Program Development Utilities

Introduction to Program Development Utilities

This volume contains reference documentation about the program development utilities available to you in Genera. These utilities include:

- The System Construction Tool (SCT), which helps manage large programs that span several files. See the section "System Construction Tool".
- The Common Lisp Developer. See the section "Developing Portable Common Lisp Programs".
- The compiler. See the section "The Compiler".
- Metering utilities. See the section "Metering a Program's Performance".
- The Genera interactive Debugger, and some related debugging utilities. See the section "Debugger".
- Other debugging tools and techniques, including tracing function execution, advising a function, stepping through an evaluation, and using the Inspector and Peek utilities. See the section "Miscellaneous Debugging Techniques".
- Files and pathnames. See the section "Files".
- The Conversion Tools, which are a series of special-purpose Zmacs commands that save you time and effort in editing large pieces of code in ways that can be done semiautomatically. See the section "Conversion Tools".

System Construction Tool

Introduction to the System Construction Tool

When a program becomes large, it is often desirable to split it up into several files. One reason is to help keep the parts of the program organized, to make things easier to find. Another is that programs broken into small pieces 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 mechanism is 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.

An even more complicated factor is that files can have interdependencies. You might have a Lisp file called "defs" that contains macro definitions (or flavor 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 would have to remember this whenever you compile any of those files. Furthermore, if "defs" has changed, other files of the program might need to be recompiled because the macros might have changed and need to be reexpanded.

Finally, you might want to generate multiple versions of the program — a stable version for general users to run, another for development purposes; source control for the various versions would be nearly impossible to maintain manually.

The *System Construction Tool* (SCT) addresses the difficulties of maintaining large programs that span several files. A *system* is a set of files and a set of rules and procedures that define the relations among these files; together these files, rules, and procedures constitute a complete program.

SCT examines the creation times of the source files to determine which ones must be recompiled to produce "clean" and "coherent" product files. It can also be used to merge patches to a system. By tracking all dependencies, SCT ensures that each released system is consistent. See the section "Directories Associated with a System".

A system can be constructed out of Lisp source files or files written in other languages. Systems can also be constructed out of text files or other types of files specified by users.

Here we summarize how you define and use systems:

- You define the system, using SCT's **defsystem** special form. The definition, called a *system declaration*, specifies such information as the names of the source files (or modules) in your system and what operations should be performed on each file in what order (for example, which files should be compiled, loaded, or both, and which should be loaded first). See the section "Defining a System".

- The body of a **defsystem** declaration names the files that compose the system and consists of one or more *module* specifications. A module is one or more files or modules that should be treated as a unit. *Operations* — compiling, loading, editing, hardcopying, and the like — are applied to the module as a whole. See the section "**defsystem** Modules".

- If the system is to be made generally available to other users, you should place the system definition in its own file. (This file should contain no more than one **defsystem** form, but there can be any number of **defsubsystem**s and other forms.) You also must create two other files that make your system site-independent. The goal is to make your system run at any site, not just the one on which it physically resides. (Imagine the problems that would occur if you moved your program to another host machine, and you had to update every single pathname listed in your system definition!)

- You can perform *operations* on your system (for example, compile, edit, load, reap-protect, distribute, release, or hardcopy) by using the appropriate Command Processor commands (for example, Load System and Compile System) or Lisp functions. See the section "Loading and Compiling Systems". See the section "Functions that Operate on Systems". You can also define your own operations to perform on systems. See the section "User-defined Operations on Systems".

- The patch facility lets you make and distribute incremental fixes and improvements to your system, called *patches*, thereby avoiding recompilation or reloading of the entire system. By maintaining a patch registry, a detailed record keeping system, the patch facility allows developers to maintain multiple versions of the same system. See the section "Patch Facility".

- Various functions exist to help you find information about existing systems. See the section "Obtaining Information About a System".

Defining a System

A *system* is a set of files and a set of rules and procedures that defines the relations among these files; together these files, rules, and procedures constitute a complete program. The definition of a system (called the *system declaration*) describes these relationships and rules. Some useful, general guidelines are:

1. Use Zmacs to enter the system declaration in its own file, with a canonical type of **:lisp**. The system declaration file also contains a package declaration for the system (if necessary), which must precede the system declaration in the file. For an example of a system declaration file: See the section "System Declaration Files".

2. Create a **defsystem** form. See the section "Using **defsystem**". Wherever a pathname is required in your system declaration use logical pathnames, not physical pathnames. Logical pathnames provide a way of referring to files in a site-independent way. They also make it possible to move the sources from one machine to another within a site.

3. Assuming that you have used logical pathnames, you need to prepare two other files:

   - The *system* file

   - The *translations* file

   The system file defines a logical host, specifies the location of the system declaration file, and loads the translations file. The translations file defines the translation from logical directories on the logical host to physical directories on a physical host. See the section "Loading System Definitions".

4. Invoke a Lisp function or Command Processor (CP) command to compile, load, or perform some other operation on your system, as in

   ```
   Load System Fortran :Version Latest
   ```

   The command uses the information in the translations file to load the system declaration file, compiling this declaration file first if necessary.

Using defsystem

defsystem *system-name options* &body *body* *Function*

Defines a system called *system-name*. This name is used for all operations on the system.

The definition of a system (called the *system declaration*) describes a group of relations among a group of files that constitute at least one complete program. The declaration provides information on (1) the files that make up the system, (2) which files depend on previous operations, and (3) the characteristics of the system, for example, the package in which the source code should be read. Note: *system-name* is not package-dependent. It is only used as a string.

Interpreting or compiling the system declaration brings your system into existence for the purposes of applying operations to it. After your system declaration is loaded into the Lisp environment, Command Processor commands (such as Load System and Compile System) and corresponding Lisp functions construct a plan of operations in accordance with the properties specified in your system declaration. The system is operated on according to this plan.

options is a list of keyword and value pairs that specify global attributes of the system being defined. *body* contains the detailed specification of the parts of the system. *body* can be written using a *long-form syntax* or an abbreviated *short-form syntax*.

CLOE 386 supports all the keyword options available under CLOE 3600 SCT, at least to the extent of preserving the values supplied. However, several options are used on the CLOE 386 side, especially **:name**, **:pretty-name**, **:short-name** and **:root-pathname-for-delivery**. Option **:root-pathname-for-delivery** specifies the system root directory for the system in question.

For information on each *option*, see the section "**defsystem** Options".

The *body* of the **defsystem** form specifies information about the system and dependencies among components of the system. This information is used on the CLOE 3600 side to generate a compilation plan. The plan is contained in the file: `file.l`, and is transmitted by the `Migrate System` command.

For information on **defsystem** modules, see the section "**defsystem** Modules".

defsystem Options

options is a list of keyword and value pairs that specify global attributes of the system being defined.

:pretty-name

Specifies the name of the system for use in printing. This is the user-visible name appearing in heralds and so on. If **:pretty-name** is not specified, the default is the name of the system: *system-name*.

Example: Based on the following declaration, the herald displays the name of the registrar system as "Automatic Registration System".

```
(defsystem registrar
  (:pretty-name "Automatic Registration System"
   :short-name "Registration"
   :default-pathname "reg:reg;")...)
```

:short-name

Specifies an abbreviated name used in constructing disk label comments and patch file names for some file systems. See the section "Names of Patch Files". If the **:short-name** is not supplied, *system-name* is used.

:required-systems

Specifies the systems that are required to be loaded before this system being defined can be loaded (or compiled). If you try to load a system when a required system is missing, SCT gives an error telling you that the required system must be loaded first.

:required-systems allows you to establish dependencies among several systems without having to lock in a specific version number, as would happen if one system were a component system of the other.

```
(defsystem registrar
  (:pretty-name "Automatic Registration System"
   :short-name "Registration"
   :default-pathname "reg:reg;"
   :required-systems "Scheduling Utilities")...)
```

:default-package

Specifies the name of an existing package into which each file in the system will be loaded or compiled. This is only useful if the file has no package attribute in its mode line. (Typically, the package declaration for a system is placed in the same file as the system declaration. See the section "Defining a Package".)

It is sometimes necessary to selectively override the system's default-package, for example, when a particular system module needs to read a file into a particular package. In this case specify a different package for a particular module. See the section "**:module** Keyword Options".

On other occasions you might want to compile or load your system in a package other than the default package for purposes of debugging new versions of the system. For information, see the discussion of the **:package-override** option, see the section "**defsystem** Options".

Example: All the modules in **mailer**, except for **macros**, are compiled/loaded into the **mail** package.

```
(defpackage mail (:size 4096.))

(defsystem mailer
    (...
     :default-package mail
     ...)
  (:module defs "defs")
  (:module macros "macros" (:package special)
           (:in-order-to :compile (:load defs)))
   ...)
```

Note: Your system should be compiled and loaded in its own unique package. If your system and someone else's system both define a function called **foo**, but with different package names, the package specification will prevent name conflicts. Avoid affecting symbols in the standard Genera packages. See the section "Packages".

:package-override

Overrides all other explicit package declarations — the system default package, a package declaration for a particular module, as well as a package specified in the attribute lists of constituent files.

Commonly, this option is used when debugging a new version of a system. For example, temporarily insert the option in your **defsystem** form, reevaluate the form, and compile and test your experimental version. Do not save the system declaration file with the **:package-override** option. When you're finished debugging the new version, delete the option from the **defsystem** form and reevaluate it.

:default-pathname

Specifies a default pathname against which all other pathnames in the system are merged. Specify that part of the pathname for which you want to establish a default. You are urged to supply a logical, not a physical, pathname. See the section "Syntax for Logical Pathnames".

Here is an example.

```
:default-pathname "sys:zwei;"
```

This eliminates the need to enter the full pathname of each of the system's files. If the system's files reside in more than one directory, furnish a pathname default for the directory storing the largest number of files. Where the pathname differs from the default, specify the full pathname.

Example: "pres-type-macro" and "pres-type-fspec" are merged here into "sys:dyno-windows;pres-type-macro" and "sys:dyno-windows;pres-type-fspec" respectively. Because "character-style-pres" resides in "sys:sys2;" a full pathname specification is given.

```
(defsystem dyno-windows
  (:pretty-name "Dynamic Windows"
   :default-pathname "sys:dyno-windows;")
  (:serial (:parallel "pres-type-macro" "pres-type-fspec")
           (:parallel "sys:sys2;character-style-pres")))
```

:default-destination-pathname

Tells SCT where `.bin` or `.ibin` files should be compiled to (or loaded from). This allows you to put the `.bin` or `.ibin` files in a different place from the source files, so that multiple versions of a system can be separated.

```
(defsystem foo
    (:default-pathname "SYS:FOO;"
     :default-destination-pathname "SYS:FOO;BIN-FILES;"
     ...)
  ...)
```

:default-module-type

Specifies a keyword, which is the default type for each module. If not furnished, the default value is **:lisp**. The type specifies the nature of the inputs to the system and determines the details of what is done for each generic operation (load, edit, hardcopy) performed on the system.

Some commonly used predefined types are: **:lisp**, **:fortran**, **:pascal**, **:text**, **:font**, **:lisp-example**, and **:system**. (The **:fortran** and **:pascal** types are supplied by the corresponding optional products.) For a complete list of predefined types and operations: See the section "System Module Types and Operations".

You can also define your own types. See the section "User-defined Module Types".

It is possible to selectively override the system's default type by specifying another type for a particular module. See the section "**:module** Keyword Options".

Example: The **action** system consists of five unnamed modules of type **:fortran**.

```
(defsystem action
    (:short-name "act"
     :default-pathname "quark: code;"
     :default-package quark
     :default-module-type :fortran)
  (:serial "defs" "macros" (:parallel "things" "rooms") "parser"))
```

:maintain-journals

Controls whether or not a system is journalled. The default is **t**, to maintain journal files. Using **:maintain-journals nil** makes the system unjournalled. An unjournalled system is not patchable and has no version number, so loading it always loads the .NEWEST version of the files in the system.

:journal-directory

Specifies the location of the *journal* directory, which contains: the *system-directory* file and all the *journal subdirectories*. See the section "Directories Associated with a System".

By default, the journal directory of a system is called the subdirectory "patch" under the default pathname. For example if the default directory is

```
sys: quux;
```

then the journal directory defaults to

```
sys: quux; patch;
```

:patchable

Specifies whether you want the system to be patchable or not. It takes one argument, either **t** or **nil**. The default is **t**, meaning that the system is patchable. (See the section "Patch Facility".)

:patches-reviewed

Controls whether patching a system prompts for the name of a reviewer for the patch. The value of **:patches-reviewed** can be **t**, **nil**, or the name of a mailing list. When you start a patch for the system, Zmacs asks you to supply a patch reviewer if appropriate. The default is **nil**, not to prompt for a patch reviewer. If it is set to the name of a mailing list, when you finish the patch and select **yes** for the question "Send mail about this patch?" in the menu, this mailing list is used as the destination for your message.

:patch-atom

Controls how the patch files for a system are named. Usually, the patch-file names are derived from the short-name of a system. **:patch-atom** lets you override the short-name.

If you explicitly supply a **:patch-atom** in a **defsystem**, it should be in interchange case. (For more information on interchange case, see the section "Case in Pathnames".)

:parameters

Specifies an "argument list" for the system. When you perform some operation on a system (compile or load it, for example), you can include extra keyword arguments that will be passed on to the methods that implement operations on the modules in the system. The value of **:parameters** is a list that reads like a keyword argument list.

Example: The **:parameters** option creates the keyword **:force-package** that can be passed on to system **foo** when it is compiled.

```
(defsystem foo
    (...
     :parameters (force-package))
 ...)

(compile-system 'foo :force-package 'foo-package)
```

In this example, the user-defined parameter **:force-package** keyword is not used by **compile-system** and is passed to the lower-level callee. In this example it could be the underlying compiler appropriate to the system being defined, such as the Pascal compiler.

:version-mapping

Controls the component mapping for component systems. For example:

```
(((:compile :newest) :released) ;compiling :NEWEST loads :RELEASED
 ((:* :keyword) :number)        ;keywords snapshot the number
 ((:* :number) :number))        ;ditto for numbers
```

:initializations

Creates a list of initializations to be run immediately after the last file in the system has been loaded and before any patches are loaded. The format is **:initializations** *argument*. If *argument* is a symbol, it is interpreted as an initialization list. If it is an arbitrary form, it is evaluated.

This example specifies an initialization list:

```
:initializations *foo-init-list*
```

One of the files in your system, preferably the first one, should create the initialization list: (**defvar** *symbol* **nil**). For example:

```
(defvar *foo-init-list* nil)
```

You can add initializations to the list in your code. For example:

```
(add-initialization "init storage"
        '(setq *storage* nil) () '*foo-init-list*)
```

See the section "Introduction to Initializations".

:initial-status

Sets the initial status of the system when a new major version is created. The system's system-directory file records the status. The valid status keywords are **:experimental** (the default), **:broken**, **:obsolete**, and **:released**.

:experimental	The system has been built but has not yet been fully debugged and released to users. The software is not stable.
:released	The system is deemed stable and is released for general use.
:obsolete	The system is no longer supported.
:broken	The system does not work properly.

:bug-reports

Specifies the mailing list for bug reports for the system and the purpose of the bug mail. The system has a bug report template with the values specified to keywords in the **:bug-reports** option. All values must be strings. The acceptable keywords are:

:name Specifies the name of the bug report template. This name is used in all menus of bug report categories and is also used in the bug report's prologue describing the state of the machine on which the report was created. The default is the pretty name of the system.

:mailing-list The name of the mailing list to which the bug report will be sent. The mailing list name must be specified exactly (that is, the system does not add "Bug-" to the string you give here). The default is "Bug-*system-name*", where *system-name* is replaced by the actual system name.

:documentation The documentation associated with this bug report template. This documentation is visible in the mouse documentation line when a menu of bug report categories is displayed. The default for this option is

`Report problems in the` *pretty-name* `system.`

where *pretty-name* is the pretty name of the system.

For example,

```
     (defsystem ip-domain-server
         (:pretty-name "IP Domain Name Server"
          :bug-reports (:mailing-list "Bug-Domains")
              ...)
       ...)
```

specifies that the bug report template for the IP-Domain-Server system is called "IP Domain Name Server", that the bug reports are sent to the "Bug-Domains" list, and that the documentation string for the template is "Report problems in the IP Domain Name Server system.".

:advertised-in

Specifies a list of zero or more keywords indicating the contexts in which the system name and version number should be displayed. Valid keywords are:

Keyword	*Meaning*
:herald	The system name and version number are displayed in the herald.
:finger	The system name and version number are displayed in the Show Users listing.
:disk-label	The system name and version number are displayed in world load comments.
nil	The system name is not displayed.

The default is **:herald**. Note that for a system not to appear in the herald, you must specify **:advertised-in ()**.

:maintaining-sites

Specifies the list of sites that maintain the system. For patchable systems this declares the sites that can patch a system. It helps you to monitor versions in order to ensure that no changes are made at "unauthorized" sites. When you attempt to patch a system that is not maintained at your site, you receive a warning.

For example:

```
(defsystem experimental-file-system
    (...
     :maintaining-sites (:sgd :scrc))...)
```

The default for **:maintaining-sites** when it is undeclared is **nil**. This has the effect of allowing any site to patch the system without a warning.

:source-category

Specifies the classification of the sources for the system for distribution purposes. It is used for writing software distribution tapes. Its valid values are **:basic** (the default), **:optional**, and **:restricted**. These categories relate to distribution dumper categories. The distribution dumper writes out the sources for a system based on whether the system fits into the specified source-category. **:basic** is less restricted than **:optional**, which is less restricted than **:restricted**.

This option can also be specified as an alist, for example:

```
(:basic
  (:restricted "secrets" "more-secrets")
  (:optional "not-quite-as-secret"))
```

This says that all files are in the **:basic** category, except "secrets," "more-secrets," and "not-quite-as-secret."

:distribute-sources

Specifies whether or not the sources for the system are distributed. It is used by the distribution dumper to decide whether or not to write sources to the distribution tape. It takes the values **t** or **nil**, and its default value is **t**.

