the array. Example:

```
    (make-array 5 ':type 'art-string ':initial-value #/a)
                        => "aaaaa"
```

7.3 New facility for handling messages to flavor objects

Whoppers is a new facility in Flavors. It is related to wrappers. A wrapper is a kind of macro that can be used to handle a message to an object of some flavor (see the *Lisp Machine Manual,* section 20.8). System 210 introduces whoppers; they can do most of the things that wrappers can do, but have several advantages.

Both wrappers and whoppers are used in certain cases in which **:before** and **:after** daemons are not powerful enough. **:before** and **:after** daemons let you put some code before or after the execution of a method; wrappers and whoppers let you put some code *around* the execution of the method. For example, you might want to bind a special variable to some value around the execution of a method. You might also want to establish a condition handler or set up a ***catch**.

Wrappers and whoppers can also decide whether or not the method should be executed. See the *Lisp Machine Manual* for a few examples of how to do these things with wrappers.

The main difference between wrappers and whoppers is that a wrapper is like a macro, whereas a whopper is like a function. If you modify a wrapper, all of the combined methods that use that wrapper have to be recompiled; the system does this automatically, but it still takes time. If you modify a whopper, only the whopper has to be recompiled; the combined methods need not be changed. Another disadvantage of wrappers is that a wrapper's body is expanded in all of the combined methods in which it is involved, and if that body is very large and complex, all of that code is duplicated in many different compiled-code objects instead of being shared. Using whoppers is also somewhat easier than using wrappers.

Whoppers are defined with the following special form:

defwhopper *(flavor-name message-name) arglist body...* *Special Form*

Defines a whopper for the specified message to the specified flavor. *arglist* is the list of arguments, which should be the same as the argument list for any method handling the specified message.

When a message is sent to an object of some flavor, and a whopper is defined for that message, the whopper runs before any of the methods (primary or daemon). The arguments are passed, and the body of the whopper is executed. If the whopper does not do anything special, the methods themselves are never run; the result of the whopper is returned as the result of sending the message. However, most whoppers usually run the methods for the message. To make this happen, the body of the whopper calls one of the following two functions:

continue-whopper &rest *arguments*

Calls the methods for the message that was intercepted by the whopper. *arguments* is the list of arguments passed to those methods. This function must be called from inside the body of a whopper. Normally the whopper passes down the same arguments that it was given. However, some whoppers might want to change the values of the arguments and pass new values; this is valid.

lexpr-continue-whopper &rest *arguments*

This is like **continue-whopper,** but the last element of *arguments* is a list of arguments to be passed. It is useful when the arguments to the intercepted message include a **&rest** argument.

The following whopper binds the value of the special variable **base** to 3 around the execution of the **:print-integer** message to flavor **foo** (this message takes one argument):

```
(defwhopper (foo :print-integer) (n)
  (let ((base 3))
    (continue-whopper n)))
```

The following whopper sets up a ***catch** around the execution of the **:compute-height** message to flavor **giant,** no matter what arguments this message uses:

```
(defwhopper (giant :compute-height) (&rest args)
  (*catch 'too-high
    (lexpr-continue-whopper args)))
```

Like daemon methods, whoppers work in outward-in order; when you add a **defwhopper** to a flavor built on other flavors, the new whopper is placed outside any whoppers of the component flavors. However, *all* whoppers happen before *any* daemons happen. Thus, if a component defines a whopper, methods added by new flavors are considered part of the continuation of that whopper and are called only when the whopper calls its continuation.

Whoppers and wrappers are considered equal for purposes of combination. If two flavors are combined, one having a wrapper and the other having a whopper for some method, then the wrapper or whopper of the flavor that is further out is on the outside. If, for some reason, the very same flavor has both a wrapper and a whopper for the same message, the wrapper goes outside the whopper.

7.4 New features in process-run-function

The first argument to **process-run-function** can now be a list of alternating keywords and values rather than a string. The keywords are:

:name The name of the process. It must be a string. The default is Anonymous".

