28. Maintaining Large Systems

When a program gets large, it is often desirable to split it up into several files. One reason for this is to help keep the parts of the program organized, to make things easier to find. It's also useful to have the program broken into small pieces that are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any piece of it changes; if the program is broken up into many files, only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more commands are needed to manipulate it. To load the program, you now have to load several files separately, instead of just loading one file. To compile it, you have to figure out which files need compilation, by seeing which have been edited since they were last compiled, and then you have to compile those files.

What's even more complicated is that files can have interdependencies. You might have a file called DEFS that contains some macro definitions (or flavor or structure definitions), and functions in other files might use those macros. This means that in order to compile any of those other files, you must first load the file DEFS into the Lisp environment so that the macros will be defined and can be expanded at compile time. You have to remember this whenever you compile any of those files. Furthermore, if DEFS has changed, other files of the program may need to be recompiled because the macros may have changed and need to be re-expanded.

This chapter describes the *system* facility, which takes care of all these things for you. The way it works is that you define a set of files to be a *system*, using the defsystem special form, described below. This system definition says which files make up the system, which ones depend on the presence of others, and so on. You put this system definition into its own little file, and then all you have to do is load that file and the Lisp environment will know about your system and what files are in it. You can then use the make-system function (see page 666) to load in all the files of the system, recompile all the files that need compiling, and so on.

The system facility is very general and extensible. This chapter explains how to use it and how to extend it. This chapter also explains the *patch* facility, which lets you conveniently update a large program with incremental changes.

28.1 Defining a System

defsystem *name* (*keyword args...*)... *Macro*

Defines a system named *name*. The options selected by the keywords are explained in detail later. In general, they fall into two categories: properties of the system and *transformations*. A transformation is an operation such as compiling or loading that takes one or more files and does something to them. The simplest system is a set of files and a transformation to be performed on them.

Here are a few examples.

```
(defsystem mysys
  (:compile-load ("OZ:<GEORGE>PROG1.LISP" "OZ:<GEORGE2>PROG2.LISP")))

(defsystem zmail
  (:name "ZMail")
  (:pathname-default "SYS: ZMAIL;")
  (:package zwei)
  (:module defs "DEFS")
  (:module mult "MULT" :package tv)
  (:module main ("TOP" "COMNDS" "MAIL" "USER" "WINDOW"
                 "FILTER" mult "COMETH"))
  (:compile-load defs)
  (:compile-load main (:fasload defs)))

(defsystem bar
  (:module reader-macros "BAR:BAR;RDMAC")
  (:module other-macros "BAR:BAR;MACROS")
  (:module main-program "BAR:BAR;MAIN")
  (:compile-load reader-macros)
  (:compile-load other-macros (:fasload reader-macros))
  (:compile-load main-program (:fasload reader-macros
                                        other-macros)))
```

The first example defines a new *system* called **mysys**, which consists of two files, stored on a Tops-20 host names OZ, both of which are to be compiled and loaded. The second example is somewhat more complicated. What all the options mean is described below, but the primary difference is that there is a file **DEFS** which must be loaded before the rest of the files (**main**) can be compiled. Also, the files are stored on logical host SYS and directory ZMAIL.

The last example has two levels of dependency. **reader-macros** must be compiled and loaded before **other-macros** can be compiled. Both **reader-macros** and **other-macros** must then be loaded before **main-program** can be compiled. All the source files are stored on host BAR, presumably a logical host defined specifically for this system. It is desirable to use a logical host for the files of a system if there is a chance that people at more than one site will be using it; the logical host allows the identical **defsystem** to be valid at all sites. See section 24.7.5, page 572 for more on logical hosts and logical pathnames.

Note that The **defsystem** options other than transformations are:

:name Specifies a "pretty" version of the name for the system, for use in printing.

:short-name
: Specified an abbreviated name used in constructing disk label comments and in patch file names for some file systems.

:component-systems
: Specifies the names of other systems used to make up this system. Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems. The format is (**:component-systems** *names*...).

:package
: Specifies the package in which transformations are performed. A package specified here overrides one in the -*- line of the file in question.

:pathname-default
: Gives a local default within the definition of the system for strings to be parsed into pathnames. Typically this specifies the directory, when all the files of a system are on the same directory.

:warnings-pathname-default
: Gives a default for the file to use to store compiler warnings in, when **make-system** is used with the :batch option.

:patchable
: Makes the system be a patchable system (see section 28.8, page 672). An optional argument specifies the directory to put patch files in. The default is the :pathname-default of the system.

:initial-status
: Specifies what the status of the system should be when **make-system** is used to create a new major version. The default is :experimental. See section 28.8.5, page 679 for further details.

:not-in-disk-label
: Make a patchable system not appear in the disk label comment. This should probably never be specified for a user system. It is used by patchable systems internal to the main Lisp system, to avoid cluttering up the label.

:default-binary-file-type
: Specifies the file type to use for compiled Lisp files. The value you specify should be a string. If you do not specify this, the standard file type :qfasl is used.