:distribute-binaries

Specifies whether or not the binary files (object files) for the system are distributed. It is used by the distribution dumper to decide whether or not to write binaries to the distribution tape. It takes the values **t** or **nil**, and its default value is **nil**.

:installation-script

Specifies the pathname of the installation script for the system. Such a script is a series of Lisp forms that load up the system and save out an incremental world with the system loaded. If you supply your system with an installation script, your users can install your software using the Install System command.

defsystem Modules

The body of a **defsystem** declaration names the files that compose the system and consists of one or more *module* specifications. A module is one or more files or modules that should be treated as a unit. *Operations* — compiling, loading, editing, hardcopying, and the like — are applied to the module as a whole.

Modules can be explicitly named or unnamed (*anonymous*). For example, in the long-form syntax,

```
(:module foo ("bar" "baz"))
```

is a named module called **foo** and contains two files — **bar** and **baz**. All operations are applied to the aggregate **foo**. The **:module** form names the aggregate (which the short-form **:parallel** would not do) and allows keyword modifiers to be associated with the module.

On the other hand, the following clause treats the files **bar** and **baz** as two separate but unnamed modules:

```
    (:serial "bar" "baz")
```

A restriction on the construction of modules is that any one file in a module cannot *depend* on the operations performed on another file in that same module. If the compilation of file "bar" depends on file "baz" having been loaded, then these files cannot be placed in the same module.

A common organizing principle for grouping files into modules is to collect together those files that perform a similar function, with the restriction that the files within the module must not depend on one another. For example, all low-level definitions (variables and macros) might be placed in the same module.

Module specifications can be expressed using a long-form syntax, a simpler short-form syntax, or a hybrid of both formulations.

- Use the short form exclusively when your system uses only default types and packages and has straightforward dependency relationships.

- Use the long form as needed when your system contains component systems (that is, when a module represents another system), non-default-type modules, explicit package specifications other than the system default package, or complicated dependency relationships.

Module Dependencies

Dependencies describe relationships among operations on modules. That is, they describe which modules depend on one another and for which operations they depend on one another. For example, modules often depend on the previous loading of other modules. The main program module in a system presumably depends on the previous loading of the low-level module definitions. Thus, the relationship of one **defsystem** module to another can be described as a hierarchy of dependencies. Within a module, however, no file can depend on any other file, but all files share the same dependencies vis-a-vis other modules.

Dependencies, which are described in the **defsystem** form, impose an order in which operations are performed on a module. The long-form module specification is needed to specify complicated dependencies among modules and operations. Note that a dependency does not guarantee that the operation will be performed, only that if the operation is requested (by the user), it will be performed in a certain order relative to other operations.

Formally defined, a module dependency states that under certain conditions, all specified operations must be performed on the indicated modules before the operation on the current module can take place.

Dependency Example 1

The following module specifications (assume they are Lisp modules) declare that:

- In order to load **main**, **defs** must be loaded first.

```
(defsystem foo
    ...
  (:module defs ("defs1"))
  (:module main ("main")
           (:in-order-to :load (:load defs))))
```

The dependency in the example applies only when **foo** is loaded, and so is called a *load-time* dependency.

Compile-time dependencies, which apply only when a compile operation is performed, are slightly more complicated.

Dependency Example 2

Assuming that the **bar** system consists of Lisp-type modules, consider the **:in-order-to** clause below. This says that **macros** depends on the compilation and loading of **defs** whenever the **bar** system is compiled. At first glance, the compilation requirement is surprising because **(:load defs)** does not mention anything about compilation. However, the system facility considers source files that can be compiled (such as Lisp or Pascal files) to have an *implicit* compile-time dependency on themselves: in order to load the files you must compile them first (if they are not already compiled).

Note: In order to prevent a Lisp file from being compiled at all, there are two predefined module types **:lisp-read-only** and **:lisp-load-only**. Declaring a module to be one of these types prevents compilation of its files.

```
(defsystem bar
    ...
    (:module defs ("defs"))
    (:module macros ("macros")
             (:in-order-to :compile (:load defs)))
    ...)
```

Dependencies can be expressed in different ways. Examples 1 and 2 declare the presence of a dependency relationship explicitly. A module can also describe a dependency implicitly using a short-form syntax.

Dependency Example 3

The **:serial** clause implies that **main** depends on **defs** and that **defs** does not depend on any other module. It also implies that operations on "defs" and "main" be performed separately and in order, even though it does not explicitly state these operations. So, if a compile operation were performed on system **foo**, first **defs** would be compiled and loaded, then **main** would be compiled and loaded.

```
(defsystem foo
    ...
  (:serial "defs" "main"))
```

In Examples 1 and 2, **defs** contains only one file, "defs", but if **defs** consisted of two files, "defs1" and "defs2", then the examples would have to be rewritten. This is relatively straightforward for Example 1; the single module specification would be edited as follows:

```
(:module defs ("defs1" "defs2"))
```

All operations would be applied to the aggregate **defs**.

Changing Example 3 requires altering the dependency to say that "defs1" and "defs2" do not depend on one another. However, **main** still depends on the prior compilation/loading of "defs1" and "defs2" but in no particular order. This dependency would be written like so:

```
(:serial (:parallel "defs1" "defs2") "main")
```

The embedded **:parallel** clause declares that the files that follow have no dependency relationship; they are operated on as a unit. The **:serial** clause still states that any operations are applied first to the **:parallel** clause, then to **main**.

Once you correctly determine (1) which files should compose a module and (2) which and how modules depend on one another, you do not have to figure out these relationships again. By constructing a plan based on the modules and their dependencies, you have finished your part of the job. Commands that operate on systems, such as Load System, will work correctly.

Short-form Module Specifications

Short-form specifications provide an abbreviated syntax for defining groups of unnamed (anonymous) modules that have a straightforward dependency relationship. All the system's files must be of the default type (defined by the **:default-module-type** option) if they are named explicitly in the short-form specification.

A short-form specification consists of a keyword, followed by one or more elements: (*keyword element1 element2 ...*)

An *element* can be another short-form or a *primary*. A primary is either a symbol, which is interpreted to be the name of a named module, or a string, which is a file spec.

The *keyword* describes the dependency relationship among the modules and can be any of the following: **:serial**, **:parallel**, **:definitions**, or **:module-group**.

Short forms can be embedded in short forms.

The meanings of the keywords are explained here.

- **:serial** means that each of the specified elements depends in some way on the preceding one. The order of specification is therefore essential.

Example: If the compile operation is performed on the system, each Lisp module in the clause shown below is compiled and then loaded in turn before the next one is compiled and loaded. The compilation and loading of **glub** depends on the previous compilation and loading of **bar**. In order to compile and load **bar**, the computer must have already compiled and loaded **foo**.

```
(:serial "foo" "bar" "glub")
```

- **:parallel** means that the specified elements do *not* depend on one another in any way; they are operated on as a group. The order of specification is therefore not important.

  Example: If the compile operation is performed, all the Lisp modules in the following clause are compiled, then all are loaded.

  ```
  (:parallel "foo" "bar" "glub")
  ```

- The syntax of the **:definitions** clause is (**:definitions** *primary element).* **:definitions** means that the *element* has a **serial** dependency on the *primary* and, in addition, it has a compile-dependency. This means that if the *primary* is compiled, the *element* must be compiled. The **:definitions** clause is useful when the *primary* contains macros that are used in the definition of the *element.*

- **:module-group** is an additional short-form syntax keyword. It provides a way to name the aggregate result of a short-form specification, so that other specifications can refer to this result. The format is: (**:module-group** *name short-form options*).

  The structure is analogous to, and the options are the same as for, the long-form specification. See the section "Long-form Module Specifications".

Short Form Syntax Examples

The following short-form syntax **defsystem** illustrates serial dependency with an embedded parallel dependency.

```
(defsystem adventure
    (:default-pathname "quark: code;"
     :default-package quark
     :default-module-type :fortran)
  (:serial "defs" "macros" (:parallel "things" "rooms") "parser"))
```

The **adventure** system consists of a sequence of modules of the type **:fortran**, compiled in the **quark** package. In the event that the system is compiled, then operations occur as follows:

1. Compile **defs**, then load it
2. Compile **macros**, then load it
3. Compile **things** and **rooms**, then load both of them
4. Compile and load **parser**

The following diagram illustrates the above dependency relationship.

```
          DEFS
           |
         MACROS
         /    \
      THINGS  ROOMS
         \    /
         PARSER
```

Both "things" and "rooms" depend on "defs" and "macros" to have been compiled and loaded, but "things" and "rooms" do not depend on each other with respect to compilation. "Parser" depends on "things" and "rooms" having been compiled and loaded but in no particular order.

The next example shows how the **:module-group** keyword is used. The **:module-group** names the result of the included short-form specification **bigstuff**, so that the **main** module can refer to it as a dependency: in order to compile **main**, first compile and load **bigstuff**.

```
(:module-group bigstuff
            (:definitions "macros" (:parallel "foo" "bar" "blech")))
(:serial bigstuff (:parallel "a" "b" "c"))
```

Long-form Module Specifications

Use the long-form module specification when your system contains component systems, non-default-type modules, explicit package specifications other than the system default package, or complicated system dependencies.

The general format of a long-form specification is:

```
(:module name inputs
         (keyword-option-1)
         (keyword-option-2)
         ... )
```

The **:module** keyword defines the module called *name*. *name* must be a symbol or **nil**; **nil** means that the module is anonymous.

inputs can be **nil** or a list of one or more of the following:

- Strings representing a file name
- Symbols representing the name of another system defined by **defsystem**

When a module consists of more than one input, the inputs must be specified as a list.

The ordering of inputs *within* a module specification is not significant. Dependencies are determined by explicit keyword directives in **:module** clauses or, failing that, by the order of the modules in the system declaration.

:module Keyword Options

:package and **:type** override the system defaults for package and types, respectively.

:package

The **:package** option takes one argument, a string, and causes operations on a module to be performed in the specified package. It overrides both the system default (specified by the **:default-package** option to **defsystem**) and any package named in the attribute lists of the system's files. It does not override the **:package-override** option to **defsystem**.

Example: The **macros** module is compiled and loaded into the **special** package. All other modules are compiled and loaded into the system default, **mail**.

```
(defsystem mailer
  (...
   :default-package mail
   ...)
  (:module defs "defs")
  (:module macros "macros" (:package special)
           (:in-order-to :compile (:load defs)))
   ...)
```

:type

The **:type** option in a module specification overrides the default module type for the system. The type specifies the nature of the inputs to that module, for example, whether it's composed of Pascal files, Lisp files, or ordinary text files, and determines the details of what is done for each generic operation (for example, load, edit, hardcopy) performed on that module. Each type has certain valid operations. You can use any of the predefined types, including **:lisp**, **:text**, **:font**, **:lisp-example**, **:system**, and so on. See the section "System Module Types and Operations".

You can also define your own module types. See the section "User-defined Module Types".

Example 1: The inputs to **adventure1** are all FORTRAN files; however, if the parser had been written in Lisp, then the **defsystem** form should be rewritten as shown in **adventure2**. **parser** is explicitly declared to be a module of type **:lisp**.

```
;;; Example 1
(defsystem adventure1
    (:short-name "advent1"
     :default-pathname "quark: code;"
     :default-package quark
     :default-module-type :fortran)
  (:serial "defs" "macros" (:parallel "things" "rooms") "parser"))

(defsystem adventure2
    (:short-name "advent2"
     :default-package quark
     :default-pathname "quark: code;"
     :default-module-type :fortran)
  (:module parser ("parser") (:type :lisp))
  (:serial "defs" "macros" (:parallel "things" "rooms") parser))
```

The **:system** type specifies the names of *component systems*, which are other systems (defined by a **defsystem** or **defsubsystem** form) that are to be included in this system. System operations are performed recursively. In the usual case, performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems.

Example 2: The moderately complicated definition of **common-lisp-internals** falls rather gracefully and readably into the serial-parallel abbreviated form. Then **common-lisp-internals** is easily made a component system of **common-lisp** by designating it as module **cl** of type **:system**. Note how neatly a compile and load dependency on **cl** is specified in the **:serial** clause.

```
;;; Example 2
(defsubsystem common-lisp-internals
    (:default-pathname "sys:clcp;"
     :default-package cli)
  (:serial "functions" "sequence-macros" "numerics"
           (:parallel "listfns" "seqfns" "hashfns")
           "type-infra" "type-supra" "type-supra2" "Type-supra3"
           "More-functions" "Stringfns" "Charfns" "Arrayfns" "Error"
           (:parallel "Iofns" "Read-print")))

(defsubsystem common-lisp
    (:default-pathname "sys:clcp;")
  (:module cl common-lisp-internals (:type :system))
  (:serial cl "Permanent-links"))
```

:in-order-to and **:uses-definitions-from** are the two main options for controlling the dependency relationships among modules.

:in-order-to

:in-order-to is the basic keyword that expresses dependency relationships among modules. The general format is (**:in-order-to** (*:operation-1 :operation-2 ...*) (*:operation module*))
Note that the first argument to **:in-order-to** can be either a symbol or a list.

Example: The following code fragment illustrates a compile-time and a load-time dependency.

```
(:module main ("main")
         (:in-order-to :compile (:load defs))
         (:in-order-to :load (:load utils)))
```

It directs that:

- If the compile operation is performed on the present module, **main**, then the **defs** module must be loaded first.
- If **main** is loaded, then the module **utils** must be loaded first.

:uses-definitions-from

:uses-definitions-from is similar to the **:in-order-to** option. The general format is (**:uses-definitions-from** *module*).

Writing **(:uses-definitions-from foo)** implies the dependency relation:

```
(:in-order-to (:compile :load) (:load foo))
```

To state it another way, **:uses-definitions-from** means that the module has a **serial** dependency on the depended-upon *module*. In addition, it requires that if the depended-upon module needs to be recompiled, then all of its dependents will be recompiled as well. Note that dependencies are transitive.

Example: Consider the following fragment, assuming that the **macros** module has been defined in the **defsystem** form.

```
(defsystem jonathan
    (:default-pathname "sys:jonathan;"
     :default-package cl)
   (:module macros ("bim" "bam" "boom"))
   (:module A ("a" "b" "c")
         (:uses-definitions-from macros))
```

The **:uses-definition-from** clause affects module **A** in the following ways:

- If **macros** is being compiled, then compile **A** whether or not it is otherwise necessary.

- If **A** is being compiled, then compile **macros**, if it needs to be compiled, first.
- If **A** is being loaded, then load **macros**, if it needs to be loaded, first.

:serial-definitions

Combines **:serial** and **:uses-definitions-from** to control the dependencies of modules.

:root-module and **:compile-satisfies-load** also control the order in which operations are performed but are far less commonly used than **:in-order-to** and **:uses-definitions-from**.

:root-module

The **:root-module** option is useful for controlling the loading and compilation of macro definitions. It has the effect of altering the normal rules of dependency. Its valid values are:

Value	*Meaning*
t	Designates the indicated module as a *root module* — a module that is always processed. **t** is the default.
nil	Indicates that the module is not a root module.

This attribute affects system building as follows: when a command or function that operates on a system (such as Compile System) constructs a step-by-step plan to operate on a system (compiling, loading, as necessary) it will not include a step for a non-root-module *unless it is explicitly depended upon by another module*. That is, compilation (or loading, and so on) of this module occurs only if a dependency exists.

Example: In the following example, the **macros** module specifies that it should not be considered a root module.

```
(defsystem rm-example
    (:default-pathname "example: code;")
  (:module defs ("defs"))
  (:module macros ("macros")
           (:in-order-to :compile (:load defs))
           (:root-module nil))
  (:module utils ("utils")
           (:uses-definitions-from macros)
           (:in-order-to :compile (:load macros))
           (:in-order-to :load (:load defs)))
  (:module main ("main")
           (:uses-definitions-from macros)
           (:in-order-to :compile (:load macros))
           (:in-order-to :load (:load utils))))
```

Assuming that the user has requested a system load, examine the load-time dependencies and note that, for purposes of loading, **macros** is *not* depended upon by any other module:

- **defs** does not depend on any other module
- **macros** depends on **defs** being loaded
- **utils** depends on **defs** being loaded
- **main** depends on **utils** being loaded

Thus, **macros** is ignored during the preparation of the system construction plan for loading **rm-example**:

1. Load **defs**
2. Load **utils**
3. Load **main**

If **:root-module** had not been specified or had been given a value of **t**, **macros** would have been loaded, according to the normal dependency rules. Since macro definitions need not be installed when a system is being loaded to be used, **(:root-module nil)** gives exactly the desired result.

When the same system is compiled, however, a load of **macros** is included in the system construction plan because **macros** is depended upon at compile-time by two modules.

- **defs** does not depend on any other module
- **macros** depends on **defs** being compiled, if necessary, and loaded
- **utils** depends on **macros** being compiled, if necessary, and loaded
- **main** depends on **macros** being compiled, if necessary, and loaded

Since macro definitions need only be loaded at compile-time, **(:root-module nil)** again gives exactly the desired result.

:compile-satisfies-load

The **:compile-satisfies-load** option, like **:root-module**, is useful for controlling the compilation and loading of macro definitions and alters the normal rules of dependency.

It has two valid values: **t** and **nil**. When set to **t**, the option declares that when a module is compiled in the current compiler environment, it should not be loaded — even if a load dependency exists, because the loading the module could destroy the current environment. The load dependency is satisfied by compiling the module.

When set to **nil**, **:compile-satisfies-load** specifies that when a module is compiled in the current compiler environment, load it if necessary. **nil** is the default.

This feature is useful because the compiler will notice entities like **defmacro**, **defsubst**, **zl:defstruct**, and **defflavor** and use them for the compilation of subsequent files without having to load them. However, if the bodies of macros (not the code produced by their expansion) call subroutines (**defun**s) in the file, then the file must be loaded in order to define those subroutines.