:restart-after-reset
: If this is **nil**, the **:reset** message to the process flushes the process. If this is **t**, the **:reset** message to the process restarts the process. The default is **nil**.

:restart-after-boot
: If this is **nil**, warm-booting the machine flushes the process. If this is **t**, warm-booting the machine restarts the process. The default is **nil**.

:warm-boot-action
: If this option is provided, its value controls what happens when the machine is warm-booted. If it is **nil** or not provided, the value of the **:restart-after-boot** option takes effect. See the **:warm-boot-action** message (*Lisp Machine Manual*, section 25.4.2) for a description of the value of the warm-boot action.

:priority
: The priority of the process. The default is 0. The meanings of these numbers are discussed in section 8.4.

:quantum
: The scheduler quantum of the process. The value should be a fixnum in units of 60ths of a second. The default is 60 (one second).

process-run-restartable-function can also have a list of alternating keywords and values as its first argument; in this case it is the same as **process-run-function** except that the default values of the **:restart-after-boot** and **:restart-after-reset** options are **t** rather than **nil**.

Note that the names of processes, like the names of everything else, are strings. Using any Lisp object or a symbol as the name of a process no longer works.

7.5 New features in resources

defresource has been changed to provide some new features and to have better paging performance. The **:initializer** is a form or function that sees *object* and its parameters as arguments when *object* is about to be allocated, whether it is being reused or was just created; it can initialize the object. The **:free-list-size** option allows you to set the size of the internal array that the resource manager uses to store free objects.

7.6 New function spec for defselect

There is a new function spec, whose format is

`(:select-method` *function-spec message*`)`

If the definition of *function-spec* is a select-method, this refers to the function to which the select method dispatches upon receiving *message*. It is an error if *function-spec* doesn't contain a select method or if the select method doesn't support *message*. **defselect** now defines its component functions using this function spec instead of creating a special symbol for each function.

Because of this change, a **defselect** compiled in System 210 does not load into an older system.

7.7 New feature in Choose Variable Values

The Choose Variable Values facility (see *Lisp Machine Choice Facilities*) has been extended so that the variable specification can be a locative or a cons cell instead of a special variable. This lets arbitrary cells in the machine be examined and modified by Choose Variable Values. (If a cons cell is passed, its car is the memory location used, and the long form of specification must be used to avoid syntactic ambiguity.)

7.8 Naming hosts not in the host table

You can now name hosts that are not in the host table. The address **CHAOS|***nnnn*, where *nnnn* is an octal numeral (of any length), refers to the host with that number on the Chaosnet. For example, **CHAOS|401** and **CHAOS|17003** are valid addresses. Normally, all hosts at your site are in the host table and have mnemonic names, so you should not need to use this syntax. However, it is useful if your host table is incorrect or out-of-date.

7.9 set-system-status is global

The function **si:set-system-status** (see the *Lisp Machine Manual*, section 24.7.5) has been moved to the global package; it can now be called by the name **set-system-status**.

Note that **set-system-status** is something that you call manually; you should *not* put it into patch files.

The function now has a fourth optional argument, called *only-update-on-disk-p*. If the value of this argument is **t**, the patch directory file is updated to show the new status, but the running Lisp environment is not modified.

7.10 Changes in property lists of files

The Lisp Machine file system allows users to define their own property names and property values for files, providing a facility for files that is analogous to the Lisp property list feature. (See *Symbolics File System*, section 1.4.) A few changes have been made.

fs:directory-list now lists all user properties as well as system properties. You no longer have to use **fs:file-properties** on the individual file to see user properties.

To remove a user property from a file, you now must set its value to a null string ("") instead of **nil**. That is, setting the value of a user property to "" is equivalent to removing the property.

The concept of "secretly changeable" properties has been removed. A file's author and creation date are now normal changeable properties.

7.11 File-naming conventions

Some details of the naming conventions for files in the Lisp Machine file system have changed.