:module
: Allows assigning a name to a set of files within the system. This name can then be used instead of repeating the filenames. The format is (:module *name files options...*). *files* is usually a list of filenames (strings). In general, it is a *module-specification*, which can be any of the following:

  a string
  : This is a file name.

  a symbol
  : This is a module name. It stands for all of the files which are in that module of this system.

  an *external module component*
  : This is a list of the form (*system-name module-names...*), to specify modules in another system. It stands for all of the files which are in all of those modules.

  a list of *module components*
  : A module component is any of the above, or the following:

  a list of file names
  : This is used in the case where the names of the input and output files of a transformation are not related according to the standard naming conventions, for

example when a QFASL file has a different name or resides on a different directory than the source file. The file names in the list are used from left to right, thus the first name is the source file. Each file name after the first in the list is defaulted from the previous one in the list.

To avoid syntactic ambiguity, this is allowed as a module component but not as a module specification.

The currently defined options for the :module clause are

:package Overrides any package specified for the whole system for transformations performed on just this module.

In the second defsystem example above, there are three modules. Each of the first two has only one file, and the third one (main) is made up both of files and another module. To take examples of the other possibilities,

```
(:module prog (("SYS: GEORGE; PROG" "SYS: GEORG2; PROG")))
(:module foo (defs (zmail defs)))
```

The prog module consists of one file, but it lives in two directories, GEORGE and GEORG2. If this were a Lisp program, that would mean that the file SYS: GEORGE; PROG LISP would be compiled into SYS: GEORG2; PROG QFASL. The foo module consists of two other modules the defs module in the same system, and the defs module in the zmail system. It is not generally useful to compile files that belong to other systems; thus this foo module would not normally be the subject of a transformation. However, *dependencies* (defined below) use modules and need to be able to refer to (depend on) modules of other systems.

si:set-system-source-file *system-name filename*

This function specifies which file contains the defsystem for the system *system-name*. *filename* can be a pathname object or a string.

Sometimes it is useful to say where the definition of a system can be found without taking time to load that file. If make-system, or require (page 672), is ever used on that system, the file whose name has been specified will be loaded automatically.

28.2 Transformations

Transformations are of two types, simple and complex. A simple transformation is a single operation on a file, such as compiling it or loading it. A complex transformation takes the output from one transformation and performs another transformation on it, such as loading the results of compilation.

The general format of a simple transformation is (*name input dependencies condition*). *input* is usually a module specification or another transformation whose output is used. The transformation *name* is to be performed on all the files in the module, or all the output files of the other transformation.

dependencies and *condition* are optional.

dependencies is a *transformation specification*, either a list (*transformation-name module-names...*) or a list of such lists. A *module-name* is either a symbol that is the name of a module in the current system, or a list (*system-name module-names...*). A dependency declares that all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. Thus in the zmail example above, the defs module must have the :fasload transformation performed on it before the :compile transformation can be performed on main.

The dependency has to be a tranformation that is explicitly specified as a transformation in the system definition, not just an action that might be performed by anything. That is, if you have a dependency (:fasload foo), it means that (fasload foo) is a tranformation of your system and you depend on that tranformation; it does not simply mean that you depend on foo's being loaded. Furthermore, it doesn't work if (:fasload foo) is an implicit piece of another tranformation. For example, the following works:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:compile-load (foo bar)))
```

but this doesn't work:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:module blort "BLORT")
  (:compile-load (foo bar))
  (:compile-load blort (:fasload foo)))
```

because foo's :fasload is not mentioned explicitly (i.e. at top level) but is only implicit in the (:compile-load (foo bar)). One must instead write:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:module blort "BLORT")
  (:compile-load foo)
  (:compile-load bar)
  (:compile-load blort (:fasload foo)))
```

condition is a predicate which specifies when the transformation should take place. Generally it defaults according to the type of the transformation. Conditions are discussed further on page 671.

The defined simple transformations are:

:fasload	Calls the fasload function to load the indicated files, which must be QFASL files whose pathnames have canonical type :qfasl (see section 24.2.3, page 551). The *condition* defaults to si:file-newer-than-installed-p, which is t if a newer version of the file exists on the file computer than was read into the current environment.

:readfile Calls the readfile function to read in the indicated files, whose names must have canonical type :lisp. Use this for files that are not to be compiled. *condition* defaults to si:file-newer-than-installed-p.

:compile Calls the compile-file function to compile the indicated files, whose names must have canonical type :lisp. *condition* defaults to si:file-newer-than-file-p, which returns t if the source file has been written more recently than the binary file.

A special simple transformation is

:do-components
(:do-components *dependencies*) inside a system with component systems causes the *dependencies* to be done before anything in the component systems. This is useful when you have a module of macro files used by all of the component systems.

The defined complex transformations are

:compile-load (:compile-load *input compile-dependencies load-dependencies compile-condition load-condition*) is the same as (:fasload (:compile *input compile-dependencies compile-condition*) *load-dependencies load-condition*). This is the most commonly-used transformation. Everything after *input* is optional.

:compile-load-init
See page 671.

As was explained above, each filename in an input specification can in fact be a list of strings when the source file of a program differs from the binary file in more than just the file type. In fact, every filename is treated as if it were an infinite list of filenames with the last filename, or in the case of a single string the only filename, repeated forever at the end. Each simple transformation takes some number of input filename arguments and some number of output filename arguments. As transformations are performed, these arguments are taken from the front of the filename list. The input arguments are actually removed and the output arguments left as input arguments to the next higher transformation. To make this clearer, consider the prog module above having the :compile-load transformation performed on it. This means that prog is given as the input to the :compile transformation and the output from this transformation is given as the input to the :fasload transformation. The :compile transformation takes one input filename argument, the name of a Lisp source file, and one output filename argument, the name of the QFASL file. The :fasload transformation takes one input filename argument, the name of a QFASL file, and no output filename arguments. So, for the first and only file in the prog module, the filename argument list looks like ("SYS: GEORGE; PROG" "SYS: GEORG2; PROG" "SYS: GEORG2; PROG" ...). The :compile transformation is given arguments of "SYS: GEORGE; PROG" and "SYS: GEORG2; PROG" and the filename argument list which it outputs as the input to the :fasload transformation is ("SYS: GEORG2; PROG" "SYS: GEORG2; PROG" ...). The :fasload transformation then is given its one argument of "SYS: GEORG2; PROG".

Note that dependencies are not transitive or inherited. For example, if module a depends on macros defined in module b, and therefore needs b to be loaded in order to compile, and b has a similar dependency on c, c need not be loaded for compilation of a. Transformations with these dependencies would be written