Example of **:compile-satisfies-load**: Assume that the user has requested a compile of the **csl-example** system.

```
(defsystem csl-example
    (:default-pathname "example: code;")
  (:module defs ("defs"))
  (:module macros ("macros")
           (:in-order-to :compile (:load defs))
           (:root-module nil)
           (:compile-satisfies-load t))
  (:module utils ("utils")
           (:uses-definitions-from macros)
           (:in-order-to :compile (:load macros))
           (:in-order-to :load (:load defs)))
  (:module main ("main")
           (:uses-definitions-from macros)
           (:in-order-to :compile (:load macros))
           (:in-order-to :load (:load utils))))
```

The compile-time dependencies expressed above indicate that:

- **defs** does not depend on any other module
- **macros** depends on **defs** being compiled, if necessary, and loaded

- **utils** depends on **macros** being compiled, if necessary, and loaded
- **main** depends on **macros** being compiled, if necessary, and loaded

If the **:compile-satisfies-load** attribute were absent or set to **nil** the system construction plan would look like this:

1. Compile **defs**
2. Load **defs**
3. Compile **macros**
4. Load **macros**
5. Compile **utils**
6. Compile **main**
7. Load **utils**
8. Load **main**

Note that because the **:compile-satisfies-load** attribute is present, the plan is amended to delete step 4.

:load-when-systems-loaded controls the loading of modules according to whether

or not required systems have been loaded.

:load-when-systems-loaded

The **:load-when-systems-loaded** option instructs SCT not to load some modules of a system when a set of required systems is not loaded. When all the required systems become loaded, SCT automatically loads the unloaded modules.

Including this option in a module has two effects:

- If the required systems are not all loaded, that module is not loaded.

- When all the required systems become loaded, SCT goes back and loads the module.

- You cannot safely patch that module, as it might not be loaded yet at the time patches are loaded.

:load-when-systems-loaded differs from the **:required-systems** option to **defsystem** in that **:required-systems** gives an error if the required systems are not present. **:load-when-systems-loaded** never gives an error.

Note that any module that contains the **:load-when-systems-loaded** option should be a *named* module, so SCT can keep track of the unloaded modules by name (since sysdcls can be reloaded).

Example:

```
(defsystem print-spooler
    (:default-pathname t)
  (:module unix-spooler ("ux-spool")
           (:load-when-systems-loaded :unix-support))
  (:serial (:parallel "defs" "macros")
           "spooler"
           unix-spooler))
```

Suppose that the system UNIX-SUPPORT is *not* loaded. When you load the PRINT-SPOOLER system, all the files in the system are loaded except for those files in UNIX-SPOOLER module (namely, UX-SPOOL). If and when you load the UNIX-SUPPORT system, the files in the UNIX-SPOOLER will get loaded.

The **:source-category**, **:distribute-sources**, and **:distribute-binaries** options supply values that override within the module the corresponding default values for the system.

:source-category

The **:source-category** option is used for writing software distribution tapes. Its valid values are **:basic** (the default), **:optional**, and **:restricted**. These categories relate to distribution dumper categories.

The distribution dumper writes out the sources for a system based on whether the system fits into the specified source-category. **:basic** is less restricted than **:optional**, which is less restricted than **:restricted**.

This module option can also be specified as an alist. See the **:source-category** option to **defsystem**.

:distribute-sources

The **:distribute-sources** option is used by the distribution dumper to decide whether or not to write sources to the distribution tape. It takes the value **t** or **nil**, and its default value is **t**.

:distribute-binaries

The **:distribute-binaries** option is used by the distribution dumper to decide whether or not to write binaries to the distribution tape. It takes the values **t** or **nil**, with a default value of **nil**.

What You Can Do With Systems

With **defsystem**, specifications of modules are intermingled with operations on modules. This stands in contrast to the syntax of **defsystem** in earlier releases in which module clauses and "transformation" clauses were separate.

This section gives a brief overview of the kinds of operations that can be applied to systems. For more details on these operations, see the referenced sections.

Seven types of predefined operations are available:

Load	Load the system into the current environment. Invoked by the Command Processor command Load System and the function **load-system**. See the section "Loading and Compiling Systems".
Compile	Compile the system, create journal files, and optionally load it into the current environment. Invoked by the Command Processor command Compile System and the function **compile-system** See the section "Loading and Compiling Systems".
Edit	Read all the files of the system into editor buffers. Invoked by the Command Processor command and the function **sct:edit-system** See the section "Functions that Operate on Systems".
Hardcopy	Hardcopy all the files in the system. Invoked by the function **sct:hardcopy-system**. See the section "Functions that Operate on Systems".
Reap-protect	Reap-protect the system. This marks all source and product files as protected from deletion. Invoked by the Command Processor command and the function **sct:reap-protect-system** See the section "Functions that Operate on Systems".
Distribute	Write the system on tape. Invoked by the Command Processor command Distribute Systems (note the plural form, since one or more systems can be written to tape at a time). See the section "Functions that Operate on Systems".
Release	Puts the **:released** keyword in the system's patch directory, and inserts a **:released** designation in the system directory file. For most operations on a system, the **:released** designator is used as the default version. Failing this, the **:latest** version is used. Invoked by the function **sct:release-system**; no corresponding Command Processor command. See the section "Functions that Operate on Systems".

For more details on operations that can be performed on the different modules types, see the section "System Module Types and Operations".

Besides the standard, predefined operations, you can define your own operations on modules. See the section "User-defined Operations on Systems".

System Module Types and Operations

This is a table of system module types and their behavior under standard operations. Note: The operation **sct:reap-protect-system** applies to all types of systems and so is not listed here. See the legend below the table to find the meaning of the various abbreviations used.

Module Type	Default file type	Compile	Load	Hard-copy	Edit	Distribute (source/product)
Lisp	:lisp	L-comp	BL	T	T	T/T
Prolog	:prolog	P-comp	BL	T	T	T/T
Ada	:ada	A-comp	BL	T	T	T/T
Fortran	:fortran	F-comp	BL	T	T	T/T
Pascal	:pascal	Pa-comp	BL	T	T	T/T
Text	:text	--	--	T	T	T/--
Font	:bfd	--	FL	N	N	T/--
System	--	*** Operate recursively ***				
Lisp-example	:lisp	--	--	T	T	T/T
Readtable	:lisp	R-comp	BL	T	T	T/T
Lisp-read-only	:lisp	--	Read-file	T	T	T/--
Lisp-load-only	:lisp	--	BL	T	T	--/T
Logical-translations	:lisp	--	Read-file	T	T	T/--
Binary-data	:bin	--	--	N	N	--/T
Text-data	:text	--	--	T	T	T/--

Legend: "i-comp" means the appropriate compiler is used, for example, "L-comp" means the Lisp compiler is invoked. "--" means this operation is meaningless on this file type. "BL" means the binary loader is invoked. "FL" refers to the font loader.

System Plan

The order in which operations are performed on the modules in a system is called the *system plan*. By default, operations occur in the order that they are defined or they are shuffled the minimum amount necessary to realize the specified constraints. These constraints are in the form of dependencies (that is, module X must be loaded before module Y is loaded).

In order to see in advance the system plan for a given system with a given operation, type the Command Processor command: Show System Plan *name-of-system* (operation). Two factors determine the system plan:

1. The order in which the modules are defined

2. The ordering constraints that derive from the dependencies that are specific to that operation

Show System Plan Command

Show System Plan *system operation keywords*

Show the system plan (the order of operations) for the specified *system* under the specified *operation*.

system	The system for which to show the plan.
operation	The operation for which to show the plan. The available operations are:

All	Count-Lines-In	Kludge-Load
Compile	Distribute	Load
Copy	Edit	Load-Patches
Copy-Toolkit-C-Files	Hardcopy	Reap-Protect
Write-Toolkit-C-Files		

keywords	:Date Checking, :Detailed, :More Processing,:Output Destination, :Version
:Date Checking	{Yes, No} Compare files against the file system. The default is No, the mentioned default is Yes.
:Detailed	{Yes, No} Whether to describe the plans for component systems. The default is No, the mentioned default is Yes.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Version	Version of the system for which to construct plans. The default is Released.

Undefining Systems

sct:undefsystem *system-name* *Function*

Removes all record of the system called *system-name* from **sct:*all-systems*** and removes all the source file name properties from the system. The effect is to make it look like a **defsystem** wasn't even done. Note: This does not undefine functions, flavors, and so on, created by loading the system.

Defining Subsystems

defsubsystem *system-name options* &body *body* *Function*

Defines a system that has no autonomous existence and is not patchable. It can only be compiled and loaded by compiling or loading its parent system. It can, however, be treated independently for some operations, like edit or hardcopy.

In a **defsystem** form, a subsystem is specified as a **:module** and is flagged with the keyword pair **(:type system)** (see the example). Subsystems are provided as a convenience for specifying groups of modules that are all in one package or directory. Subsystems have no associated component directory. Their files are journaled in the parent system's component directory.

Subsystems retain identity as systems on which you can select as a tag table in Zmacs.

In the following example of **defsubsystem**, we have not listed all the file names for each system and subsystem. The places where these names normally go are marked by ellipses.

```
(defsystem fortran
    (:default-pathname "sys: fortran;"
     :journal-directory "sys: fortran;"
     :patchable t)
  (:module macros ("macros") (:root-module nil))
  (:module language-tools (language-tools) (:type :system))
  (:module front-end (fortran-front-end) (:type :system))
  (:module back-end (fortran-back-end) (:type :system))
  (:serial macros language-tools front-end back-end))

;;; Component system definition
(defsystem language-tools
    (:default-pathname "sys: language-tools;"
     :patchable t)
  (:serial ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-front-end
    (:default-pathname "sys: fortran;")
  (:serial "tokenizer" "grammar" ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-back-end
    (:default-pathname "sys: fortran;")
  (:serial "code-generator" "optimizer" ... ))
```

In the example, **language-tools** is a patchable component system, and **fortran-front-end** and **fortran-back-end** are both subsystems.

User-defined Module Types

You can define your own module types using the function **sct:define-module-type**.

sct:define-module-type *type source-default product-default* &body *base-flavors*
Function

Defines a new module type called *type* with a *source-default* module type and a *product-default* module type.

The *base-flavors* are the previously defined module type upon which this type is built. The new *type* inherits the properties of the *base-flavors* and interprets operations like the *base-flavors* do, except in the case that special methods are defined for the *type* that override the *base-flavors* operations.

One you have defined a module type, you define methods with **defmethod** that implement the special behavior of the new module type for the standard operations: compile, load, and so on.

The purpose of this example is to define a module type called **lisp-read-only** whose sources are Lisp code but which is meant to be read and not compiled. According to the definition of **lisp-read-only** in the example, a module of this type will respond according to the definition of its base flavor **lisp-module** for all operations except loading and compiling.

```
(define-module-type :lisp-read-only :lisp nil
  lisp-module)

(defmethod (:compile lisp-read-only-module) (system-op &rest keys)
  (ignore system-op keys)
  nil)

(defmethod (:load lisp-read-only-module) (system-op &rest keys)
  (lexpr-send self :read system-op keys))
```

User-defined Operations on Systems

It is usually more useful to define your own type of system module than it is to define your own operation. However, SCT provides a facility for defining your own operations, should you need it. The macro **sct:define-system-operation** is the primary tool for this purpose.

sct:define-system-operation *operation driving-function documentation* &key (*arglist* **'(system-name &key query :confirm silent batch (version :latest) (include-components t)** &rest **keys** &allow-other-keys)) (*class* **:normal**) (*subsystems-ok* **t**) *body-wrapper* (*encache* **:both**) *Function*

Defines a manipulation called *operation* to be applied to a system, creating a function called *operation*-**system**. The *driving-function* is a closure — the operation itself at the level of what is done to a single file. Higher-level mechanisms take care of applying this operation to each file in a system. The *documentation* is another closure — an operation that prints what will be done to the file. The *arglist* specifies the arguments that are accepted by the operation. The operation can also process the keyword arguments **:query**, **:batch**, **:version**, and **:include-components**. For the meaning of these keywords: See the function **load-system**.

The *encache* argument is used by SCT to optimize calls to **fs:multiple-file-plists**. Typically, you should use **:both** if the operation needs to look at any file properties (the compile operation, for example) or **nil** if the operation does not need to look at any properties (the edit or hardcopy operations). *class* should be **:normal** for operations that construct a plan according to dependencies (for example, compile, load, edit) or should be **:simple** for operations that work on everything in the system (for example, reap-protect).

The definition of the standard hardcopy operation is shown next as an example of the use of the **sct:define-system-operation** macro.