The most important change is in directory listings. The file system now recognizes names such as "foo*" as wildcards; "foo*" matches any name that starts with the string "foo". This works for directory names as well; you can use many wildcard names within a single pathname. For example, the wildcard pathname ">>a*>b*>foo*.*" matches anything of that form; if you pass such a pathname to **fs:directory-list,** it returns a list of all files that match this pattern, even though some files are in different directories from other files.

When you need to specify the pathname of a directory, you can use the null string ("") as the type component of the pathname. In fact, there is no way to tell whether the pathname ">foo" refers to a directory or to a file without actually opening it or probing it to find out.

The root directory itself can be referred to as ">The Root Directory". A special case is needed for the root directory because it doesn't have any parent. This mode of reference is extremely uncommon; you can use it to change the properties of the root directory itself with Meta-X Change File Properties in Zmacs.

7.12 Other file-system changes

Several other changes to various parts of the file system deserve mention.

The local file system does not attempt to initialize itself, read disks, look for a file partition, or engage in any other form of activity until the first time it is invoked, at which time it seeks the file partition and initializes itself or returns a NFS (No File System) error to its caller. In earlier systems, this initialization occurred when the machine was booted, and it caused trouble if the file system did not exist or were damaged.

The performance of many file-system operations, notably directory listing and opening and probing files, has been substantially improved.

A bug in the salvager (which caused it to run at one-fourth its normal speed) has been fixed. The salvager can now run while other people are using the file system (although the orphan-seeker, if elected, still locks out other users). The salvager reports an estimated time of completion, and periodically revises and re-reports its estimate as load conditions vary. It also "ticks" (like the disk-partition copiers) in thousands of records.

If the file system runs out of space while trying to write out a file, the error that it signals can now be continued with the c-C command in the Debugger. If you are writing out a file and the file system runs out of space, you can delete and expunge some files in another process, and continue your process to get your file written out.

The incremental dumper has been made much faster, by using a special bit in the directory to keep track of whether or not a file needs to be dumped. To take advantage of this performance in your file system, you must run a special function, **(lmfs:update-dumped-bits)**. This sets up the new bits; as new file activity happens, the System 210 file system keeps these bits up to date. You must run **(lmfs:update-dumped-bits)** once over a file system to bring it up to date. This might take up to an hour if you have a large file system.

Many bugs in user property lists and the File System Editor have been fixed.

- Reloaded files no longer acquire the date of reload as a creation date.

- Files larger than four megabytes no longer have bad file headers.

- Completion now assumes file type from the default if a unique match with that file type can be made.

- Random file trashing no longer occurs while backup sets its backup dates.

- Backup maps are shorter and contain more information.

- The backup date-setting operation is much faster.

- The dumper does not halt if it cannot dump a file; it now reports the error online, to the map, and to the tape.

7.13 Changes to flow control in the serial I/O facility

Some changes have been made to the way the serial I/O facility supports the ASCII XON/XOFF protocol. The serial I/O port and its associated software are documented in *LM-2 Serial I/O*; the following information updates that document.

- The name **:ascii-protocol** is changed to **:xon-xoff-protocol**. The old name continues to work for compatibility with existing programs.

- A new parameter, **:buffering-capacity**, has been introduced to allow programs to control how often an XOFF character is checked.

- The explanation in *LM-2 Serial I/O* was inaccurate; the following new descriptions explain how the protocol works.

:xon-xoff-protocol

Allows the external device to limit the rate at which characters are transmitted by the serial I/O facility. If this is **t**, output to the serial stream is flow-controlled using the ASCII XON/XOFF (c-S/c-Q) protocol. After every *n* characters transmitted, the stream checks the receiver to see if any characters have arrived. If no character has arrived, the transmission proceeds for *n* more characters. If an ASCII XOFF or c-S character (23 octal, 19. decimal) has arrived, the stream reads characters from the receiver until an ASCII XON or c-Q character (21 octal, 17. decimal) arrives, and then proceeds with the transmission. If any other character has arrived, the stream signals an error. *n* is the **:buffering-capacity** parameter (see below). The default is **nil** (XON/XOFF feature not enabled). This parameter can only be specified when creating the stream.