```
        (:compile-load a (:fasload b))
        (:compile-load b (:fasload c))
```

To say that compilation of a depends on both b and c, you would instead write

```
        (:compile-load a (:fasload b c))
        (:compile-load b (:fasload c))
```

If in addition a depended on c (but not b) during loading (perhaps a contains defvars whose initial values depend on functions or special variables defined in c) you would write the transformations

```
        (:compile-load a (:fasload b c) (:fasload c))
        (:compile-load b (:fasload c))
```

28.3 Making a System

make-system *name* &rest *keywords*

The make-system function does the actual work of compiling and loading. In the example above. if PROG1 and PROG2 have both been compiled recently, then

```
        (make-system 'mysys)
```

loads them as necessary. If either one might also need to be compiled, then

```
        (make-system 'mysys :compile)
```

does that first as necessary.

The very first thing make-system does is check whether the file which contains the defsystem for the specified system has changed since it was loaded. If so, it offers to load the latest version, so that the remainder of the make-system can be done using the latest system definition. (This only happens if the filetype of that file is LISP.) After loading this file or not, make-system goes on to process the files that compose the system.

If the system name is not recognized, make-system attempts to load the file SYS: SITE; *system-name* SYSTEM, in the hope that that contains a system definition or a call to si:set-system-source-file.

make-system lists what transformations it is going to perform on what files, then asks the user for confirmation. If the user types S when confirmation is requested, then make-system asks about each file individually so that the user can decide selectively which transformations should be performed; then collective reconfirmation is requested. This is like what happens if the :selective keyword is specified. If the user types Y, the transformations are performed. Before each transformation a message is printed listing the transformation being performed, the file it is being done to, and the package. This behavior can be altered by *keywords*.

If the system being made is patchable, and if loading has not been inhibited, then the system's patches are loaded afterward. Loading of patches is silent if the make-system is, and requires confirmation if the make-system does.

These are the keywords recognized by the make-system function and what they do.

:noconfirm Assumes a yes answer for all questions that would otherwise be asked of the user.

:selective	Asks the user whether or not to perform each transformation that appears to be needed for each file.
:silent	Avoids printing out each transformation as it is performed.
:reload	Bypasses the specified conditions for performing a transformation. Thus files are compiled even if they haven't changed and loaded even if they aren't newer than the installed version.
:noload	Does not load any files except those required by dependencies. For use in conjunction with the :compile option.
:compile	Compiles files also if need be. The default is to load but not compile.
:recompile	This is equivalent to a combination of :compile and :reload: it specifies compilation of all files, even those whose sources have not changed since last compiled.
:no-increment-patch	When given along with the :compile option, disables the automatic incrementing of the major system version that would otherwise take place. See section 28.8, page 672.
:increment-patch	Increments a patchable system's major version without doing any compilations. See section 28.8, page 672.
:no-reload-system-declaration	Turns off the check for whether the file containing the defsystem has been changed. Then the file is loaded only if it has never been loaded before.
:batch	Allows a large compilation to be done unattended. It acts like :noconfirm with regard to questions, turns off more-processing and fdefine-warnings (see inhibit-fdefine-warnings, page 240), and saves the compiler warnings in an editor buffer and a file (it asks you for the name).
:defaulted-batch	This is like :batch except that it uses the default for the pathname to store warnings in and does not ask the user to type a pathname.
:print-only	Just prints out what transformations would be performed; does not actually do any compiling or loading.
:noop	Is ignored. This is useful mainly for programs that call make-system, so that such programs can include forms like `(make-system 'mysys (if compile-p :compile :noop))`

28.4 Adding New Keywords to make-system

make-system keywords are defined as functions on the si:make-system-keyword property of the keyword. The functions are called with no arguments. Some of the relevant variables they can use are

si:*system-being-made* *Variable*
The internal data structure that represents the system being made.

si:*make-system-forms-to-be-evaled-before* *Variable*
A list of forms that are evaluated before the transformations are performed.

si:*make-system-forms-to-be-evaled-after* *Variable*
A list of forms that are evaluated after the transformations have been performed. Transformations can push entries here too.

si:*make-system-forms-to-be-evaled-finally* *Variable*
A list of forms that are evaluated by an unwind-protect when the body of make-system is exited, whether it is completed or not. Closing the batch warnings file is done here. Unlike the si:*make-system-forms-to-be-evaled-after* forms, these forms are evaluated outside of the "compiler warnings context".

si:*query-type* *Variable*
Controls how questions are asked. Its normal value is :normal. :noconfirm means ask no questions and :selective means asks a question for each individual file transformation.

si:*silent-p* *Variable*
If t, no messages are printed out.

si:*batch-mode-p* *Variable*
If t, :batch was specified.

si:*redo-all* *Variable*
If t, all transformations are performed, regardless of the condition functions.

si:*top-level-transformations* *Variable*
A list of the types of transformations that should be performed, such as (:fasload :readfile). The contents of this list are controlled by the keywords given to make-system. This list then controls which transformations are actually performed.

si:*file-transformation-function* *Variable*
The actual function that gets called with the list of transformations that need to be performed. The default is si:do-file-transformations.

si:define-make-system-special-variable *variable value [defvar-p]* *Macro*
Causes *variable* to be bound to *value* during the body of the call to make-system. This allows you to define new variables similar to those listed above. *value* is evaluated on entry to make-system. If *defvar-p* is specified as (or defaulted to) t, *variable* is defined with defvar. It is not given an initial value. If *defvar-p* is specified as *nil*, *variable* belongs to some other program and is not defvar'ed here.

The following simple example adds a new keyword to make-system called :just-warn, which means that fdefine warnings (see page 239) regarding functions being overwritten should be printed out, but the user should not be queried.