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: SCT; Base: 10 -*-

(define-system-operation :hardcopy
  ;        input output module      keywords
  (lambda (source ignore ignore &rest ignore)
    (declare (special hardcopy:*default-text-printer*))
    (hardcopy:hardcopy-file source hardcopy:*default-text-printer*))
  ;        input output module      keywords
  (lambda (source ignore ignore &rest ignore)
    (format standard-output "~&Hardcop~[y~;ying~;ied~] file ~A"
      *system-pass* source))
  :arglist
    (system-name &key (query :confirm) silent batch
                      (include-components t) (version :newest)
                 &rest keys &allow-other-keys)
  :encache nil
  :class :normal)
```

Loading and Compiling Systems

The **load-system** and **compile-system** forms, with their Command Processor equivalents Load System and Compile System, are the means of loading and compiling systems.

Load System **Command**

Load System *system keywords*

Loads a system into the current world.

system	Name of the system to load. The default is the last system loaded.
keywords	:Component Version, :Condition, :Include Components :Load Patches, :More Processing, :Output Destination, :Query, :Redefinitions Ok, :Silent, :Simulate, :Version
:Component Version	{Released, Latest, Newest, *version-designator*} The version of any component systems to load. Released means the version designated as released in the journal file. Latest means the most recent version recorded in the journal file. Newest means to ignore the versions in the journal file and just find the newest files. The default is the version with which the system was compiled.
:Condition	{Always, Never, Newly-Compiled} Under what conditions to load each file in the system. Always means load each file. New-

	ly-compiled means load a file only if it has been compiled since the last load. The default is Newly-Compiled.
:Include Components	{Yes, No} Whether to load component systems. The default is Yes.
:Load Patches	{Yes, No} Whether to load patches after loading the system. The default is Yes.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Query	{Everything, Confirm-only, No} Whether to query before loading. Everything means query before loading each file. Confirm-only means create a list of all the files to be loaded and then ask for confirmation before proceeding. No means just go ahead and load the system without asking any questions. The default is No. The mentioned default is Everything.
:Redefinitions Ok	{Yes, No} Controls what happens if the system asks for confirmation of any redefinition warnings during the loading process. Yes means assume that all requests for confirmation are answered yes and proceed. No means pause at each redefinition and await confirmation. The default is No. The mentioned default is Yes. This allows you to start loading a system that you know will take a long time to load and leave it to finish by itself without interruption for questions such as "Warning: *function-name* being redefined, ok? (Y or N)".
:Silent	{Yes, No} Whether to turn off output to the console while the system is loading. The default is No. The mentioned default is Yes.
:Simulate	{Yes, No} Print a simulation of what compiling and loading would do. The default is No. The mentioned default is Yes.
:Version	{Released, Latest, Newest, *version-designator*} Which version number to load. Released means the version designated as released in the journal file. Latest means the most recent version recorded in the journal file. Newest means to ignore the ver-

sions in the journal file and just find the newest files. The default is Released.

Note: This command only loads a system. If you want to compile and load a system, see the section "Compile System Command".

load-system *system-name* &rest *keys* &key *(:version* **:released***) :system-branch :machine-types (:query* **:confirm***) :silent :batch (:include-components* **t***) :no-warn :reload :no-load :never-load :dont-set-version (:load-patches* **t***) :component-version* &allow-other-keys *Function*

Loads the system named by *system-name* into the current environment, according to the specified keyword options.

These are the predefined keyword options to **load-system**. Note that the allowable keywords can include those declared in the **:parameters** part of the **defsystem**.

:query	Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each and every operation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.
:silent	Takes **t** or **nil**. If **t**, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.
:no-warn	Takes **t** or **nil**. If **t**, does not query or print a redefinition warning when a function is redfined. If set to **:just-warn**, it prints a warning but does not query. Default value: **nil**.
:batch	Takes **t**, **nil**, or *pathname*. Simulate **:query :confirm :silent t :no-warn t** and collect the compiler warnings and write them to *system-name*.cwarns. If *pathname*, do the same as *t* but write compiler warnings to *pathname*. Default value: **nil**.
:reload	Takes **t** or **nil**. If **t**, reload all the binary files, even if the version in the environment is the most recent version. Default value: **nil**.
:no-load	Takes **t** or **nil**. If **t**, do not load binary files unless they are required by a specific dependency in the **defsystem**. Default value: **nil**.
:never-load	Takes **t** or **nil**. If **t**, never load any binary files, no matter what dependencies say. Default value: **nil**.
:include-components	Takes **t** or **nil**. If **t**, perform the requested system operation on component systems. Default value: **t**.
:load-patches	Takes **t** or **nil**. After the system has been loaded, implicitly perform a **load-patches** operation. Default value: **t**.

:version	Takes **:Latest**, **:Newest**, **:Released**, a number, or another designator. **:Latest** means the latest major version recorded in the journal directory. **:Newest** means ignore the journal directory and find the newest version of the files.
:dont-set-version	Takes **t** or **nil**. If **t**, do not worry about setting the version number of the system in the running world. This is an optimization used to speed up the loading of some systems such as the Logical Pathname Translations Files system. Default-value: **nil**.

See the section "Load System Command".

Compile System **Command**

Compile System *system keywords*

Compile the files that make up *system.*

system	The name of the system to compile. The default is the last system loaded.
keywords	:Batch, :Component Version, :Condition, :Copy Compile,:Include Components, :Load, :More Processing, :New Major Version, :Output Destination, :Query, :Redefinitions Ok :Silent, :Simulate, :Update Directory , :Version
:Batch	{Yes, No} Whether to save the compiler warnings in a file instead of printing them on the screen. The default is No, to print them on the screen. The mentioned default is Yes.
:Component Version	{*version-designator*} The version of any component system to load for the compilation. The default is Released.
:Condition	{Always, New-Source} Under what conditions to compile each file in the system. Always means compile each file. New-source means compile a file only if it has been changed since the last compilation. The default is New-Source.
:Copy Compile	{Yes, No, Query} For those systems where the product of the compilation is not a true binary file (notably documentation systems) and is usable on both Ivory and 3600 architectures, :Copy Compile Yes has the effect of compiling it for the other architecture. For example, if you compile a documentation system on a 3600-family machine, to "copy compile" it would have the effect of compiling it for the Ivory architecture as well. Yes does the copy compile. No does not. Query prints an explanation of what is being offered, and queries about whether to do it. The copy compile works by copying the form in the `component-dir`.

:Include Components	{Yes, No} Whether to load any component systems. The default is Yes. If :Include Components is Yes, :Component Version is used to select the appropriate version of any component systems.
:Load	{Everything, Newly-Compiled, Only-For-Dependencies, Nothing} Whether to load the system you have just compiled into the world. The default is Newly-Compiled.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:New Major Version	{Yes, No} Whether to give your newly compiled version of the system the next higher version number. The default is Yes. Giving the choice No will ask you to confirm that you really want to "prevent incrementing system major version number". (Note that if your goal is to compile a system version for another machine type, you should use the :Version keyword, instead of specifying :New Major Version No. For more information, see the section "Compiling a System for Multiple Machine Types".)
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Query	{Everything, Yes, Confirm-Only, No} Whether to query before compiling. Everything means query before compiling each file. Confirm-Only means create a list of all the files to be compiled and then ask for confirmation before proceeding. No means just go ahead and compile the system without asking any questions. The default is No. The mentioned default is Everything.
:Redefinitions Ok	{Yes, No} Controls what happens if the system asks for confirmation of any redefinition warnings during the compilation. Yes means assume that all requests for confirmation are answered yes and proceed. No means pause at each redefinition and await confirmation. The default is No. The mentioned default is Yes. This allows you to start a compilation that you know will take a long time and leave it to finish by itself without interruption for questions such as "Warning: *function-name* being redefined, ok? (Y or N)".

:Silent	{Yes, No} Whether to suppress output to the screen. The default is No, to allow output. The mentioned default is Yes.
:Simulate	{Yes, No} Print a simulation of what compiling would do. The default is No. The mentioned default is Yes.
:Update Directory	{Yes, No, *version-designator*} Whether to update the directory of the system's components. The default is Yes.
:Version	{Newest, *version-number*} Specifies the version number of a system. When this option is used, SCT checks the journals for the specified system and compiles a new version of the system using the same version of the source files. The new system must be compiled for a different machine type. See the section "Compiling a System for Multiple Machine Types". The default is Newest.

compile-system *system-name* &rest *keys* &key *:system-branch :machine-types (:query* **:confirm***) :silent :batch (:include-components* **t***) (:version* **:newest***) :component-version :no-warn :recompile :no-compile :reload :no-load :never-load (:increment-version* **t***) (:update-directory* **t***) :initial-status :update-from-world (:load-patches* **t***) (:copy-compile-p* **:query***)* &allow-other-keys *Function*

Compiles the system named by *system-name* with the specified keyword options. Note that if you are using the CLOE 386 Application Generator, you can only use these keywords: **:recompile**, **:version**, **:verbose**, **:compile-print**, and **:root-path**.

These are the predefined keyword options to **compile-system**. Note that the allowable keywords can include those declared in the **:parameters** part of the **defsystem**.

:recompile	Takes **t** or **nil**. If **t**, recompile all the source files, even if the binary file is newer than the source file. Default value: **nil**.
:no-compile	Takes **t** or **nil**. If **t**, do not compile any source files, no matter what anyone else says. This is useful in conjunction with **:update-directory t** and **:increment-version nil**, since it buys the ability to fix up the journal files after you have hand-compiled some source files. Default value: **nil**.
:increment-version	Takes **t** or **nil**. If **t**, create a new major version number. Default value: **t**.
:update-directory	Takes **t**, **nil**, or *keyword*. If **t**, update the journal files. If *keyword*, update the journal files and add a designator of *keyword* for the newly created version. Furthermore, if *keyword* is **:released**, then declare the status of the system to be released. Default value: **t**.

:initial-status	Takes *keyword*. Declare the initial status of the system to be *keyword*. Default value: **:experimental**.
:query	Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each and every operation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.
:silent	Takes **t** or **nil**. If **t**, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.
:no-warn	Takes **t** or **nil**. If **t**, does not query or print a redefinition warning when a function is redfined. If set to **:just-warn**, it prints a warning but does not query. Default value: **nil**.
:batch	Takes **t**, **nil**, or *pathname*. Simulate **:query :confirm :silent t :no-warn t** and collect the compiler warnings and write them to *system-name*.cwarns. If *pathname*, do the same as *t* but write compiler warnings to *pathname*. Default value: **nil**.
:reload	Takes **t** or **nil**. If **t**, reload all the binary files, even if the version in the environment is the most recent version. Default value: **nil**.
:no-load	Takes **t** or **nil**. If **t**, do not load binary files unless they are required by a specific dependency in the **defsystem**. Default value: **nil**.
:never-load	Takes **t** or **nil**. If **t**, never load any binary files, no matter what dependencies specify. Default value: **nil**.
:include-components	Takes **t** or **nil**. If **t**, perform the requested system operation on component systems. Default value: **t**.
:load-patches	Takes **t** or **nil**. After the system has been loaded, implictly perform a **load-patches** operation. Default value: **t**.
:version	Takes **:newest** or *n*, where *n* is the version number of a system. Used with **:increment-version nil** to compile identical systems for multiple types of machines. For further information, see the section "Compiling a System for Multiple Machine Types".
:copy-compile-p	Takes **t**, **nil**, or **:query**, which is the default. For those systems where the product of the compilation is not a true binary file (notably documentation systems) and is usable on both Ivory and 3600 architectures, **:copy-compile-p** allows "compilation" for the other machine type by copying the form in the `component-dir`. This saves the necessity of doing the compilation over for the other machine type. See the section "Compiling a System for Multiple Machine Types".

If you are using the CLOE 386 Application Generator, **compile-system** compiles the corresponding system, according to the compilation plan contained in the `component.l` file and the values of the keyword options. If no keywords are specified for the **:released** version of system *name*, components unlisted in the compilation plan as already compiled will be compiled. This will be executed by using **compile-file** to files with the same name, and of type `b`. The compilation plan is updated to include these newly compiled files. Other versions can be compiled if specified in the **:version** option. If the **:recompile** option is not **nil**, all files will be compiled. If the **:verbose** switch is true, then information about the compilations will be printed as the files are compiled. Otherwise, the compilations are silent. Note that this function is only available on the 386 side.

An error is signalled if no system is found. If this due to incorrect information or unavailablity, you will have an opportunity to supply the missing information.

```
(compile-system "frobozz" :recompile t :version :latest)
```

Compiling a System for Multiple Machine Types

The System Construction Tool (SCT) incorporates capabilities that allow you to easily compile systems for different machine types (such as the 3600 and Ivory) from the same set of sources. It also provides functions that allow you to share patches for systems compiled for more than one machine type.

The Compile System command and **compile-system** function accept the :**version** option, which takes a value *N* or **Newest** and specifies a system's version number.

You can use **:version** *N* to compile systems for different machine types in this way: First, compile the system on one machine type. From the second type of machine, recompile the system specifying the version number of the system you have just compiled. Specifying the **:version** option makes SCT compile the same version of each source file that was compiled when the system was compiled before, instead of compiling the newest version of each source file. It causes SCT to look at the journals and compile the same version of the system from the same sources for the new machine type.

For example, compile the system `Sample` on a 3600-series machine with:

```
Compile System Sample
```

Assuming that the resulting version was numbered 5, type one of the following lines on the second, Ivory-based machine to compile `Sample` for that machine type:

```
Compile System Sample :Version 5
```

or

```
(compile-system Sample :version 5)
```

The same version of the system `Sample` is now compiled for both machines.

For the 3600-series, the resulting binary files are identified by the extension `.bin`. For Ivory-based systems, binary files are identified by the extension `.ibin`.

Maintaining Parallel Systems for Multiple Machine Types

Use the following functions to maintain parallel systems for multiple machine types.

sct:compile-uncompiled-patches *systems* &optional *(machine-type* **sct:*local-machine-type*** *)* *Function*

Compiles those patches in *systems* that are uncompiled for the current machine type. For instance, if you create a patch for a system from a 3600, the patch is compiled only for the 3600 machine type. To compile the patch for another machine type, such as the MacIvory, use **sct:compile-uncompiled-patches** from a MacIvory.

Using the form `(sct:compile-uncompiled-patches)` compiles patches for all systems. To specify some systems, use a form like:

```
(sct:compile-uncompiled-patches :zmail :lmfs :my-system)
```

See the function **sct:recompile-changed-patches** for related information.

sct:recompile-changed-patches *systems* &optional *(machine-type* **sct:*local-machine-type*** *)* *Function*

Recompiles those patches in *systems* whose Lisp files are newer than their binary files. For example, if you recompile a patch for a system on a 3600, you can use **sct:recompile-changed-patches** on an Ivory-based machine to recompile it for an Ivory.

Using the form `(sct:recompile-changed-patches)` compiles patches for all systems. To specify some systems, use a form like:

```
(sct:recompile-changed-patches :my-system)
```

See the function **sct:compile-uncompiled-patches** for related information.

Loading System Definitions

Once you have written a large program and defined it as a system, use the function **load-system** (or the Command Processor (CP) commands Compile System and Load System) to compile and load the system (plus any patches related to it).

For information about the function **load-system**, see the function **load-system**. For information about the Load System and Compile System Command Processor (CP) commands:

- See the section "Compile System Command".
- See the section "Load System Command".

Loading System Definitions Using Logical Pathnames

So that your system definition can use logical pathnames, create these files:

System File	This file (named sys:site;*system-name*.system) contains a pointer to the system declaration file (defined within this section). The system file enables the **load-system** function to find and load your system (so that others can easily use it). If yours is an experimental or private system, you may not require a separate sys:site;*system-name*.system file. Instead, compile the **defsystem** in an editor buffer (or put a form that loads the system declaration in your initialization file). For more information about system files, see the section "System Files".
Translations File	This file (named sys:site;*logical-host*.translations) describes each logical host defined in the current world. When you transport a world load to a new site, the translations file is reloaded from the site's sys:site; directory, and the site's logical pathnames are mapped into the appropriate, corresponding set of physical pathnames. For more information about translations files, see the section "Translations Files".
System Declaration File	This file (named *logical-host:logical-directory;system-name*.lisp or *logical-host:logical-directory*;sysdcl.lisp) contains the **defsystem** for a system. For more information about system declaration files, see the section "System Declaration Files".

System Files

System files (named sys:site;*system-name*.system) enable the function **load-system** (which looks in the sys:site; logical directory) to identify a system name that is undefined in your environment. For example, if you type the following at a Lisp Listener:

```
Load System graphic-lisp
```

the **load-system** function looks for the file SYS:SITE;GRAPHIC-LISP.SYSTEM.

The system file must contain this form:

(**sct:set-system-source file** *"system-name"*
"logical-host:logical-directory;system-declaration-file")

If a logical host other than "sys" is needed, use the additional form:

(**fs:make-logical-pathname-host** *"logical-host"*)

For example, for the system **graphic-lisp**, the file SYS:SITE;GRAPHIC-LISP.SYSTEM contains the following:

```
;;; -*- Mode: LISP; Package: USER -*-
```

```
(fs:make-logical-pathname-host "graphic-lisp")
(sct:set-system-source-file "graphic-lisp"
                            "graphic-lisp: graphic-lisp; glisp-sys")
```

The first form, a call to **fs:make-logical-pathname-host**, defines a logical host. Commonly, the *"logical-host"* has the same name as *"system-name"*. **fs:make-logical-pathname-host** also loads the translations file, which defines the translation from logical pathnames to physical pathnames.

Make sure that **fs:make-logical-pathname-host** is the first form in the file, as the second form depends on having the logical host defined already. **sct:set-system-source-file** specifies the logical pathname of the system declaration file. **load-system**, after referring to the translation definitions, loads the system declaration file.

Translations Files

Translations files (named sys:site;*logical-host*.translations) define the translations from logical directories (on the logical host) to physical directories (on a physical host). A translations file looks like this:

```
(fs:set-logical-pathname-host "logical-host"
 :physical-host "host-name"
 :translations '(("logical-directory;" "physical-directory"))
```

For example, for the system **graphic-lisp**, the file graphic-lisp.translations contains the following:

```
;;; -*- Mode: LISP; Package: USER -*-

(fs:set-logical-pathname-host "graphic-lisp"
  :physical-host "puzzle"
  :translations '(("graphic-lisp;" ">sys>graphic-lisp>")))
```

Notice the translations list in the previous example; the list consists of two-element lists (strings) that represent the logical directories specified in the system declaration and their associated physical directories.

To specify a hierarchy of directories (instead of a one-to-one translation), change the translations list as follows (where the double asterisk [**] means include all subdirectories of "graphic-lisp;"):

```
  :translations '(("graphic-lisp;**;" ">sys>graphic-lisp>**")))
```

In simple applications, where all system files are stored in one directory, it is common for the logical directory name (for example, "graphic-lisp;") to be the same as the system name ("graphic-lisp").

The sys:site;*logical-host.translations* file is loaded by **fs:make-logical-pathname-host**. Use **load-patches** to reload the file in the event that it has been changed.

System Declaration Files

System declaration files contain a **defsystem** form for defining your system and, if you need one, a **zl:defpackage** form (which must precede the system declaration). Any user-defined **defsystem** transformations should also precede the system declaration within this file.

Currently, a system declaration file can contain no more than one **defsystem** form, although any number of **defsubsystem** forms can appear in the file. This constraint exists because the system declaration can potentially be reloaded for each **defsystem** present (a situation difficult for the System Construction Tool (SCT) to resolve).

More information is available about **defsystem**, **zl:defpackage**, and **defsubsystem**.

- See the function **defsystem**.
- See the special form **defpackage**.
- See the function **defsubsystem**.

Here is a sample system declaration file:

```
;;; -*- Mode: LISP; Package: CL-USER; -*-
;;; Fortran package specifications
(defpackage fortran-global
  (:use)
  (:nicknames fortran for)
  (:prefix-name "FORTRAN")
  (:colon-mode :external)
  (:size 200))

(defpackage fortran-system
  (:use)
  (:nicknames for-sys)
  (:prefix-name "FOR-SYS")
  (:colon-mode :external)
  (:size 200))

(defpackage fortran-compiler
  (:use fortran-system fortran-global symbolics-common-lisp)
  (:nicknames for-compiler)
  (:prefix-name "FOR-COMPILER")
  (:colon-mode :external)
  (:size 1500))
```

```
(defpackage fortran-user
  (:use fortran-global symbolics-common-lisp)
  (:nicknames for-user)
  (:prefix-name "FOR-USER")
  (:relative-names-for-me (fortran-global user))
  (:size 2000))

;;; System definition using SCT
(defsystem fortran
    (:default-pathname "sys: fortran;"
     :journal-directory "sys: fortran;"
     :patchable t)
  (:module macros ("macros") (:root-module nil))
  (:module language-tools (language-tools) (:type :system))
  (:module front-end (fortran-front-end) (:type :system))
  (:module back-end (fortran-back-end) (:type :system))
  (:serial macros language-tools front-end back-end))

;;; Component system definition
(defsubsystem language-tools
    (:default-pathname "sys: language-tools;")
  (:serial ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-front-end
    (:default-pathname "sys: fortran;")
  (:serial "tokenizer" "grammar" ... ))

;;; Subsystem definition (non-patchable)
(defsubsystem fortran-back-end
    (:default-pathname "sys: fortran;")
  (:serial "code-generator" "optimizer" ... ))
```

Since you specify the pathname explicitly with the form **sct:set-system-source-file** (inside the system file), system declaration filenames do not require an exact format. Typically, though, the logical pathname for them is *logical-host:logical-directory;system-name*.

Give the system declaration source file the lisp canonical file type. When you call the **load-system** function, **sct:set-system-source-file** loads the system declaration file (.newest version).

Loading System Definitions Using Physical Pathnames

To load system definitions that use physical pathnames, specify the name of the system and the pathname of the system declaration file in an **sct:set-system-source-file** form. Have your init file evaluate the form (or type the form at a Lisp

Listener) prior to calling the function **load-system**. For more information about the function **sct:set-system-source-file**, see the section "Lisp Functions for Loading System Definitions".

Note: Logical pathnames enable you to change only translations (instead of editing all of your files to contain new file names) when moving programs between hosts (that use different operating systems, for example). Use logical pathnames — rather than physical pathnames — to ensure the site-independence of your systems.

Lisp Functions for Loading System Definitions

The Lisp functions described within this section are especially useful for site maintainers who make and distribute worlds.

sct:set-system-source-file *system-name source-file* *Function*

Specifies the pathname (*source-file*) of a file containing the system declaration for a system called *system-name*. Although **sct:set-system-source-file** can be used in two ways, Symbolics recommends the first.

1. When your system is defined with logical pathnames, include the **sct:set-system-source-file** form in the file sys:site;*system-name*.system. **load-system** loads the sys:site;*system-name*.system file the first time you attempt to load the system.

2. When your system is defined using physical pathnames, have your init file evaluate the **sct:set-system-source-file** form (or type the form at a Lisp Listener) prior to calling **load-system** or to using the Load System or Compile System Command Processor (CP) commands. *Source-file* is loaded the first time you compile or load your system.

More information is available about using the function **sct:set-system-source-file** in system files. See the section "System Files".

fs:set-logical-pathname-host *logical-host* &key *:physical-host :translations :rules :site-rules (:no-translate* **t**) *:no-search-for-shadowed-physical* *Function*

Creates a logical host named "*logical-host*" if one does not already exist. This form appears in `sys:site;`*logical-host*`.translations` files. It establishes the translations of logical directories on *logical-host* to physical directories on one or more physical hosts. The machine specified by the *:physical-host* keyword serves as the default physical host.

The *:translations* keyword specifies the list of translations from logical to physical directories.

- For more information about translations lists: See the section "Translations Files".
- For the format of the lists and the translation rules: See the section "Pathname Translation".
- For a discussion of the *:rules* and *:site-rules* keywords: See the section "Defining a Translation Rule".

If *no-translate* is **nil**, the translation of every interned logical pathname is checked. Properties are copied from the old physical pathname to the the new one, and logical pathnames that now have no corresponding physical pathnames are uninterned.

If *no-translate* is not **nil** or not supplied, this mapping is suppressed, and some physical pathnames might not get the properties of the logical pathname. This is not normally of any consequence, so *no-translate* defaults to **t**.

The argument *no-search-for-shadowed-physical* (default **nil**) means to look only in the existing pathname hosts for a host with the same name as the logical host. This saves time by not asking the namespace server whether the name of the newly defined logical host conflicts with the names of any physical hosts, but it prevents you from seeing the following warnings:

```
Warning: the host ~A must now be referred to as ~A: in pathnames,
                           since ~A is now a logical pathname host.
                           This affects ~[no~:;~:*~D~] extant pathnames.