:buffering-capacity

Exists only if the XON/XOFF protocol is selected. The value is a fixnum, which is the number of characters output between checks for an incoming XOFF character. Because of double buffering, the maximum number of characters that the device can see after it sends an XOFF is twice the value of the **:buffering-capacity** parameter, plus 3 or 4 characters that might be buffered in the communications interface hardware in the LM-2 and in the device.

For example, if you have an output device that sends an XOFF character when it has 32 characters of space remaining in its buffer, you should specify a **:buffering-capacity** of no more than 14 (which is (32-4)/2). The default value is 10. (These numbers are all decimal.) This parameter can only be specified when creating the stream.

8. NOTES

This section does not discuss changes. It answers some commonly asked questions and provides various news items.

8.1 Clarification of make-symbol

A poorly written example in the *Lisp Machine Manual* has confused some people about the function **make-symbol's** treatment of upper and lower case. Following is a clearer explanation of what **make-symbol** does. **make-symbol** creates a symbol whose print-name is exactly the string that it is given. Examples:

```
(make-symbol "FOO") => FOO
(make-symbol "Foo") => |Foo|
```

The vertical bars exist to inhibit the Lisp reader's normal action, which is to convert a symbol to upper case upon reading it.

8.2 How to get panes out of the Select menu

The following section answers a frequent question about the window system. Some users who have created frames only to find that every pane of the frame appears separately in the **Select** menu have asked how to get rid of them. The way to do this is to mix **tv:dont-select-with-mouse-mixin** with the flavors of all of your panes. Since you have to mix in **tv:pane-mixin** for all of your panes anyway, the flavor **tv:pane-no-mouse-select-mixin** is a mixture of both of these, so you can mix it in with all of the flavors of your panes and take care of both problems with one flavor.

8.3 The MAR facility doesn't work

The MAR facility (see the *Lisp Machine Manual*, section 26.7) currently does not work. It sometimes damages the system sufficiently that a warm-boot is needed to recover. We will attempt to fix this problem as soon as possible.

8.4 How to choose process priority levels

The *Lisp Machine Manual* explains that processes have priorities and tells you how to examine and modify the priority of a process. Following are some guidelines about what values to use.

Processes run with a priority of 0. This is the default, which most processes use. If the priority number is higher, the process receives higher priority. You should avoid using priority values higher than 20, since some critical system processes use priorities of 25 and 30; setting up competing processes could lead to degraded performance or system failure. You can also use negative values to get processes to run in the background. Values of -5 or -10 for background processes and 5 or 10 for urgent processes are reasonable. Only the relative values of these numbers are important. (You can use floating point numbers to squeeze in more intermediate levels, though there should never be any need to do so!)

8.5 How to interpret directory listings

The system displays directories of file systems in two contexts: in the File System Editor and with Meta-X View Directory and Meta-X Dired in Zmacs. Directories are displayed in a standard format, regardless of the context and regardless of what kind of file system (Lisp Machine, TOPS-20, UNIX) the directory came from. Since this format is designed to express a great deal of information in a single line, it is rather abbreviated. Some of the ways it expresses things are not clear without an explanation.

The basic format usually looks something like this:

```
pal.lisp.65    7  25548(8)    03/12/82 12:42:41 (05/13/82)   dlw
```

where:

item	*explanation*
pal	file name
lisp	file type
65	file version number
7	length of the file in blocks
25548	length of the file in bytes
8	byte-size of the file
03/12/82	date file created
12:42:41	time file created
5/13/82	date file last referred to
dlw	author

Many other things can appear in such a line; some of these things are seen only on certain types of file systems. If the first character in the line is a D, the file has been deleted (this makes sense only on file systems that support undeletion, such as the Lisp Machine and TOPS-20 file systems). After the D, if any, and before the name of the file, is the name of the physical volume that the file is stored on (on ITS, this is the disk-pack number).