```
(si:define-make-system-special-variable
   inhibit-fdefine-warnings inhibit-fdefine-warnings nil)

(defun (:just-warn si:make-system-keyword) ()
  (setq inhibit-fdefine-warnings :just-warn))
```

(See the description of the inhibit-fdefine-warnings variable, on page 240.)

make-system keywords can do something directly when called, or they can have their effect by pushing a form to be evaluated onto si:*make-system-forms-to-be-evaled-after* or one of the other two similar lists. In general, the only useful thing to do is to set some special variable defined by si:define-make-system-special-variable. In addition to the ones mentioned above, user-defined transformations may have their behavior controlled by new special variables, which can be set by new keywords. If you want to get at the list of transformations to be performed, for example, the right way is to set si:*file-transformation-function* to a new function, which then can call si:do-file-transformations with a possibly modified list. That is how the :print-only keyword works.

28.5 Adding New Options for defsystem

Options to defsystem are defined as macros on the si:defsystem-macro property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return nil. There are several variables they can use; the only one of general interest is

si:*system-being-defined* *Variable*

The internal data structure that represents the system that is currently being constructed.

si:define-defsystem-special-variable *variable value* *Macro*

Causes *value* to be evaluated and *variable* to be bound to the result during the expansion of the defsystem special form. This allows you to define new variables similar to the one listed above.

si:define-simple-transformation *Macro*

This is the most convenient way to define a new simple transformation. The form is

(si:define-simple-transformation *name function*
default-condition input-file-types output-file-types
pretty-names compile-like load-like)

For example,

```
(si:define-simple-transformation :compile si:qc-file-1
          si:file-newer-than-file-p (:lisp) (:qfasl))
```

input-file-types and *output-file-types* are how a transformation specifies how many input filenames and output filenames it should receive as arguments, in this case one of each. They also, obviously, specify the default file type for these pathnames. The si:qc-file-1 function is mostly like compile-file, except for its interface to packages. It takes input-file and output-file arguments.

pretty-names, *compile-like*, and *load-like* are optional.

pretty-names specifies how messages printed for the user should print the name of the transformation. It can be a list of the imperative ("Compile"), the present participle ("Compiling"), and the past participle ("compiled"). Note that the past participle is not capitalized, because when used it does not come at the beginning of a sentence. *pretty-names* can be just a string, which is taken to be the imperative, and the system will conjugate the participles itself. If *pretty-names* is omitted or nil it defaults to the name of the transformation.

compile-like and *load-like* say when the transformation should be performed. Compile-like transformations are performed when the :compile keyword is given to make-system. Load-like transformations are performed unless the :noload keyword is given to make-system. By default *compile-like* is t but *load-like* is nil.

Complex transformations are defined as normal macro expansions, for example,

```
(defmacro (:compile-load si:defsystem-macro)
                (input &optional com-dep load-dep
                                 com-cond load-cond)
  '(:fasload (:compile ,input .com-dep ,com-cond)
             ,load-dep ,load-cond))
```

28.6 More Esoteric Transformations

It is sometimes useful to specify a transformation upon which something else can depend, but which is performed not by default, but rather only when requested because of that dependency. The transformation nevertheless occupies a specific place in the hierarchy. The :skip defsystem macro allows specifying a transformation of this type. For example, suppose there is a special compiler for the read table which is not ordinarily loaded into the system. The compiled version should still be kept up to date, and it needs to be loaded if ever the read table needs to be recompiled.