Warning: the nickname ~A: for the physical host ~A
                              will now refer instead to the
                              logical pathname host ~A.
                              Use ~A: in pathnames.
```

For more information about `sys.translations` files, see the section "Pathname Translation". Also see the section "Translations Files".

fs:make-logical-pathname-host *name* &key *no-search-for-shadowed-physical*
Function

Defines *name* (a string or symbol) to be the name of a logical pathname host. *Name* should not conflict with the name of any existing host, logical or physical. An **fs:make-logical-pathname-host** form often appears in the file `sys:site;`*system-name*`.system`.

fs:make-logical-pathname-host loads the file `sys:site;`*name*`.translations`. **load-patches** checks the translations file for each logical host that is defined in the current world; if any translations file has been changed it is reloaded (if and only if no specific systems are specified in its arguments).

The argument **:no-search-for-shadowed-physical** (default **nil**) means to look only in the existing pathname hosts for a host with the same name as the logical host.

This saves time by not asking the namespace server whether the name of the newly defined logical host conflicts with the names of any physical hosts, but it prevents you from seeing the following warnings:

```
Warning: the host ~A must now be referred to as ~A: in pathnames,
                              since ~A is now a logical pathname host.
                              This affects ~[no~:;~:*~D~] extant pathnames.

Warning: the nickname ~A: for the physical host ~A
                                will now refer instead to the
                                logical pathname host ~A.
                                Use ~A: in pathnames.
```

Note: fs:add-logical-pathname-host is an obsolete name for this function.

More information is available about using the function **fs:make-logical-pathname-host** in system files. See the section "System Files".

Functions that Operate on Systems

Besides being loaded and compiled, systems can be edited, hardcopied, reap-protected, released, and copied. You can also set the system status and designate a system version. This section describes those operations. For information on distributing systems, see the section "Distributing Systems".

Setting the system status and version are invoked by functions. The other operations can be invoked by either functions or Command Processor commands.

sct:edit-system *system-name* &rest *keys* &key *:machine-types (:query* **:confirm***) :silent :batch (:include-components* **t***) (:version* **:newest***)* &allow-other-keys *Function*

Edits all the source files of the system called *system-name* according to the specified keyword options. This can also be accomplished with the Command Processor command Edit System or the Zmacs command `(m-X)` Edit System Files.

These are the keyword options to **sct:edit-system**.

:machine-types
Whether the operation on the system(s) should be for 3600 Family machines, Ivory machines, or all machine types. Valid values are :|3600| and :IMACH. The default for **sct:edit-system** and **sct:hardcopy-system** is the type of the current machine. The default for **sct:reap-protect-system** is all machine types.

:query	Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each suboperation, such as whether to load the system declaration file. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the suboperations and then ask for confirmation. Default-value: **:confirm**.

:silent Takes **t** or **nil**. If **t**, perform all suboperations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.

:include-components
Takes **t** or **nil**. If **t**, perform the requested system operation on component systems. Default value: **t**.

sct:hardcopy-system *system-name* &rest *keys* &key *:machine-types (:query* **:confirm***) :silent :batch (:include-components* **t***) (:version* **:newest***) (:hardcopy-device* **hardcopy:*default-text-printer****) :title (:copies* **1***) :landscape-p :page-headings :body-character-style :heading-character-style* &allow-other-keys *Function*

Hardcopies the source files of the system specified by *system-name* according to the specified keyword options.

These are the keyword options to **sct:hardcopy-system**.

:machine-types
Whether the operation on the system(s) should be for 3600 Family machines, Ivory machines, or all machine types. Valid values are :|3600| and :IMACH. The default for **sct:edit-system** and **sct:hardcopy-system** is the type of the current machine. The default for **sct:reap-protect-system** is all machine types.

:query Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each suboperation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.

:silent Takes **t** or **nil**. If **t**, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.

:include-components
Takes **t** or **nil**. If **t**, perform the requested system operation on component systems. Default value: **t**.

:version Takes a version number or indicator and performs the requested system operation on that version. The default is **:newest**.

:hardcopy-device
Takes a printer name, the device to which to send the hardcopy request. The default is the value of **hardcopy:*default-text-printer***.

:copies Takes a number, the number of copies to make. The default is **1**.

:landscape-p Takes **t** or **nil**. If **:landscape-p** is **t**, the printing is done in landscape mode.

sct:reap-protect-system *system-name* &rest *keys* &key *(:query* **:confirm***) :silent :batch (:include-components* **t***) (:version* **:latest***) (:machine-types* **:all***) (:reap-protect* **t***)* &allow-other-keys *Function*

Reap-protects all the files in the system specified by *system-name* according to the specified options.

These are the keyword options to **sct:reap-protect-system**.

:query	Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each suboperation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.
:silent	Takes **t** or **nil**. If **t**, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.
:reap-protect	Takes **t** or **nil**. If **t**, reap-protect the files. If **nil** un-reap-protect them. Default value: **t**.
:machine-types	Whether the operation on the system(s) should be for 3600 Family machines, Ivory machines, or all machine types. Valid values are :|3600| and :IMACH. The default for **sct:edit-system** and **sct:hardcopy-system** is the type of the current machine. The default for **sct:reap-protect-system** is all machine types.
:include-components	Takes **t** or **nil**. If **t**, perform the requested system operation on component systems. Default value: **t**.

sct:set-system-status *system new-status* &optional *major-version only-update-on-disk* *Function*

Changes the status of the specified *system* to *new-status*. Valid values of *new-status* are: **:experimental**, **:released**, **:frozen**, **:obsolete**, and **:broken**. Note that declaring a system to have a status of **:released** is not the same as designating a system as being the **:released** version. When *only-update-on-disk* is **t**, this does not update in-core datastructures if the system has not been loaded.

sct:designate-system-version *system designator major-version* &optional *only-update-on-disk* *Function*

Adds a version designator of *designator* to the specified *major-version* of the *system*. For example, if you want to claim that version 29 of the Tools system is to be called the in-house version:

```
(sct:designate-system-version 'Tools :in-house 29.)
```

If *major-version* is **nil**, *designator* is removed. When *only-update-on-disk* is **t**, this does not update in-core datastructures if the system has not been loaded.

sct:release-system *system major-version* &key *:only-update-on-disk (:reap-protect* **t***)* *Function*

Puts the **:released** keyword in the *system*'s patch directory, inserts a **:released** designation in the system directory. Invoked by the function **sct:release-system**; no corresponding Command Processor command. Releases the system specified by *system-name* with the specified *major-version* number. When **:only-update-on-disk** is **t**, it does not update in-core datastructures if the system has not been loaded.

When **:reap-protect** is **t**, it sets the reap-protect bit for all the files that make up the system.

Note: This operation is equivalent to a **sct:set-system-status** operation followed by a **sct:designate-system-version**.

sct:copy-system *system-name* &rest *keys* &key *:system-branch (:query* **:confirm***) :silent :batch (:include-components* **t***) (:version* **:newest***) (:machine-types* **:all***) :destination :copy-patches (:copy-journals* **sct:copy-patches***) (:copy-sources* **:use-system-value***) (:copy-binaries* **:use-system-value***) (:copy-creation-date* **t***) (:copy-author* **t***) :create-directories :clobber :never-clobber :flatten-files :compress-files* &allow-other-keys *Function*

Moves the files of a system from one place to another. *system-name* is the system to move.

:destination

Specifies where to copy the system files. The **:destination** keyword is required. It can take one of the following values:

- A non-wild filename (host, device, and directory only)

  Copies all files in the given system into this directory. In this case, the source directory structure is flattened so that all files appear in the same result directory regardless of the shape of the directory structure in which they originally resided. (Note that in this case, if files with the same name exist in two different source directories, name collisions can occur.)

  For example:

  ```
  (sct:copy-system "Physics" :destination "Cobalt:>Marie>physics>")
  ```

  If the Physics system contained these files:

  ```
  Hydrogen:>Albert>toys>macros.lisp
  Helium:>Isaac>general>utilities.lisp
  ```

  they would be copied to:

```
Cobalt:>Marie>physics>macros.lisp
Cobalt:>Marie>physics>utilities.lisp
```

- A wild filename (host, device, and directory only)

  Constructs a matching set of wildcards to match against the directory structure in the source, and copies all files in the source into the specified pathname.

  It is an error if the wildcards in the destination do not match the shape of the source structure. For example, in the example shown below, "Cobalt:>Marie>physics>*>*>" would be an appropriate alternative for this input data, but "Cobalt:>Marie>physics>*>" would not (because there are two levels of directory structure in the source files). In general, it is best to use a **:wild-inferiors** designator, such as the "**" notation for LMFS pathnames, if at all possible.

  For example:

  ```
  (sct:copy-system "Physics" :destination "Cobalt:>Marie>physics>**>")
  ```

  If the Physics system contained these files:

  ```
  Hydrogen:>Albert>toys>macros.lisp
  Helium:>Isaac>general>utilities.lisp
  ```

  they would be copied to:

  ```
  Cobalt:>Marie>physics>Albert>toys>macros.lisp
  Cobalt:>Marie>physics>Isaac>general>utilities.lisp
  ```

- A translation alist

  This translation alist has the same format as the **:translations** argument to **fs:set-logical-pathname-host**. Entries are tried in succession until the first match. When an entry matches an element in the car of some entry, the destination pathname is the cadr of that entry.

  For example:

  ```
  (SCT:COPY-SYSTEM
    "Physics"
    :DESTINATION '(("Hydrogen:>**>*.*.*"
                    "Cobalt:>Marie>Hydrogen>**>*.*.*")
                   ("Helium:>Isaac>**>*.*.*"
                    "Cobalt:>Marie>Isaac>**>*.*.*")
                   ("Helium:>**>*.*.*"
                    "Cobalt:>Marie>Helium-Other>**>*.*.*")))
  ```

If the Physics system contained these files:

```
Hydrogen:>Albert>toys>macros.lisp
Helium:>Isaac>general>utilities.lisp
```

they would be copied to:

```
Cobalt:>Marie>Hydrogen>Albert>toys>macros.lisp
Cobalt:>Marie>Isaac>general>utilities.lisp
```

:copy-sources
If **t**, then sources are copied. If **nil**, they are not. The default is **:use-system-value**, which means that the defaults for distribution as determined in the system declaration (and the defaults for the Distribute System command) are used.

:copy-binaries
If **t**, then binaries are copied. If **nil**, they are not. The default is **:use-system-value**, which means that the defaults for distribution as determined in the system declaration (and the defaults for the Distribute System command) are used.

:flatten-files
Reserved for future use.

:machine-types
Takes a list of machine types or **:all**. The default is **:all**. The possible machines types are **:|3600|** and **:imach**.

Distributing Systems

Distribute Systems Command

Distribute Systems *systems-and-versions-pairs keywords*

Writes systems to tape for distribution. If you do not specify a system, the Distribute Systems Activity window is selected for you. Distribute Systems lists the systems to write to tape, and asks if you want to perform the Distribute Systems operation. Type `Y` for Yes, `N` for No, `Q` for Quit, or `S` for Selective.

If you choose Selective, each file is listed, and you are asked if you want to distribute that particular file. You can select as many files as you want. After you enter this information, you are prompted for a tape specification, if you did not specify one already.

systems-and-versions-pairs
A list consisting of items separated by commas, each item being a system name followed by a space and a version number.

keywords	:Compress Files, :Default Version, :Distribute Patch Sources, :File Types, :Full Length Tapes, :Include Components, :Include Patches, :Included Files Checkpoint, :Machine Types, :Menu, :More Processing, :Output Destination, :Query, :Source Category, :Tape Spec, :Use Cached Checkpoint, :Use Disk
:Compress Files	{Yes, No} Whether to compress the files when writing them to tape. The default is No. The mentioned default is Yes.
:Default Version	{Released, Latest, Newest, *version-designator*} Version of the system to distribute if not individually specified in Systems. The default is Released.
:Distribute Patch Sources	{Yes, No} Whether to include patch sources for system patches. The default is No. The mentioned default is Yes.
:File Types	{Sources, Binaries, Both, Patches-Only, Default} What file types to distribute. The default leaves it to the specifications in individual **defsystem** forms.
:Full Length Tapes	{Yes, No} Write all tracks of the tape. Use this *only* if you are sure that you don't have to read the tape on a 3600 Cipher drive. The default is No. The mentioned default is Yes.
:Include Components	{Yes, No} Whether to include any component systems of the systems being distributed. The default is Yes.
:Include Patches	{Yes, No, Selective} Whether to include the patch files for the systems being distributed. The default is Yes. If you include patch files and also distribute source files, the source file corresponding to the patch level, not necessarily the source used for the compilation, is the one included on the tape. For example, suppose you have a system that includes the file blue.lisp.1. You put this file in an editor buffer, modify the code, make a patch file, and then save the buffer with the altered code to blue.lisp.2. When you use Distribute Systems, blue.lisp.2 is distributed, but not blue.lisp.1.
:Included Files Checkpoint	{Patch, Release, None} Limit distributed files to those after this patch number or release name, or None (do not limit). The default is None.
:Machine Types	{3600, Imach, All} Specifies whether the systems to distribute should be for 3600-family machines, Ivory-based machines, or all machine types. The default is to distribute systems for all machine types.
:Menu	{Yes, No} Whether to use a menu interface to specify details of the distribution. Choosing Yes presents a Distribute Systems

frame to select which files are distributed. For detailed information about this frame, see the section "Distribute Systems Activity". The default is No. The mentioned default is Yes.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Query {Everything, Yes, Confirm-Only, No} Whether to ask about distributing each file. The default is Confirm-Only. The mentioned default is Everything. Everything queries you about each file being distributed, and again for each system being distributed. Yes queries you about each file being distributed. Confirm-Only queries you about each system being distributed. No does not query.

For queries about individual files, the possible responses are:

Y	Yes, distribute this file.
N	No, do not distribute this file.
I	Include the remaining files in this system.
B	Bypass (do not include) the remaining files in this system.
D	Directory. Show the directory containing this file.
E	Edit this file.
S	Source compare this file.

For queries about systems, the possible responses are:

Y	Yes, distribute all files listed in this system.
N	No, do not distribute any files listed.
Q	Quit. Do not distribute any files listed in this system.
S	Selective. Query about each file in this system individually.

:Source Category {Basic, Optional, Restricted, Optional-only, Restricted-only} Indicates which source category or categories to write to tape for distribution. The default is Basic.

:Tape Spec The specification for the tape. The default is the default drive on the local machine. For more information about tape specifications, see the section "Tape Specifications".

:Use Cached Checkpoint
: {Yes, No} Use the last checkpoint gathered for this system. Using the cached checkpoint information, if there is any, saves time. But it is safe to use only if you are sure no more patches have been made since the cached information was computed. The default is No. The mentioned default is Yes.

:Use Disk
: On all machines other than a MacIvory, the choices are {Yes, No}. If Yes, the input is written to disk as a special file that is an image of what would be written to tape. When writing to disk, the distribution plan is not divided into parts according to any size limit. You can use this either to distribute on disk, or when you are preparing a distribution and want to see what files would be written to tape. The default is No. The mentioned default is Yes.

  On a MacIvory, the choices are {Tape, Disk, Floppy}. Disk indicates the hard disk, and Floppy indicates the floppy disk. These two values also write a special file that is an image of what would be written to tape. Additionally, when a floppy disk is used, the size limit for a "reel" is set to the capacity of the floppy (that is, 800 Kbytes). Tape means to write to tape; this is the default.

Distribute Systems Activity

When you use the Distribute Systems command, and specify the keyword :Menu with the value Yes, you invoke the Distribute Systems activity. This activity enables you to define and edit the specifications for one or more systems to be written on a distribution tape. You can add specifications for new systems one at a time, make changes to the details of the existing systems, or delete them entirely. You can also access the Distribution Systems activity by using the command Select Activity Distribution System.

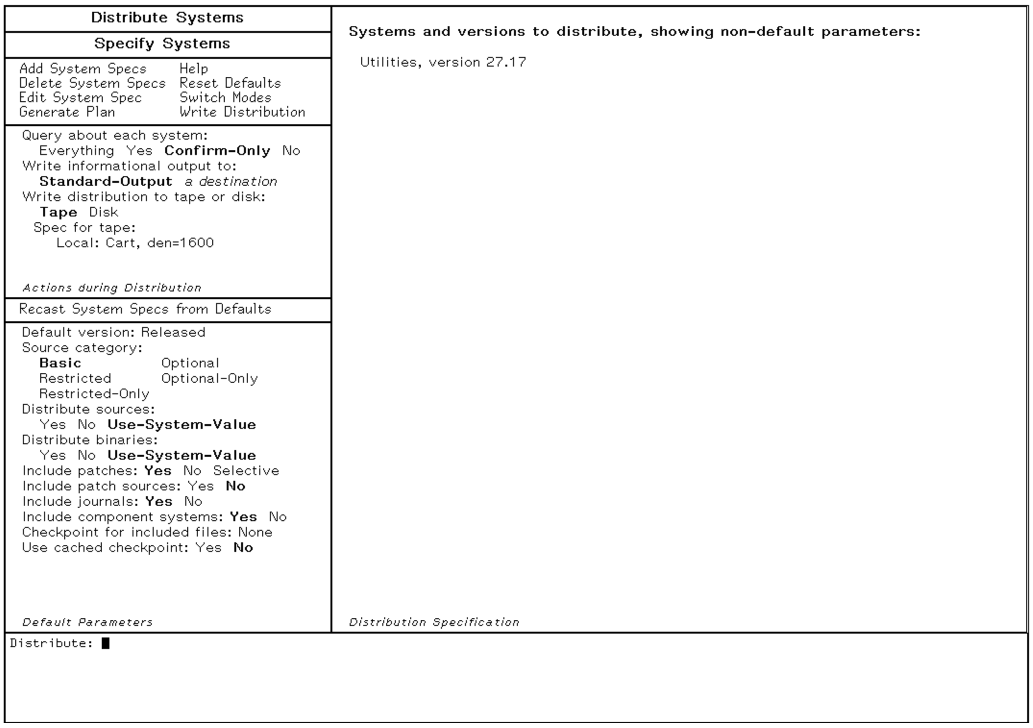
Figure ! shows the Distribute Systems activity.

Overview:

The Distribution Systems window consists of six different sections: one distribution display specification pane, one pane indicating your status in the process of distributing systems, two command menus, and two panes for setting parameters.

While using the Distribute Systems activity, you are in different phases of the process, depending on which activity are performing. The top-left pane always displays the name of the activity you are performing. Initially, the top-left pane displays the words Specify Systems, since this is the first activity you perform.

The first phase of the process occurs when you make your system specifications. These specifications appear in the large Distribution Specification pane if you enter the Distribute Systems activity with the Distribute Systems command, and specify the keyword :Menu, with the value Yes. When you are in this first phase, the top-left pane displays the words Specify Systems.