On a line that describes a link rather than a file, the length numbers are replaced by a little arrow (=>), followed by the name of the target of the link. On a line that describes a subdirectory rather than a file, the length-in-blocks number is shown (if provided by the file system), but the length-in-bytes is replaced by the string `DIRECTORY`.

Next, before the dates, the line might contain any of several punctuation characters indicating things about the file. Only some of the file systems understand these flags. Following is a list of the various characters and the flags they indicate:

character	*flag*
!	not backed up
@	don't delete
$	don't reap

For lines indicating subdirectories, the reference date can be replaced with a date preceded by X=, the date this directory was last expunged. The dates are followed by the file author's name, which is followed by the name of the last user to read the file.

Only certain file systems support certain features. Most file systems don't keep track of the last reader's name and don't have something comparable to a "don't delete" flag. Therefore, any of the above fields might be omitted on certain file systems. However, the same general format is followed for all file systems and so you can interpret the meaning of a line in a directory listing, even for a file system that you are not familiar with.

8.6 Status report on problems with UNIX file systems

Two major UNIX file-system support problems, announced in the System 78 Release Notes, are still not fixed.

- *System file-type abbreviations*
  Files of type lisp are represented in the UNIX file system with names that end with .l. This means that a file named foo.l is parsed as having the name foo and the type lisp. Thus, when the system encounters a file that is actually named foo.lisp, it cannot represent it. This means that the system cannot handle files whose names end with .lisp or .text or any of the other system types.

- *Lack of case distinction*
  The file system's internal representation of file names does not distinguish upper-case characters from lower-case characters, whereas UNIX does. Therefore, if you refer to a file with the wrong case, the system cannot refer to it with the correct case until it is cold-booted.

These are high-priority problems that we will try to solve in the next release.

8.7 Several prominent bug fixes

The Zmail Background process now knows how to deal with errors that occur while doing I/O to its file computer.

The functions **monitor-variable** and **unmonitor-variable** now work.

Redefining a wrapper now automatically triggers the necessary recompilation of the combined method of the flavor (see the *Lisp Machine Manual*, section 20.8). If a wrapper is given a new definition, the combined method is recompiled so that it gets the new definition. If a wrapper is redefined with the same old definition, the existing combined methods will keep being used, since they are still correct.

8.8 How to establish Chaosnet servers

The following elaborates on the explanation of how to establish servers for new Chaosnet protocols in *Chaosnet FILE Protocol.*

Suppose you created a new protocol with contact name ALPHA and wanted to set up a server for the ALPHA protocol. You'd put the following top-level form into your program:

```
(add-initialization
  "ALPHA"
  '(process-run-function "ALPHA Server" 'alpha-server)
  nil
  'chaos:server-alist)
```

To remove the server, you would evaluate the following form:

```
(delete-initialization "ALPHA" nil 'chaos:server-alist)
```

In other words, an initialization list, stored in the variable **chaos:server-alist,** associates contact names with forms to be run when some network host tries to connect to that contact name. Your form should always create a new process to handle the connection; in this example, the name of the process is "ALPHA Server", and the first thing that the process does is call the function **alpha-server**. This function should start off by calling **chaos:listen** to get the connection object; then it can call **chaos:accept, chaos:reject, chaos:answer,** or **chaos:forward,** as appropriate.

8.9 How to add new keywords for add-initialization

If you want to add new keywords that can be understood by **add-initialization** and the other initialization functions, you can do so by pushing a new element onto the following variable:

si:initialization-keywords *Variable*

Each element on this list defines the name of one initialization list. Each element is a list of two or three elements. The first is the keyword symbol that names the initialization list. The second is a special variable, whose value is the initialization list itself. The third, if present, is a symbol defining the default time at which initializations added to this list should be evaluated; it should be **si:normal, si:now, si:first,** or **si:redo**. The third element is the default; if the list of keywords passed to **add-initialization** contains one of the keywords **normal, now, first,** or **redo,** it overrides this default. If the third element is not present, **si:normal** is assumed.