```
(defsystem reader
  (:pathname-default "SYS: IO;")
  (:package system-internals)
  (:module defs "RDDEFS")
  (:module reader "READ")
  (:module read-table-compiler "RTC")
  (:module read-table "RDTBL")
  (:compile-load defs)
  (:compile-load reader (:fasload defs))
  (:skip :fasload (:compile read-table-compiler))
  (:rtc-compile-load read-table (:fasload read-table-compiler)))
```

Assume that there is a complex transformation :rtc-compile-load, which is like :compile-load except that is is built on a transformation called something like :rtc-compile, which uses the read table compiler rather than the Lisp compiler. In the above system, then, if the :rtc-compile transformation is to be performed, the :fasload transformation must be done on read-table-compiler first, that is the read table compiler must be loaded if the read table is to be recompiled. If you say (make-system 'reader :compile), then the :compile transformation is done on the read-table-compiler module despite the :skip, compiling the read table compiler if need be. If you say (make-system 'reader), the reader and the read table are loaded, but the :skip keeps this from happening to the read table compiler.

So far nothing has been said about what can be given as a *condition* for a transformation except for the default functions, which check for conditions such as a source file being newer than the binary. In general, any function that takes the same arguments as the transformation function (e.g. compile-file) and returns t if the transformation needs to be performed, can be in this place as a symbol, including for example a closure. To take an example, suppose there is a file that contains compile-flavor-methods for a system and that should therefore be recompiled if any of the flavor method definitions change. In this case, the condition function for compiling that file should return t if either the source of that file itself or any of the files that define the flavors have changed. This is what the :compile-load-init complex transformation is for. It is defined like this:

```
(defmacro (:compile-load-init si:defsystem-macro)
                (input add-dep &optional com-dep load-dep
                 &aux function)
  (setq function (let-closed ((*additional-dependent-modules*
                                add-dep))
                   'compile-load-init-condition))
  `(:fasload (:compile ,input ,com-dep ,function) ,load-dep))

(defun compile-load-init-condition (source-file qfasl-file)
  (or (si:file-newer-than-file-p source-file qfasl-file)
      (local-declare ((special *additional-dependent-modules*))
        (si:other-files-newer-than-file-p
                         *additional-dependent-modules*
                         qfasl-file))))
```

The condition function generated when this macro is used returns t either if si:file-newer-than-file-p would with those arguments, or if any of the other files in add-dep, which presumably is a *module specification*, are newer than the QFASL file. Thus the file (or module) to which the :compile-load-init transformation applies will be compiled if it or any of the source files it depends on has been changed, and will be loaded under the normal conditions. In most (but not all cases), com-dep is a :fasload transformation of the same files as add-dep specifies, so that all the files this one depends on will be loaded before compiling it.

28.7 Common Lisp Modules

In Common Lisp, a *module* is a name given to a group of files of code. Modules are not like systems because nothing records what the "contents" of any particular module may be. Instead, one of the files which defines the module contains a provide form which says, when that file is loaded, "Module foo is now present." Other files may say, using require, "I want to use module foo."

Normally the require form also specifies the files to load if foo has not been provide'd already. This is where the information of which files are in a module is stored. If the require does not have file names in it, the module name foo is used in an implementation-dependent manner to find files to load. The Lisp Machine does this by using it as a system name in **make-system**.

provide *module-name*
Adds *module-name* to the list ***modules*** of modules already loaded. *module-name* should be a string; case is significant.

require *module-name* &rest *files*
If module *module-name* is not already loaded (on ***modules***), *files* are loaded in order to make the module available. *module-name* should be a string; case is significant. The elements of *files* should be pathnames or namestrings. If *files* is nil, (**make-system** *module-name* **:noconfirm**) is done. Note, however, that case is not significant in the argument to **make-system**.

modules *Variable*
A list of names (strings) of all modules **provide**'d so far.

28.8 The Patch Facility

The patch facility allows a system maintainer to manage new releases of a large system and issue patches to correct bugs. It is designed to be used to maintain both the Lisp Machine system itself and applications systems that are large enough to be loaded up and saved on a disk partition.

When a system of programs is very large, it needs to be maintained. Often problems are found and need to be fixed, or other little changes need to be made. However, it takes a long time to load up all of the files that make up such a system, and so rather than having every user load up all the files every time he wants to use the system, usually the files just get loaded once into a Lisp world, which is then saved away on a disk partition. Users then use this disk partition, copies of which may appear on many machines. The problem is that since the users don't load up the system every time they want to use it, they don't get all the latest changes.