```
Distribute Systems
Specify Systems

Add System Specs     Help
Delete System Specs  Reset Defaults
Edit System Spec     Switch Modes
Generate Plan        Write Distribution

Query about each system:
   Everything  Yes  Confirm-Only  No
Write informational output to:
   Standard-Output  a destination
Write distribution to tape or disk:
   Tape  Disk
  Spec for tape:
     Local: Cart, den=1600

Actions during Distribution

Recast System Specs from Defaults

Default version: Released
Source category:
   Basic            Optional
   Restricted       Optional-Only
   Restricted-Only
Distribute sources:
   Yes  No  Use-System-Value
Distribute binaries:
   Yes  No  Use-System-Value
Include patches: Yes  No  Selective
Include patch sources: Yes  No
Include journals: Yes  No
Include component systems: Yes  No
Checkpoint for included files: None
Use cached checkpoint: Yes  No

Default Parameters

Systems and versions to distribute, showing non-default parameters:

 Utilities, version 27.17

Distribution Specification

Distribute: █
```

Figure 1. Distribute Systems Menu

After you finish specifying systems, you click on the Generate Plan command in the command menu. The top-left pane then quickly displays the words Generating Plan.... The distribution plan then appears in the Distribution Specification pane, and the words Edit Distribution appear in the top-left pane.

When you are satisfied with the distribution plan, you click on Write Distribution in the command menu, and the words Writing Distribution appear in the top-left pane, signifying that that you are writing the distribution. While writing the distribution, the process prints the progress log in a typeout window, and also in another location if you specify it to do so.

Here is a description of the different panes:

Distribution Specification Pane -- the right-side pane

This pane displays the system specs, as you have defined them up to now.

Specify Systems Pane -- the top-left pane

The top-left pane always displays the name of the phase you are in. The name changes as you move from one activity to another.

The Command Pane -- the second-left pane

The second-left pane offers a command menu for setting parameters of the systems you want to distribute. You can choose one of the commands by clicking on the command name.

Here is an explanation of these commands.

Command	*Description*
Add System Specs	Adds specifications for new systems.
Delete System Specs	Deletes a specification in the list of systems. You can specify individual systems or **all**.
Edit System Spec	Enables you to edit the detailed parameters of an individual specification already in the list of parameters.
Generate Plan	Generates the plan so you can examine it before writing the final distribution. Computes the exact list of files to write, according to your specifications. When it finishes, it switches to the Edit Distribution Plan phase, and displays the list of files.
Help	Gives information about Specify Systems commands.
Reset Defaults	Enables you to reset the actions and default parameters to their initial values.
Switch Modes	Enables you to switch back and forth between the Specify Systems and the Edit Distribution Plan phases. (This command does not change the state of the activity window in any other respect).
Write Distribution	Creates the distribution. Clicking on this writes the distribution to the file or tape device specified in the Actions During Distribution pane.

To add specifications for new systems, you can use Add System Specs in the Specify Systems menu, or you can click left on the title line of the system spec display. The parameter values initially offered in the pop-up editor are from the current set of Default Parameters.

To edit the detailed parameters of an individual specification already in the list, you can use the Edit System Spec in the Specify Systems menu, or you can click left on the displayed line for that system specification.

To delete specifications for a system you can use Delete System Specs, or you can click middle on the displayed line for that system specification.

Actions During Distribution Pane -- the third-left pane

The third-left pane is a parameters pane for changing the settings that control parameters (for example, where to direct the information lines that tell which files are being written). You can change these parameters by clicking on their values.

The first parameter, "query about each system", determines (as the distribution is being written) whether the user will be queried by file, by system, or not at all.

The second parameter, "write Informational output to", determines (as the distribution is being written) where the typeout will go. Values are:

```
Standard-Output, or
```

```
    Destination (Buffer, File, Printer, Stream, or
    Window).
```

The default (standard-output) is to write the information on a typeout window on the screen.

The third parameter, "write distribution to device", determines where the distribution data will be written. Values are machine dependent:

Tape, Disk, or Floppy.

If you choose tape, it prompts you for the tape specification:

```
    Spec for tape: Local: Cart, den=1600
```

For more information about tape specifications, see the section "Tape Specifications".

If you choose disk, the input is written to disk as a special file that is an image of what would be written to tape. You are queried for the pathname of a tape image file. When disk is chosen, the distribution plan is not divided into parts according to any size limit.

If you choose floppy, you are queried for the pathname of a file on a floppy disk. Additionally, when a floppy disk is used, the size limit for a "reel" is set to the capacity of the floppy (that is, 800 Kbytes).

Recast System Specs from Defaults Pane -- the fourth-left pane

The fourth-left pane is a command menu for setting the parameters of the systems to distribute. You may click on the command in this pane after you have changed some parameters in the bottom-left pane, Default Parameters.

This command is useful if you have changed some system specifications in the Distribution Specification activity, and you wish to revert all of them to the defaults displayed in the Default Parameters pane. When you click on Recast System Specs from Defaults, it changes all the specifications for each system to the ones specified in the Default Parameters pane.

Default Parameters Pane -- the bottom-left pane

The bottom-left pane contains the default values for the details of how Distribute Systems writes each system. When you create a new system specification, Distribute Systems uses these defaults as the initial settings for the parameters for that system specification.

Once you create a system specification, it contains a full set of parameter values. These parameter values are displayed in the Default Parameters pane. Note that the Distribution Specification pane only displays the settings that do not correspond to the default parameters.

If you change a default parameter, the display of the existing system specifications changes to reflect the fact that one of its parameter values is no longer the default. You can change a parameter by clicking on its value. Here are the parameters:

Default Version (one of the following):

Released	The version designated as released in the journal file. This is the default.
Latest	The most recent version recorded in the journal file.
Newest	Newest means to ignore the versions in the journal file and just find the newest files.
A positive integer	The version of the system you want to distribute.

Source Category (one of the following):

Basic	Distribute only sources marked Basic.
Restricted	Distribute only sources marked Restricted, Optional, or Basic.
Restricted-Only	Distribute only sources marked Restricted.
Optional	Distribute only sources marked Optional, or Basic.
Optional-Only	Distribute only sources marked Optional.

Distribute Sources (one of the following):

Yes	All sources in the specified source category distributed.
No	No sources distributed.
Use-System-Value	Use the parameter set in the **defsystem** for each system.

Distribute Binaries (one of the following):

Yes	All binary files included.
No	No binary files included.
Use-System-Value	Uses the parameter set in the **defsystem** for each system.

Include patches (one of the following):

Yes	All patches included.
No	No patches included.
Selective	Prompts you for which patches to include.

Include Patch Sources (one of the following):

Yes	All patch sources included.
No	No patch sources included.

Include journals (one of the following):

Yes	All journals included.
No	No journals included.

Include component systems (one of the following):

Yes	All component systems included.
No	No component systems included.

Checkpoint for included files: limit the distributed files to those after (one of the following):

A positive integer	A patch number.
Release	A release name.
None	Do not limit.

Use cached checkpoint (one of the following):

Yes	Use the last checkpoint gathered for this system. Using the cached checkpoint information, if there is any, saves time. But it is safe to use only if you are sure no more patches have been made since the cached information was computed.
No	Do not use the last checkpoint gathered for this system.

After Setting All of the Parameters

After you set all of the parameters, use the Generate Plan command in the command menu to compute the exact list of files to write, according to your specifications. When it finishes, it switches to the Edit Distribution Plan phase, and displays the list of files. You can edit the distribution plan if you like. When you are done editing the distribution plan, or are satisfied with it, click on the Write Distribution command to write the distribution.

How to Scroll the Screens

In the screens with scroll bars, you can scroll the screen display with the `SCROLL` key, or by pressing `m-SCROLL`. In addition, you can use `m-<` to move to the beginning of the display, and `m->` to move to the end of the display.

Directories Associated with a System

Each system is associated with a set of directories and files. This section explains the directory structure associated with a system called *zoo*. Under a single logical directory, `zoo;` reside these files:

1. The System declaration file which contains the **defsystem** form for this system.

2. Multiple versions of source and product files that comprise the system.

3. A *journal* directory, by default called `patch;`.

The journal directory contains a subhierarchy of files that contain the history of the compilations done and the patches made to the system.

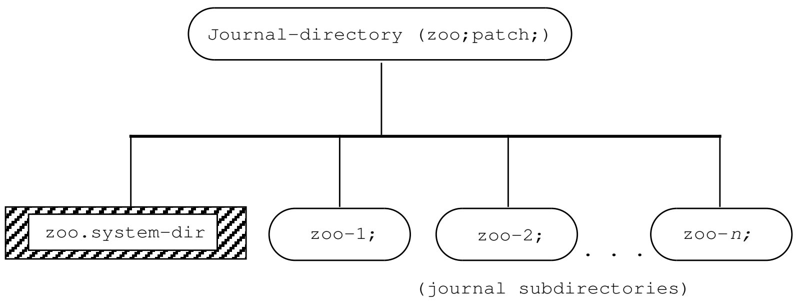


Figure 2. Journal Directory for system Zoo

The *system-dir* file is a registry of the location of the component-directory file for a given version of a system for a given machine. Here is the `zoo.system-dir` file looks like:

```
;;;  -*- Mode: LISP; Base: 10 -*-
;;; Written 1/30/89 13:39:22 by Ellen,
;;; while running on James Baldwin from FEP0:>Writer-from-SCRC-7-3I-A-etc.ilod.1
;;(("ZOO" :RELEASED 404 :LATEST 404)
;;; System versions: ...
 (404
  (:|3600|
   (:COMPONENT-DIRECTORY
    ("SYS:ZOO;PATCH;ZOO-404.COMPONENT-DIR" :NEWEST NIL))))
 (403
  (:|3600|
   (:COMPONENT-DIRECTORY
    ("SYS:ZOO;PATCH;ZOO-403.COMPONENT-DIR" :NEWEST NIL))))
 (402
  (:|3600|
   (:COMPONENT-DIRECTORY
    ("SYS:ZOO;PATCH;ZOO-402.COMPONENT-DIR" :NEWEST NIL))))
 (401
  (:|3600|
   (:COMPONENT-DIRECTORY
    ("SYS:ZOO;PATCH;ZOO-401.COMPONENT-DIR" :NEWEST NIL))))
 (8
  (:|3600|
   (:COMPONENT-DIRECTORY
    ("SYS:ZOO;PATCH;ZOO-8.COMPONENT-DIR" :NEWEST NIL))))
 (7
  (:|3600|
   (:COMPONENT-DIRECTORY
    ("SYS:ZOO;PATCH;ZOO-7.COMPONENT-DIR" :NEWEST NIL))))
```

Each journal subdirectory is associated with a particular version of a system. Here is the structure of the journal subdirectory `zoo-2;`.

Component Directory File

The *component directory file* is not an actual directory in the file system, but is rather a registry of the source and product version numbers for a major version of a system. Whenever you perform an operation on a system, that operation uses this file to determine the versions of the system files to operate on.

Here is an example of the contents of a component-directory file.

```
;;;  -*- Mode: LISP; Base: 10 -*-
;;; Written 11/10/89 12:05:58 by Zippy,
;;; while running on Brown Creeper from FEP0:>Base-System-424-0.load.1
```

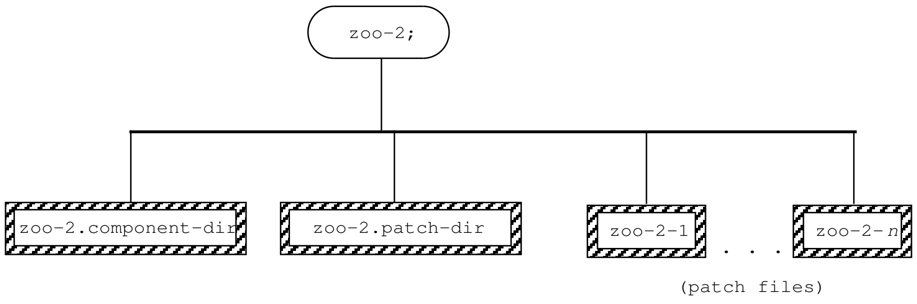


Figure 3. Journal Subdirectory for Zoo Version 2

```
(("IP-TCP" 420)
 ;; Files for version 420:
 (:|3600|
  (:DEFSYSTEM
   ("SYS:IP-TCP;SYSTEM" 107 NIL))
  (:INPUTS-AND-OUTPUTS
   ("SYS:IP-TCP;TCP-STRUCTURE.TEXT" 10 NIL)
   .
   .
   .
   ("SYS:IP-TCP;EGP" 83 58))))
```

When you compile a system a new component directory file is created. The major benefit of this detailed record keeping is that your site can support multiple versions of the same system. General users and system developers can load specific versions of systems and specific versions of system files, even when newer and possibly incompatible versions have been made. Some examples:

- System developers can work on the *latest* versions of systems, editing and recompiling some files, without forcing the average user to contend with new and experimental changes to the system.
- General users, on the other hand, can load the stable, *released* versions.
- Symbolics can more easily distribute versions of the system other than the newest version.
- You can use old versions of systems after recompiled versions have been made for the latest system software.

In addition, you can load a system in several different ways:

- by version number

- by version name
- by designation as released, latest, or newest

To load a specific system, use the **:version** option for **load-system**

The *released* version is the fully debugged version intended for general use. To designate a system as the released version use either **sct:release-system** or **compile-system** with the **:update-directory** option to make the change in the component directory file.

The *latest* version is the most recently compiled version of the system. The component directory file is automatically updated whenever you compile or recompile the system; **compile-system** assigns the **:latest** keyword to this system.

The *newest* version of a system consists of the most recently compiled version of each *file* of a system. The newest version differs from the latest version when individual files have been compiled by hand. The newest version of a system has no version number. Note that you cannot define patches for the newest system.

Contents of the Patch Directory Files

Two *patch-directory* files are created for each patch (one when the patch is begun and another when the patch is finished). The patch-directory file is not a directory; it is a registry of minimum information about a patch including the number of the patch, a comment, the author, and a timestamp. A new patch directory file is created automatically when you recompile a system.

Here is an example of the contents of a patch-directory file.