Note that the keywords used in **add-initialization** need not be keyword-package symbols (you are allowed to use **first** as well as **:first**), because **string-equal** is used to recognize the symbols.

Two new system initialization lists (see the *Lisp Machine Manual,* section 29.1) have been created, **site** and **full-gc** (see the *Software Installation Guide* and section 6.17, respectively).

Of the predefined system initialization lists, the default time for almost all of them is **normal,** with the following exceptions: the **system** and **once** initialization lists default to **first,** and the **site** initialization list defaults to **now**.

8.10 Remote tape support and new reload/retrieve

A new Chaosnet protocol allows a Lisp Machine to access the tape drive of any server on the network and use that tape drive as if it were directly connected to the Lisp Machine. This lets you use the tape drive of any machine on your network for Lisp Machine backup dumps as well as for loading distribution tapes and moving information around on tapes. Unfortunately, documentation on the details of this facility is not yet available.

There is also a new, more advanced command for getting files back from backup tapes. This is the new **Reload/Retrieve** command in the File System Maintenance program. Documentation on this is not yet available; however, information is available in the menu-driven interface.

8.11 Hints for TOPS-20 users

If the TOPS-20 is the SYS host (that is, if you store your copies of the Lisp Machine source files on your TOPS-20), it is important to allow all of your Lisp Machine users to access those files. Rather than having to create a separate TOPS-20 account for each Lisp Machine user, you can instead create a single account named "Anonymous". The latest version of the TOPS-20 Chaosnet file server (CHAOS.FILE.372 or any later version) knows about the name "Anonymous", so instead of using the password that the FILE protocol user supplies, it reads the password from the file SYS:ANONYMOUS.USERFILE. With this arrangement, you can tell Lisp Machine users to log in as "Anonymous" and supply a null password; the file server will then log in under the "Anonymous" account and have its priviliges. To set this up at your site, you must must do the following:

1. Install the latest version of the file server.
2. Create the <ANONYMOUS> account and give it read-access to the Lisp Machine sources.
3. Create the file SYS:ANONYMOUS.USERFILE, with protection 777700, containing the password for the "Anonymous" account.

If you have done this, you might then want to edit your SITE file (see the *Software Installation Guide*) to make "Anonymous" the value of the **:system-login-user** site option. This way, when the Lisp Machine logs in to load patches or microcode error tables, it will not need to ask the user for a password.

If your account on TOPS-20 is set up as a "wheel", and you want your file server to enable itself so that it takes advantage of your wheel privileges, you should type * immediately before your password when your file server logs in. For example, if account SMITH has password SECRET and has wheel privileges, log into that account by typing the password as *SECRET, and the file server will enable itself.

8.12 Setting the base and package in your INIT file

Several users have run into the following confusing problem: if you try to change the current package by putting a **pkg-goto** form into your INIT file, it doesn't work. If you try to change the current base by setting the values of **base** and **ibase** in your INIT file, it sometimes doesn't work. The reason is that when a file is loaded into the Lisp Machine, the values of several variables are bound while the forms inside the file are evaluated. The value of **package** is always bound when a file is being loaded, so the effects of any **pkg-goto** form inside the file are undone at the end of the loading. The value of **base** is bound if and only if the file has a **base** attribute in its attribute list. Therefore, your INIT file cannot change the current package; it can change the values of **base** and **ibase,** but only if it does *not* contain a **base** attribute in its attribute list.

INDEX

symbolics inc.

System 210 Release Notes
990090

Design: Schafer/La Casse
Typesetting: Cover — Litho Composition Co.
Printing: Henry Sawyer Co.