The purpose of the patch system is to solve this problem. A *patch* file is a little file that, when you load it, updates the old version of the system into the new version of the system. Most often, patch files just contain new function definitions; old functions are redefined to do their new thing. When you want to use a system, you first use the Lisp environment saved on the disk, and then you load all the latest patches. Patch files are very small, so loading them doesn't take much time. You can even load the saved environment, load up the latest patches, and then save it away, to save future users the trouble of even loading the patches. (Of course, new patches may be made later, and then these will have to be loaded if you want to get the very latest version.)

For every system, there is a series of patches that have been made to that system. To get the latest version of the system, you load each patch file in the series, in order. Sooner or later, the maintainer of a system wants to stop building more and more patches, and recompile everything, starting afresh. A complete recompilation is also necessary when a system is changed in a far-reaching way, that can't be done with a small patch; for example, if you completely reorganize a program, or change a lot of names or conventions, you might need to completely recompile it to make it work again. After a complete recompilation has been done, the old patch files are no longer suitable to use; loading them in might even break things.

The way all this is kept track of is by labelling each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made; the patch is identified by the major and minor version number together. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number. Thus, patch 93.9 is for major version 93 and minor version 9; it is followed by patch 93.10.

To clarify this, here is a typical scenario. A new system is created; its initial version number is 1.0. Then a patch file is created; the version of the program that results from loading the first patch file into version 1.0 is called 1.1. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2. Then the entire system is recompiled, creating version 2.0 from scratch. Now the two patch files are irrelevant, because they fix old software; the changes that they reflect are integrated into system 2.0.

Note that the second patch file should only be loaded into system 1.1 in order to create system 1.2; you shouldn't load it into 1.0 or any other system besides 1.1. It is important that all the patch files be loaded in the proper order, for two reasons. First, it is very useful that any system numbered 1.1 be exactly the same software as any other system numbered 1.1, so that if somebody reports a bug in version 1.1, it is clear just which software is being complained about. Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

The patch facility keeps track of all the patch files that exist, remembering which version each one creates. There is a separate numbered sequence of patch files for each major version of each system. All of them are stored in the file system, and the patch facility keeps track of where they all are. In addition to the patch files themselves, there are *patch directory* files that contain the patch facility's data base by which it keeps track of what minor versions exist for a major version, and what the last major version of a system is. These files and how to make them are described below.

In order to use the patch facility, you must define your system with **defsystem** (see chapter 28, page 660) and declare it as patchable with the **:patchable** option. When you load your system (with **make-system**, see page 666), it is added to the list of all systems present in the world. The patch facility keeps track of which version of each patchable system is present and where the data about that system reside in the file system. This information can be used to update the Lisp world automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to be loaded, ask which patches are already loaded, and add new patches.

You can also load in patches or whole new systems and then save the entire Lisp environment away in a disk partition. This is explained on section 35.11, page 804.

When a Lisp Machine is booted, it prints out a line of information for each patchable system present in the booted Lisp world, saying which major and minor versions are loaded. This is done by print-herald (see page 674).

print-system-modifications &rest *system-names*
With no arguments, this lists all the systems present in this world and, for each system, all the patches that have been loaded into this world. For each patch it shows the major version number (which is always the same since a world can only contain one major version), the minor version number, and an explanation of what the patch does, as typed in by the person who made the patch.

If print-system-modifications is called with arguments, only the modifications to the systems named are listed.

print-herald &optional *format-dest*
Prints the names and loaded version numbers of all patchable systems loaded, and the microcode. Also printed are the number of the band you booted, the amount of physical and virtual memory you have, the host name of the machine, and its associated machine name. Example:

```
MIT System, band 7 of CADR-1.
640K physical memory, 16127K virtual memory.
 System        98.43
 CADR          3.6
 ZMail         53.10
 MIT-Specific  22.0
 Microcode     309
MIT Lisp Machine One, with associated machine OZ.
```

format-dest defaults to t; if it is nil the answer is returned as a string rather than printed out. *format-dest* can also be a stream to print on.

si:get-system-version &optional *system*
Returns two values, the major and minor version numbers of the version of *system* currently loaded into the machine, or nil if that system is not present. *system* defaults to "System".

si:system-version-info &optional (*brief-p* nil)
Returns a string giving information about which systems and what versions of the systems are loaded into the machine, and what microcode version is running. A typical string for it to produce is:

```
"System 98.48, CADR 3.6, MIT-Specific 22.0, microcode 309"
```

If *brief-p* is t, it uses short names, suppresses the microcode version, any systems which should not appear in the disk label comment, the name System, and the commas:

```
"98.48"
```

28.8.1 Defining a System

In order to use the patch facility, you must declare your system as patchable by giving the :patchable option to defsystem (see chapter 28, page 660). The major version of your system in the file system is incremented whenever make-system is used to compile it. Thus a major version is associated with a set of QFASL files. The major version of your system that is remembered as having been loaded into the Lisp environment is set to the major version in the file system whenever make-system is used to load your system and the major version in the file system is greater than what you had loaded before.

After loading your system, you can save it with the disk-save function (see page 807). disk-save asks you for any additional information you want printed as part of the greeting when the machine is booted. This is in addition to the names and versions of all the systems present in this world. If the system version does not fit in the partition comment field allocated in the disk label, disk-save asks you to type in an abbreviated form.

28.8.2 Loading Patches

load-patches &rest *options*

This function is used to bring the current world up to the latest minor version of whichever major version it is, for all systems present, or for certain specified systems. If there are any patches available, load-patches offers to read them in. With no arguments, load-patches updates all the systems present in this world. If you do not specify the systems to operate on, load-patches also reloads the site files if they have changed (section 35.12, page 810), and reloads the files defining logical host translations if they have changed (page 574).

options is a list of keywords. Some keywords are followed by an argument. The following options are accepted:

:systems *list*	*list* is a list of names of systems to be brought up to date. If this option is not specified, all patchable systems loaded are processed.
:unreleased	Loads unreleased patches with no special querying. These patches should be loaded for experimental use if you wish the benefit of the latest bug fixes, but should not be loaded if you plan to save a band.
:site	Loads the latest site files if they have been changed since last loaded. This is the default if you do not specify explicitly which systems to process.
:nosite	Prevents loading of site files. This is the default when you specify the systems to process.
:hosts	Reloads the files defining logical host translations if they have been changed since last loaded. This is the default if you do not specify explicitly which systems to process.
:nohosts	Prevents loading of logical host translation files. This is the default when you specify the systems to process.

:verbose	Prints an explanation of what is being done. This is the default.
:selective	For each patch, says what it is and then ask the user whether or not to load it. This is the default. If the user answers P (for 'Proceed'), selective mode is turned off for any remaining patches to the current system.
:noselective	Turns off :selective.
:silent	Turns off both :selective and :verbose. In :silent mode all necessary patches are loaded without printing anything and without querying the user.
:force-unfinished	Loads patches that have not been finished yet, if they have been compiled.

load-patches returns t if any patches were loaded.

When you load a patchable system with make-system, load-patches is called automatically on that system.

si:patch-loaded-p *major-version minor-version* &optional (*system-name* "SYSTEM")
Returns t if the changes in patch *major-version.minor-version* of system *system-name* are loaded. If *major-version* is the major version of that system which is currently loaded, then the changes in that patch are loaded if the current minor version is greater than or equal to *minor-version*. If the currently loaded major version is greater than *major-version* then it is assumed that the newer system version contains all the improvements patched into earlier versions, so the value is t.

28.8.3 Making Patches

There are two editor commands that are used to create patch files. During a typical maintenance session on a system you will make several edits to its source files. The patch system can be used to copy these edits into a patch file so that they can be automatically incorporated into the system to create a new minor version. Edits in a patch file can be modified function definitions, new functions, modified defvar's and defconst's, or arbitrary forms to be evaluated, even including load's of new files.

The first step in making a patch is to *start* it. At this stage you must specify which patchable system you are making a patch for. Then you *add* one or more pieces of code from other source files to the patch. Finally you *finish* the patch. This is when you fill in the description of what the patch does; this description is what load-patches prints when it offers to load the patch. If you have any doubts about whether the patch will load and work properly, you finish it *unreleased*; then you can load it to test it but no bands can be saved containing the patch until you explicitly release it later.

It is important that any change you patch should go in a patch for the patchable system to which the changed source file belongs. This makes sure that nobody loads the change into a Lisp world which does not contain the file you were changing—something that might cause trouble.

Also, it ensures that you never patch changes to the same piece of code in two different patchable systems' patches. This would lead to disaster because there is no constraint on the order in which patches to two different systems are loaded.

Starting a patch can be done with Meta-X Start Patch. It reads the name of the system to patch with the minibuffer. Meta-X Add Patch can also start a patch, so an explicit Meta-X Start Patch is needed only infrequently.

Meta-X Add Patch adds the region (if there is one) or the current "defun" to the patch file currently being constructed. If you change a function, you should recompile it, test it, then once it works use Add Patch to put it in the patch file. If no patch is being constructed, one is started for you; you must type in the name of the system to patch.

A convenient way to add all your changes to a patch file is to use Meta-X Add Patch Changed Sections or Meta-X Add Patch Buffer Changed Sections. These commands ask you, for each changed function (or each changed function in the current buffer), whether to add it to the patch being constructed. If you use these commands more than once, a function which has been added to the patch and has not been changed since is considered "unchanged".

The patch file being constructed is in an ordinary editor buffer. If you mistakenly Add Patch something that doesn't work, you can select the buffer containing the patch file and delete it. Then later you can Add Patch the corrected version.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. This way if two people are patching a system at the same time, they do not both get the same minor version number.

After testing and patching all of your changes, use Meta-X Finish Patch to install the patch file so that other users can load it. This compiles the patch file if you have not done so yourself (patches are always compiled). It also asks you for a comment describing the reason for the patch; load-patches and print-system-modifications print these comments. If the patch is complex or it has a good chance of causing new problems, you should not use Meta-X Finish Patch; instead, you should make an unreleased patch.

A finished patch can be *released* or *unreleased*. If a patch is unreleased, it can be loaded in the usual manner if the user says 'yes' to a special query, but once it has been loaded the user will be strongly discouraged from saving a band. Therefore, you still have a chance to edit the patch file and recompile it if there is something wrong with it. You can be sure that the old broken patch will not remain permanently in saved bands.

To finish a patch without releasing it, use the command Meta-X Finish Patch Unreleased. Then the patch can be tested by loading it. After a sufficient period for testing, you can release the patch with Meta-X Release Patch. If you discover a bug in the patch after this point, it is not sufficient to correct it in this patch file; you must put the fix in a new patch to correct any bands already saved with the broken version of this patch.

It is a good principle not to add any new features or fix any additional bugs in a patch once that patch is released; change it only to correct problems with that patch. New fixes to other bugs should go in new patches.

You can only be constructing one patch at any time. Meta-X Add Patch automatically adds to the patch you are constructing. But you can start constructing a different patch without finishing the first. If you use the command Meta-X Start Patch while constructing a patch, you are given the option of starting a new patch. The old patch ceases to be the one you are constructing but the patch file remains in its editor buffer. Later, or in another session, you can go back to constructing the first patch with the command Meta-X Resume Patch. This commands asks for both a patchable system name and the patch version to resume constructing. You can simply save the editor buffer of a patch file and resume constructing that patch in a later session. You can even resume constructing a finished patch; though it rarely makes sense to do this unless the patch is unreleased.

If you start to make a patch and change your mind, use the command Meta-X Cancel Patch. This deletes the record that says that this patch is being worked on. It also tells the editor that you are no longer constructing any patch. You can undo a finished (but unreleased) patch by using Resume Patch and then Cancel Patch. If a patch is released, you cannot remove it from saved bands, so it is not reasonable to cancel it at that stage.

28.8.4 Private Patches

A private patch is a file of changes which is not installed to be loaded automatically in sequence by all users. It is loaded only by explicit request (using the function load). A private patch is not associated with any particular patchable system, and has no version number.

To make a private patch, use the editor command Meta-X Start Private Patch. Instead of a patchable system name, you must specify a filename to use for the patch file; since the patch is not to be installed, there is no standard naming convention for it to follow. Add text to the patch using Meta-X Add Patch and finish it using Meta-X Finish Patch. There is no concept of release for private patches so there is no point in using Meta-X Finish Patch Unreleased. There is also no data base recording all private patches, so Meta-X Start Private Patch will resume an existing patch, or even a finished patch. In fact, finishing a private patch is merely a way to write a comment into it and compile it.

Once the private patch file is made, you can load it like any other file.

The private patch facility is just an easy way to copy code from various files into one new file with Patch-File: T in its attribute list (to prevent warnings about redefining functions defined in other files) and compile that file.

28.8.5 System Status

The patch system has the concept of the *status* of a major version of a system. A status keyword is recorded in the Lisp world for each patchable system that is loaded. There is also a current status for each major version of each system, recorded in the patch directory file for that major version. Loading patches updates the status in the Lisp world to match the current status stored in the patch directory. The status in the patch directory is changed with **si:set-system-status**.

The status is displayed when the system version is displayed, in places such as the system greeting message (**print-herald**) and the disk partition comment.

The status is one of the following keywords:

:experimental	The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the **:initial-status** option to **defsystem**.
:released	The system is released for general use. This status produces no extra text in the system greeting and the disk partition comment.
:obsolete	The system is no longer supported.
:broken	This is like **:experimental**, but is used when the system was thought incorrectly to have been debugged, and hence was **:released** for a while.
:inconsistent	Unreleased patches to this system have been loaded. If any patchable system is in this status, **disk-save** demands extra confirmation, and the resulting saved band is identified as "Bad" in its disk partition comment.

si:set-system-status *system status* &optional *major-version*

Changes the current status of a system, as recorded in the patch directory file. *system* is the name of the system. *major-version* is the number of the major version to be changed; if unsupplied it defaults to the version currently loaded into the Lisp world. *status* should be one of the keywords above.

Do not set the current system status to **:inconsistent**. A status of **:inconsistent** is set up in the Lisp world when an unreleased patch is loaded, and once set that way it never changes in that Lisp world. The status recorded in the system's patch directory file should describe the situation where all currently released patches are loaded. It should never be **:inconsistent**.

28.8.6 Patch Files

The patch system maintains several different types of files in the directory associated with your system. This directory is specified to **defsystem** via either the :patchable option or the :pathname-default option. These files are maintained automatically, but they are described here so that you can know what they are and when they are obsolete and can be deleted.

If the :patchable option to **defsystem** had no argument, then the patch data files are stored on the host, device and directory specified as the system's pathname default. The names and types of the filenames are all standard and do not include the name of the system in any way.

If the :patchable option to **defsystem** is given an argument, this argument is a file namestring specifying the host, device and directory to use for storing the patch data files. In addition, the system's short name is used in constructing the names of the files. This allows you to store the patch data files for several systems in the same directory.

There are three kinds of files that record patch information:

* the system patch directory

  This file records the current major version number, so that when the system is recompiled a new number can be allocated.

  On Tops-20, this file has, by default, a nafne like OZ:PS:<MYDIR>PATCH.DIRECTORY, where the host, device, and directory (OZ:PS:<MYDIR>) come from the system's :pathname-default as explained above.

  If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO.PATCH-DIRECTORY, where the host, device and directory come from :patchable's argument.

* the patch directory of a major version

  There is a file of this kind for each major version of the system. It records the patches that have been made for that major version: the minor version, author, description and release status of each one.

  The data in this file are in the form of a printed representation of a Lisp list with two elements. The first is the system status of this major version (:experimental, :released, :broken or :obsolete). The second is another list with an element for each patch. The element for a patch is a list of length four: the minor version, the patch description (a string) or nil for an unfinished patch, the author's name (a string), and a flag that is t if the patch is unreleased.

  On a Tops-20, for major version 259, this file has, by default, a name like OZ:PS:<MYDIR>PATCH-259.DIRECTORY.

  If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO-259.PATCH-DIRECTORY.

* the individual patch

For each patch made, there is a Lisp source file and a QFASL file.

On a Tops-20, for version 259.12, these files have, by default, names like OZ:PS:<MYDIR>PATCH-259-12.LISP and OZ:PS:<MYDIR>PATCH-259-12.QFASL.

If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO-259-12.PATCH-DIRECTORY.

On certain types of file systems, slightly different naming conventions are used to keep the names short enough to be legal.