```
;;; -*- Mode: Lisp; Package: ZL-User; Base: 10.; Patch-File: T -*-
;;; Patch directory for IP-TCP version 420
;;; Written 1/17/90 12:25:42 by Hornig,
;;; while running on Brown Creeper from FEP0:>Base-System-424-0.load.1
;;; ...
(:EXPERIMENTAL
 ((0 "IP-TCP version 420 loaded." "SWM" 2729958587)
  (1
 "Make TFTP work again.
Function TCP::IP-STORE-16:  Needs ONCE-ONLY.
Function TCP::IP-STORE-32:  ditto
Function (DEFUN-IN-FLAVOR TCP::TFTP-FLUSH-BUFFER
TCP::TFTP-OUTPUT-STREAM):
Recompile caller."
 "Hornig" 2730718516)
  (2
 "Function (DEFUN-IN-FLAVOR TCP::IP-RETRANSMIT-PACKET
TCP::IP-PROTOCOL):
Don't ever forward or redirect broadcast packets."
 "Hornig" 2730990265)
  ))
```

The *patch files* themselves are found in both source and product form, with one source and one product file associated with each patch.

Patch Facility

How Patching Works

Software development is usually a process of incremental changes to many large programs. Many developers can be involved, and the changes can be distributed to any number of users, including the same developers. (Note: the term *large program* refers to one defined by **defsystem**. Only these programs can make use of the patch facility.)

Briefly, developers fix or improve existing functional and other definitions (or write new ones), and then, after thorough testing, decide to issue their changes to the users at their site. They effect release in two ways: (1) they write new versions of the source files containing the edited or new definitions, and (2) they create *patch files*, which contain only the new or changed definitions. Every time a patch is created (written to disk), the patch facility automatically records the event in a sort of "patch registry", noting the number of the patch, the system being patched, and a brief summary of the patch, as described by the developer. Zmacs, the Symbolics editor, provides special tools that make this process relatively easy for the developer.

The patch facility creates a patch file. Saving your buffer after you make a change creates a new version of your source file. When the system is recompiled, your source file, and not the patch file, will be used to construct the new system. The important point is that the patch files — and not the newly written source files — allow the changes to be put into widespread use immediately. The patch facility allows users to obtain all the incremental changes to a system simply by loading its associated patch files.

Basically what occurs during the loading of patches is this: the current state of the patch registry is compared to the registry as last loaded by the user. If patches have been written since that time, just the new patches are loaded, and their summary descriptions are displayed. At that point, the state of the given system in the user's machine is presumably the same as in the developer's machine when the patch was finished.

Genera provides a number of convenient tools and several interfaces for loading patches. For example, users can load patches by calling one of several Lisp functions or alternatively using Command Processor commands. Users also have the choice of loading patches to virtual memory (which means they disappear when the machine is booted) or of saving the patches to disk. (Of course, new patches can be made later, and then these will have to be loaded to get the very latest version of a system.) In the case where users load a particular system whenever they want to use it, the system-loading facility automatically loads all the patches for that system.

Inevitably, a developer or system maintainer must stop accumulating patches and recompile all the source files in a large program, for example, when a system is changed in a far-reaching way that cannot be accomplished with a patch. Only at this point do the source files become important to system maintenance and distribution. After a complete recompilation, the old patch files are useless and should not be loaded.

To keep track of all the changing number of files in a large program, the patch facility labels 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.

The following typical scenario should clarify this scheme.

1. A new system is created; its initial version number is 1.0.
2. 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.
3. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2.
4. Then the entire system is recompiled, creating version 2.0 from scratch.
5. 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 should not 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 cited.
- Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

In addition to enabling users to have the most up-to-date programs available, the patch facility performs another important function. Via the patch registry, it allows a site to support multiple versions of the same system. Thus, general users can load a stable, debugged version, while system developers can run the *latest* version of the same system, editing and recompiling files, without forcing the general user to deal with experimental changes. The detailed record keeping that this capability requires is maintained in a hierarchy of files that is created automatically and updated whenever a system is compiled.

The patch registry also keeps track of all the individual patch files that exist, remembering which version each one creates. A separate numbered sequence of patch files exists for each major version of each system, for example, lmfs-37-15.lisp, lmfs-37-16.lisp, and so forth. All patches for each major version are stored in the *journal subdirectory* associated with that version of the system. See the section "Directories Associated with a System".

In addition to the patch files themselves, the *patch-directory file* keeps track of what minor versions exist for a major version. For example, lmfs-37.patch-dir contains a listing of the patches made for major version 37, their author, a timestamp, and a comment on why each patch was made.

In order to use the patch facility, you must define your system with **defsystem** and declare it as patchable with the **:patchable** option. (**:patchable** is the default.) When you load your system, it is added to the list of all systems present in the world. Whenever you compile your patchable system, its major version in the file system is incremented; thus a major version is associated with a set of compiled code files.

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 Genera 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 patches or whole new systems and then save the entire Genera environment away in a FEP file. See the function **zl:load-and-save-patches**.

Types of Patch Files

The patch facility maintains several different types of files in the journal subdirectory associated with a major version of your system:

- The patch directory files (two versions for each patch)
- Individual patch files (both source and product versions)

The patch directory file constitutes a sort of "patch registry", recording the number of the patch, the name and version of the system being patched, and a brief description of the patch. One version of the patch directory file is created when starting a patch, and another is created when finishing a patch. (Of course, old versions can be deleted and expunged.) See the section "Component Directory File".

Patch Directory File

The *patch directory file* in the journal subdirectory keeps a listing of the patches (minor versions) that exist for a major version. Each major version of the system has its own patch directory file, which lists the minor version number, any comments about the patch, and the patch author. A new patch directory file is created automatically when you recompile a system.

See the section "Directories Associated with a System".

See the section "Component Directory File".

See the section "Contents of the Patch Directory Files".

Individual Patch Files

Each minor version of the system has a patch source file and a corresponding compiled code file. The individual patch files for a major system version reside in the subdirectory for that major version. (The patch directory file also resides in this subdirectory.) Each patch file is uniquely identified by the major and minor version numbers of the system. For example, lmfs-37-3.lisp would be the name of the patch source file for major version #37 and minor version #3 of **lmfs**.

Organization of Patch Files

The component directory file, the patch directory file, and the individual patch files are created and maintained automatically, but you will need to know where the patch facility stores these patch files and how to find them on your host.

The patch facility knows which directories to associate with your system by looking at how you specified the **:patchable** option and the **:default-pathname** option in your system declaration. For example, the following **defsystem** declaration will cause the patches to be stored in the logical directory "george: patch;" rather than in the directory that holds the other files of the system, the pathname default.

```
:default-pathname "george: george;"
:patchable t
:journal-directory "george: patch;"
```

When you do not supply the journal-directory then the patches are stored in the directory specified by **:default-pathname**; plus **patch;**. In the following example this is the logical directory "george: george; patch;".

```
:default-pathname "george: george;"
:patchable t
```

The source and compiled code patch files for a major system version are kept in the component directory, along with the component directory file. The patch directory file for a major version resides in this same directory.

Names of Patch Files

The patch facility chooses names for your patch files based on your system definition and on the host.

The host determines the file type and the number of characters in the file name. For example, VMS, UNIX 4.2, and ITS use a computer-generated contraction of the file name. A system directory file name like charlie.system-dir on LMFS would be CHARLI (SDIR) on ITS. Similarly, a patch directory file name like charlie-1.patch-dir on LMFS would be CHA001 (PDIR) on ITS.

The following tables show the physical file types of the system directory file, the patch directory file, and the component directory file for various hosts.

Host	*File type of the system directory file*
TOPS-20	SYSTEM-DIR
UNIX 4.1	sd
UNIX 4.2	system-dir (also sd for compatibility)
VMS	SPD
ITS	(SDIR)
LMFS	system-dir
Multics	system-dir

Host	*File type of the patch directory file*
TOPS-20	PATCH-DIR
UNIX 4.1	pd
UNIX 4.2	patch-dir (also pd for compatibility)
VMS	VPD
ITS	(PDIR)
LMFS	patch-dir
Multics	patch-dir

Host	*File type of the component directory file*
TOPS-20	COMPONENT-DIR
UNIX 4.1	cd
UNIX 4.2	component-dir (also cd for compatibility)
VMS	CPD
ITS	(CDIR)
LMFS	component-dir
Multics	component-dir

The format of patch file names varies with the type of file.

- The format of the system directory file is some name chosen by the patch facility followed by the appropriate file type and file version number. For example, the system directory file on LMFS for the **george** system might be:

  ```
  q:>sys>george>patch>george.system-dir.1
  ```

- The format of the patch directory file name is some name followed by the major version number and the appropriate file type and file version number. For example, the patch directory file on LMFS for major version #38 of **george** might be:

  ```
  q:>sys>george>patch>george-38>george-38.patch-dir.44
  ```

  Note that the file resides in a subdirectory of the same name.

- The format of the individual patch file is some name chosen by the patch facility followed by the major version number, the minor version number, and the appropriate file type and file version number. For example, source patch file #1 for major version #38 of **george** might be:

```
q:>sys>george>patch>george-38>george-38-1.lisp
```

Because the translation rules for generating patch file pathnames are fairly complicated, they are not given here. Instead use the **sct:patch-system-pathname** function to determine the names of your patch files.

sct:patch-system-pathname *system type* &rest *args* *Function*

Given a system name and the type (**:component-directory**, **:system-directory**, **:patch-directory**, **:patch-file**) and additional args required by that type, returns the pathname for the file in question. Additional args are, in order, *system-major-version*, *system-minor-verison,* and *file-canonical-type*. **:system-directory** requires none of these, **:component-directory** and **:patch-directory** require one, and **:patch-file** all three.

Returns the logical pathname of a patch file. *system* is the name of the system. *type* is **:patch-file**, **:system-directory**, **:component-directory**, **:patch-directory**, or **:patch-file**. Specify also any additional *args* required by the type.

:component-directory
Returns the logical pathname of the component directory file for the system specified by a major version number, for example:

```
(sct:patch-system-pathname "LMFS"
                           :component-directory 37)
```

The form returns `#P"SYS:LMFS;PATCH;LMFS-37.COMPONENT-DIR.NEWEST"`.

:system-directory
Returns the logical pathname of the system directory file for the specified system, for example:

```
(sct:patch-system-pathname "LMFS"
                           :system-directory)
```

The form returns `#P"SYS:LMFS;PATCH;LMFS.SYSTEM-DIR.NEWEST"`.

:patch-directory
Supplied with a *major-version-number* argument, it returns the logical pathname of that patch directory file for the given system, for example:

```
(sct:patch-system-pathname "LMFS"
                           :patch-directory 51.)
```

The form returns `#P"SYS:LMFS;PATCH;LMFS-51.PATCH-DIR.NEWEST"`.

:patch-file Supplied with the *major-version-number*, *minor-version-number*, and *canonical-type* arguments, it returns the logical pathname of the patch file.

```
(sct:patch-system-pathname "LMFS"
                           :patch-file 51. 2.
                           :lisp)
```

The form returns `#P"SYS:LMFS;PATCH;LMFS-51-2.LISP.NEWEST"`.

To find the physical pathname translation of any of these, send the returned value the **:translated-pathname** message. For example, send the **:translated-pathname** message to the returned value of `(sct:patch-system-pathname "LMFS" :system-directory)`. The form would return `#P"Q:>SYS>LMFS>PATCH>LMFS.SYSTEM-DIR>"`.

Making Patches

During a typical maintenance session you might make several changes to existing definitions or write new ones. Rather than recompiling the entire system every time you change a source file, you can copy only the new or revised code into a *patch file* and write the file ("finish" the patch). Whenever you finish a patch, the patch facility automatically compiles the file and records the event in a "patch registry" for the system, noting the number of the patch, the system being patch, and a brief user-supplied description. As soon as a user loads the patch file (after the system is loaded), the state of the given system in the user's machine is presumably the same as in the developer's machine when the patch was finished.

The patch facility allows you to have several patches in progress at once. Thus you can patch several different systems or several different minor versions of the same system during one work session. The patch facility manages this potentially dangerous situation in the following way. Every time you start a patch, a number and a place in the patch registry is reserved for the patch in production. The patch is marked *in-progress*. When the patch is finished, the entry is completed and the in-progress mark removed. If you decide to abort the patch, the registry entry is automatically deleted.

The ability to have more than one patch in-progress to more than one system makes it imperative that you keep track of the state of your various patches. If a patch is left unfinished (unwritten), the **load-patches** function will load neither the in-progress patch nor any subsequent finished patches.

The patch facility considers patches to be active or inactive and in one of the following states: initial, in-progress, aborted, or finished. Show Patches (`m-X`) displays the state of all patches started in this work session. If more than one patch is in progress, one of them is known as the *current patch*. The commands that add patches, like Add Patch (`m-X`), add only to the patch considered by the patch facility to be the current patch. The command Select Patch (`m-X`) displays a menu of active patches and allows you to make another patch the current one.

In general you should adhere to the following steps in making a patch. It is assumed that your system is patchable; that is, the **:patchable** option appears in the system declaration.

1. You must load (via **load-system**) the major version of the system that you want to patch.
2. Read in the source files you want to edit into a Zmacs buffer. Make all changes and test them thoroughly. Write the source file.
3. Use the appropriate Zmacs commands to make your patch. Begin the patch, using Start Patch (`m-X`).

4. Add the changed code to the patch buffer by using Add Patch (m-X), Add Patch Changed Definitions of Buffer (m-X), or Add Patch Changed Definitions (m-X).

5. Finish the patch, using Finish Patch (m-X), or abort the patch, using Abort Patch (m-X).

Commands provided for initiating a patch are Start Patch (m-X), Start Private Patch (m-X), and Add Patch (m-X).

Start Patch (m-X)

Starts a new patch, prompting you for the name of the system to be patched; it must be a system currently loaded. It assigns a new minor version number for that particular system by writing a new version of the patch directory file with an entry for that minor version number. The patch is marked as in-progress. It starts constructing the patch file in an editor buffer, but does not select the buffer.

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. Thus, if two people are patching the same system at the same time, they cannot be assigned the same minor version number.

The command does not actually move any definitions into the patch file. You must explicitly do so with Add Patch Changed Definitions of Buffer (m-X), Add Patch Changed Definitions (m-X), or Add Patch (m-X).

The patch facility permits you to start another patch before finishing the current one. However, if your new patch is to the same system, the patch facility warns you that you already have a patch in progress and allows you to take one of four actions:

- Abort the in-progress patch and start a new patch.
- Finish the in-progress patch and start a new patch.
- Proceed with the second patch (initial patch) for this system and leave the in-progress patch intact.
- Use the existing buffer and do not start a new patch.

Start Private Patch (m-X)

Although similar to Start Patch (m-X), Start Private Patch (m-X) does not have any relationship to systems, major and minor version numbers, and official patch directories. Rather it allows you to make a private patch file that you can load, test, and share with other users before you install a numbered patch that is automatically available to all users.

Instead of prompting for a system name, the command prompts for a file name. It also prompts for a patch note to be saved with the patch. The default for this private patch note is the same as the name component of the private patch pathname, except that spaces are converted to hyphens. This patch note is also offered as the subject line of a mail message if you select **yes** for *Send mail about this patch* in the Finish Patch menu.

Start Private Patch does not actually move any definitions into the patch file. Use Add Patch Changed Definitions of Buffer (m-X), Add Patch Changed Definitions (m-X), or Add Patch (m-X) to insert the code. Finishing the patch (using Finish Patch (m-X)) writes it out to the specified file.

Note: Use the Load File command or Load File (m-X) to load a private patch; the Load Patches command and the **load-patches** function do not load private patches.

Add Patch (m-X)

Starts a new patch if none is underway, prompts you for a system name, and inserts the region or current definition into the patch buffer. If a patch was in progress, Add Patch (m-X) just adds the region or current definition to the current patch file.

Warns you if your editor buffer conflicts with the system being patched. If you mistakenly use Add Patch on code that does not work, select the buffer containing the patch file and delete it. Then later you can use Add Patch (m-X) on the corrected version. For each patch you add, it queries for a patch comment, which it then inserts in the patch file. Just pressing END means "no comment".

Add Patch (m-X), Add Patch Changed Definitions (m-X), or Add Patch Changed Definitions of Buffer (m-X) insert code into the patch file. If the patch is being made to the system the current buffer's file came from, the commands proceed.

If there is a current patch, and it is not appropriate for the system that the buffer's file came from, you see a menu showing all of the current patches, plus an option to create a new patch appropriate for the buffer, plus an option to abort.

Add Patch Changed Definitions of Buffer (m-X)

Add Patch Changed Definitions of Buffer (m-X) selects each definition that was changed in the buffer and asks you whether or not you want the definition patched.

For each definition, you can respond as follows:

Response	*Action*
Y	Patches the definition.
N	Skips the definition.
P	Patches the definition and any additional modified definitions in the same buffer without asking any more questions.

A definition needs to be patched if it has been changed since it was last patched or if it has not been patched since the file was read into the buffer.

For each patch you add, it queries for a patch comment, which it then inserts in the patch file. Just pressing END means "no comment".

Add Patch Changed Definitions (m-X)

Add Patch Changed Definitions (m-X) selects a buffer in which definitions were changed and asks whether or not you want to patch the changed definitions. Answering N skips the buffer and proceeds to the next buffer of the same mode, if any. Answering Y selects each definition that has changed in that buffer and asks you whether or not you want the definition patched. For each definition, you can respond as follows:

Response	*Action*
Y	Patches the definition.
N	Skips the definition.
P	Patches the definition and any additional modified definitions in the same buffer without asking any more questions; when done, it proceeds to the next buffer.

If there are more buffers containing definitions to be patched, it asks questions again when it gets to the next buffer.

A definition needs to be patched if it has been changed since it was last patched and if it has not been patched since the file was read into the buffer.

For each patch you add, it queries for a patch comment, which it then inserts in the patch file. Just pressing END means "no comment".

Add Patch Changed Definitions selects buffers based on the mode of the buffer from which the command is issued. Thus if you are in a Lisp mode buffer, any Lisp mode buffers with definitions to be patched are offered, and if you are in another mode buffer, only buffers of that mode are offered.

When making multiple patches during one work session use the Select Patch and Show Patches commands to keep track of patches.

Select Patch (m-X)

When you are making more than one patch during a work session, Select Patch (m-X) allows you to choose a different patch as the current patch from a menu of active patches. The patching commands (like Add Patch and Add Patch Changed Definitions of Buffer) insert definitions into the patch file that you have selected as the current patch. To insert patch definitions into another buffer, use Select Patch to choose that buffer as the current patch.

Show Patches (m-X)

Show Patches (m-X) displays the state of all patches started in this session. Patches are either active or inactive and can be in one of the following states: initial, in-progress, aborted, or finished. *Inactive patches* are in an aborted or finished state. *Active patches* are in an initial or in-progress state. *Initial* means that the patch buffer has been initialized but as yet no definitions have been added to the buffer. *In-progress* means that the patch buffer has been initialized and definitions have been added to the buffer.

Show Patches groups the active and inactive patches and identifies the current patch.

After making and testing all of your patches, use the Finish Patch command to install the patch in the system.

Finish Patch (m-X)

Finish Patch (m-X) installs the patch file so that other users can load it. This command saves and compiles the patch file (patches are always compiled). If the compilation produces compiler warnings, the command asks whether or not you want to finish the patch anyway. If you do, or if no warnings are produced, a new version of the patch directory file is written. The in-progress mark is removed from the entry in the patch registry.

The command allows you to edit the patch comments, which are written to the patch directory file. (**load-patches** and **zl:print-system-modifications** print these comments.) It then asks you whether you want to send mail about the patch. If you say "yes", it opens a mail buffer and inserts initial contents, including the name of the patch file and your patch comment.

Note: By default the Finish Patch command queries you about sending mail. You can alter this behavior by changing the value of the variable **zwei:*send-mail-about-patch***. Its valid values are **:ask**, the default value, which queries the user; **t**, which opens a Zmacs mail buffer without querying; and **nil**, which takes no action regarding the sending of patch mail.

The Finish Patch menu lists the modified source files for the patch and offers to save them as part of the Finish Patch process. If you do not save your files as part of the Finish Patch process, Finish Patch displays a reminder to save your files when it finishes writing the patch directory. You can set the variable **zwei:*finish-patch-save-sources-default*** (default **nil**) to **t** in your init file to have Finish Patch save your files automatically.

Sometimes you start making a patch file and for a variety of reasons do not finish it — for example, you decide to abort the patch, you need to end your work session at this machine, or your machine crashes. In each of these situations it is of the utmost importance that you leave the patch directory file in a clean state; that is, either go back and finish the patch (as soon as possible!) or deallocate the patch number reserved to you. Failure to do so has unfortunate consequences: users at your site will not be able to load patches.

If your machine has crashed, use Resume Patch (m-X) to reclaim access to the patch number previously assigned to you. You can continue with the patch (assuming you saved the source files just prior to the crash) or use Abort Patch (m-X) to deallocate the patch number. Begin the patch again if you wish. If you simply decide to abandon the patch file, then just use Abort Patch. If you must boot your machine before finishing the patch, then save the patch buffer and as soon as possible use Resume Patch to read in the relevant patch file; finish the patch or abort it, as you wish.

Abort Patch (m-X)

Abort Patch (m-X) deallocates the minor version number that was assigned by the Start Patch or Add Patch commands. It tells Zmacs that you are no longer interested in making the current patch and offers to kill the patch buffer. The next time you do Add Patch (m-X), Zmacs starts a new patch instead of appending to the one in progress.

Resume Patch (m-X)

Resume Patch (m-X) allows you to return to a patch that you were not able to finish in the same boot session in which you started it; for example, your machine might have crashed or you had to boot your machine suddenly. It reads in the relevant patch file if it was previously saved; otherwise it just reclaims your access to the minor version number allocated to you when you started the patch. Abort or finish the patch.

Under certain circumstances you might find it necessary to recompile and reload a patch file.

Recompile Patch (m-X)

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

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

Reload Patch (m-X)

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

You might want to have your herald announce private patches that you make. **note-private-patch** adds a private patch to the database in your world and includes the name of the patch in the herald.

note-private-patch *string* *Function*

Adds a private patch to the database in your world. **note-private-patch** takes a *string* argument. For example, the following adds the private patch called patch.lisp:

```
(note-private-patch "s:>maria>patch.lisp")
```

Subsequent displays of your herald show the inclusion of that patch in your world.

You create private patches using the Start Private Patch (`m-X`) command and then the standard patch commands for adding to and finishing the patch. Use the Load File command or Load File (`m-X`) to load a private patch; the Load Patches command and the **load-patches** function do not load private patches.

sct:require-patch-level-for-patch &rest *system-major-minor-specs* *Function*

Enforces a patch's dependency on some particular patch level in another system or systems. It is used at the head of any patch file that requires a certain patch level in some other system to load or operate correctly. For example:

```
(sct:require-patch-level-for-patch '(system 357. 510.) '(tape 69. 10.))
```

If the patch level requirements are not all met, loading the patch (and subsequent patches for that system) is skipped. After patches are loaded for all systems, Load Patches is called again to see if the patch levels of the other systems are now high enough to permit loading of additional patches. You can specify the patch level requirements from the Finish Patch menu.

zwei:*send-mail-about-patch* *Variable*

Controls whether the Finish Patch menu question "Send mail about this patch?" comes up set to **yes**. The possible values are **t**, **nil**, or **:ask**. The default is **:ask**. If the value is **t**, the question always appears with **yes** set.

zwei:*finish-patch-save-sources-default* *Variable*

Controls whether the Finish Patch menu question "Save sources for this patch?" comes up set to **yes** or **no**. The possible values are **t** or **nil**. The default is **nil**. If the value is **t**, the question always appears with **yes** set.

Dangerous Patches

Occasionally you need to make a patch to a system that, in some circumstances, might damage the system. For example, it might make changes to very low level internal functions or initialize parts of the system. Loading such a patch into a running system could unpleasantly affect the operation of the system. Such a patch is referred to as a *dangerous patch*. You can declare a patch dangerous by placing the form **sct:dangerous-patch** at the beginning of the patch file.

sct:dangerous-patch *format-string* &rest *format-args* *Function*

Specifies a patch as being problematic to load (a *dangerous patch*) in some circumstances.

To declare a patch a dangerous patch, place a form containing **sct:dangerous-patch** at the beginning of the patch file, before the contents of the patch, to test for the conditions under which the patch should not be loaded.

For example, if you have a program that creates a list called ***my-results*** to store its results, you would not want to load a patch that reinitializes that list if the program were running. You should put a form like this at the beginning of the patch file:

```
(when (listp *my-results*)
  (sct:dangerous-patch "This patch cannot be loaded because it
reinitializes *my-results*"))
```

When you attempt to load the patch, **load-patches** checks the value of **sct:*dangerous-patch-action*** to determine the action to take.

Using **sct:dangerous-patch** at top level (not inside a conditional form) produces an error when you attempt to finish the patch.

sct:with-dangerous-patch-action *(action)* &body *body* *Function*

Allows you to bind **sct:*dangerous-patch-action*** during a **load-patches** operation. This is useful if you are loading patches under program control.

```
(sct:with-dangerous-patch-action :load (load-patches))
```

The possible values for **sct:*dangerous-patch-action*** are:

:skip	The default. Skips loading patches for the system.
:query	Queries you, allowing you to skip loading patches for the system or load the dangerous patch.
:load	Loads the patch inspite of its dangerous status.

The Load Patches CP command takes a keyword argument, :Dangerous Patch Action, that is the same as **sct:with-dangerous-patch-action**.

sct:*dangerous-patch-action* *Variable*

Controls the action taken when a dangerous patch is encountered in loading patches for a system. See the function **sct:dangerous-patch**.

:skip	The default. Skips loading patches for the system.
:query	Queries you, allowing you to skip loading patches for the system or load the dangerous patch.
:load	Loads the patch inspite of its dangerous status.

Loading Patches

When you command the loading of patches for a software system the current state of the patch registry is compared to the registry as last loaded by the user. If patches have been written since that time, just the new patches are loaded, and their summary descriptions are displayed. As each patch is loaded, the state of the given system in your machine is at the same level as in the developer's machine when that particular patch was finished.

The patch registry manages the appropriate loading of patches for a particular system. New patches for a system (since the last loading, if any) are installed until no more remain or until an in-progress patch is encountered. In this last case, loading is halted before the patch in-progress is installed, because the consistency of patches that might follow cannot be guaranteed. The system displays a message indicating the presence of unfinished patches.

Genera provides a number of convenient tools and several interfaces for loading patches. For example, you can load patches by calling one of several Lisp functions — **load-patches** or **zl:load-and-save-patches** — or alternatively, by issuing the Load Patches command in the Command Processor. The effect of these tools differs: **load-patches** and its Command Processor equivalent loads patches to virtual memory, which means they disappear when the machine is booted; **zl:load-and-save-patches** writes the patches to disk. (Of course, new patches can be made later, and then these will have to be loaded to get the very latest version of a system.)

When you call **load-system**, the System Construction Tool automatically loads all the patches for that system, using the same options specified in the call to **load-system**.

Load Patches **Command**

Load Patches *system keywords*

Loads patches into the current world for all systems, locally maintained systems, or the indicated systems.

system	{All Local *system-name1, system-name2* ... } The system(s) for which to load patches. The default is All.
keywords	:Dangerous Patch Action, :Excluding, :Include Components, :More Processing, :Output Destination, :Query, :Save, :Show

:Dangerous Patch Action
: {Skip, Query, Load} Whether to skip loading *dangerous* patches, that is, patches that might make data structures in your world inconsistent, causing unexpected behavior. The default is Skip.

:Excluding
: {*System(s)*} Excludes loading patches for these systems.

:Include Components
: {Yes, No} Whether to load patches for any component systems. The default is No. The mentioned default is Yes.

:More Processing
: {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
: {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Query
: {Yes, No, Ask} Yes asks for confirmation before beginning the load patches process and again before loading each patch. Ask asks whether or not it should query before each patch, and then for confirmation before beginning the load patches process. The default is No. The mentioned default is Yes.

:Save
: {*pathname*, Prompt, No-Save} The file in which to save the world with all patches loaded. Omitting this keyword means do not save the world. The mentioned default is Prompt, which means save the world and then prompt for a pathname.

:Show
: {Yes, No, Ask} Whether to print the patch comments as each patch is loaded. The default is Yes.

See the function **load-patches**.

load-patches &optional *systems* &key (*query* **t**) *silent no-warn reload include-components* *Function*

Brings 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. **load-patches** also loads the translations file (`sys:site;`*logical-host*`.translations` file) if it has changed. **load-patches** returns **t** if any patches were loaded, and **nil** otherwise.

Note: When you do a **load-system** of a patchable system, **load-system** calls **load-patches** after loading the system. If **load-system** is silent, **load-patches** is silent; if **load-system** asks for confirmation, **load-patches** asks for confirmation.

With no arguments, **load-patches** assumes you want to update all the systems present in this world and asks you whether you want to load each patch.

:query	Takes **t**, **nil**, **:confirm**, or **:no-confirm**. If **t**, ask for approval of each and every operation. If **nil** or **:no-confirm**, don't ask about anything. If **:confirm**, list all the operations and then ask for confirmation. Default-value: **:confirm**.
:silent	Takes **t** or **nil**. If **t**, perform all operations without printing anything. If **:query** is non-**nil**, **:silent t** is overridden. Default value: **nil**.
:no-warn	Takes **t** or **nil**. If **t**, don't bother to print a redefinition warning. Default value: **nil**.

zl:load-and-save-patches &rest *keyword-args* *Function*

Disables network services and `MORE` processing and then loads any patches that need to be loaded and any new versions of the site files, calling **load-patches** with arguments of **:query nil**.

If no one is logged in, it logs in anonymously. If any patches have been loaded, **zl:load-and-save-patches** prompts for the name of a FEP file in which to save the world load and then calls **zl:disk-save** to actually save the resulting world load. If no patches have been loaded, it restores network services to their state before **zl:load-and-save-patches** was called, and logs out if it has logged in anonymously.

Call **zl:load-and-save-patches** *before* you log in in order to avoid putting the contents of your init file into the saved world load.

Note that loading files asynchronously — particularly patch files — is neither guaranteed to work nor an efficient use of resources. The main process and the background process would compete for resources, and you would lose a lot of time to paging and the scheduler. Furthermore, you cannot expect the correct results from loading patch files in a background process for the following reasons:

- **load-patches** can reset and rebuild the site information.
- When a foreground bug occurs while patches are loading, you cannot determine what system the bug occurred in.
- When you are using a subsystem in the foreground while it is being patched in the background, unexpected problems could arise.

- The file could be doing something that maps over all pathnames, expecting that pathnames would not change while it was running.
- **defflavor** has no locking at load time. Thus, the flavor data structures can be damaged if two processes evaluate **defflavor** simultaneously.

Obtaining Information About a System

The Command Processor command Show System Definition and the Lisp function **describe-system** are useful means of finding information about a system.

Show System Definition Command

Show System Definition *system-or-subsystem keywords*

Displays a the system definition of *system* including its current patch level, status (experimental or released), and the files that make up the system.

system-or-subsystem	The system whose definition to display.
keywords	:Detailed, :More Processing, :Output Destination, :Use Journals, :Version
:Detailed	{Yes, No} Whether to display the information about all the component systems of the system or not. The default is No, the mentioned default is Yes.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Use Journals	{Yes, No} Use the information in the system's journal files instead of the information loaded into the world. The default is No, the mentioned default is Yes.
:Version	{*version-designator*} What version of the system for which to display the definition. The default is :Released.

describe-system *system-name* &rest *keys* &key *(:show-files* **t***) :use-journals :system-op :reload :recompile :version :detailed :system-branch* &allow-other-keys

Function

Displays useful information about the system named *system-name*. This includes the name of the system source file, the system package default if any, and component systems. For a patchable system, **describe-system** displays the system version and status, a typical patch file name, the sites maintaining the system, and, if the user wants, a listing of patches.

If **:show-files** is **t** (the default), it displays the history of the files in the system. Other possible values are **nil** (do not show file history) and **:ask** (ask the user).

If **:system-op** is **t**, it displays the operations required to load the system. Other possible values are **nil** (do not display operations) and **:ask** (ask the user).

If **:reload** is **t** (the default is **nil**) the files are reloaded.

If **:recompile** is **t** (the default is **nil**) the files are recompiled.

The default version of the system is **:latest**.

The **:detailed** argument (**t** or **nil**) indicates whether to display the plans for the component systems.

Other useful commands include Show System Modifications and Show System Plan.

Show System Modifications **Command**

Show System Modifications *system-name keywords*

With no arguments, Show System Modifications lists the locally maintained systems present in this world and, for each system, all the modifications that have been loaded into this world. For each modification 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 modification does, as entered by the person who made it.

If Show System Modifications is called with an argument, only the modifications to *system-name* are listed.

system-name	{All, Local, *system-name1, system-name2* ... } The system for which to show modifications. The default is All.
keywords	:Author, :Before, :From, :Matching, :More Processing, :Newest, :Number, :Oldest, :Output Destination, :Reviewer, :Since, :Through

:Author — A name. Show modifications by a particular person. For example:

```
:show modifications system :author kjones
```

would only show those modifications made by the person whose user ID is kjones.

:Before	A date to serve as one limit for modifications to show: `:before 1/23/90`
:From	A number to use as the first modification to show.
:Matching	A string to search for in the comments and show only modifications whose comment contain that string: `:matching namespace`
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "`FUNCTION M`").
:Newest	A number of modifications to show, for instance the ten most recent ones: `:newest 10` Using this keyword without a number is the same as `:newest 1`.
:Number	A number. Show only this particular modification. For example: `Show Modifications :number 6` would show modification number 6.
:Oldest	A number of modifications to show, for instance the ten earliest ones: `:oldest 10` Using this keyword without a number is the same as `:oldest 1`.
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Reviewer	A name. Show modifications reviewed by a particular person.
:Since	A date to serve as one limit for modifications to show.
:Through	A number to use as the last modification to show: `:through 17`

Show System Plan Command

Show System Plan *system operation keywords*

Show the system plan (the order of operations) for the specified *system* under the specified *operation*.

system	The system for which to show the plan.
operation	The operation for which to show the plan. The available operations are:

All	Count-Lines-In	Kludge-Load
Compile	Distribute	Load
Copy	Edit	Load-Patches
Copy-Toolkit-C-Files	Hardcopy	Reap-Protect
Write-Toolkit-C-Files		

keywords	:Date Checking, :Detailed, :More Processing,:Output Destination, :Version
:Date Checking	{Yes, No} Compare files against the file system. The default is No, the mentioned default is Yes.
:Detailed	{Yes, No} Whether to describe the plans for component systems. The default is No, the mentioned default is Yes.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Version	Version of the system for which to construct plans. The default is Released.

Obtaining Information on System Versions

When a Symbolics computer is booted, it displays a line of information telling you what systems are present, and which version of each system is loaded. This information is returned by the function **sct:system-version-info**. It is followed by a text string containing any additional information that was specified by whoever created the current world load. See the function **zl:disk-save**.

sct:system-version-info &optional *brief-format* *Function*

Returns a string giving information about which systems and what versions of the systems are loaded into the machine (for systems that differ from the released versions) and what microcode version is running. A typical string for it to produce is:

```
"System 424.210, Zmail 416.13, LMFS 417.5, Tape 418.9,
microcode 3640-MIC 420, FEP 127"
```

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

sct:get-system-version &optional *(system* **"System"***)* *Function*

Returns three values. The first two are the major and minor version numbers of the version of *system* currently loaded into the machine. The third is the status of the system, as a keyword symbol: **:experimental**, **:released**, **:obsolete**, or **:broken**. *system* defaults to **System**. This returns **nil** if that system is not present at all.

For CLOE, it uses *name*, which may be a symbol, string or system denoting a system, and returns information about the corresponding system. The three returned values are the system major version number, the minor version number, and the system status (such as **:released** or **:experimental**). Note that this function is only available on the 386 side.

```
(get-system-version "FROB") =>
3
2
:experimental
```

sct:get-release-version *Function*

Releases have version numbers and status associated with them, just as systems do. Symbolics staff assign the release number.

sct:get-release-version returns three values, the release numbers and the status of the current world load:

- Major version number
- Patch version number or string describing minor patch level
- Status of the world load as a keyword symbol:
  - **:experimental**
  - **:released**
  - **:obsolete**
  - **:broken**
  - **nil** (when status cannot be determined)

zl:print-system-modifications &rest *system-names* *Function*

With no arguments, this function 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 will always be the same since a world can only contain one major version), the minor version number, and an explanation of what the patch does, as entered by the person who made the patch.

If **zl:print-system-modifications** is called with arguments, only the modifications to *system-names* are listed.

sct:patch-loaded-p *major-version minor-version* &optional *(system* **"System"***)* *Function*

A predicate that tells whether the loaded version of *system* is past (or at) the specified patch level. Returns **t** if:

- the major version loaded is *major-version* and the minor version loaded is greater than or equal to *minor-version*
- the major version loaded is greater than *major-version*

Otherwise, the function returns **nil**.

sct:get-system-input-and-output-source-files *system* &optional *version* &key *:system-branch* *Function*

Returns a list of pairs of the form (input-file output-file) for *system*.

:version	Lists the files for the specified version. If *version* is not specified, returns the **:newest** version of the files.
:system-branch	Reserved for future use.

sct:get-system-input-and-output-defsystem-files *system* &optional *version* &key *:system-branch* *Function*

Returns the system declaration file for a specified *version* of the specified *system*. If *version* is not specified, returns the **:newest** version of the files.

sct:get-all-system-input-files *system* &key *:version :include-components :system-branch* *Function*

Returns a list of the system declaration file (sysdcl) and the input (source) files that make up *system*.

:version	Lists the files for the specified version of the system. If no version is supplied, it returns the **:newest** version of all the files.
:include-components	Includes the list of files for component systems also.

:system-branch Reserved for future use.

sct:check-system-patch-file-version &key *(:system* **"System"***)* *(:major-version* **(sct:get-system-version sct:system)***) :minor-version :file-version*

Function

Checks to see if the patch file to the system, with the specified *major version* and *minor-version*, of *file-version* has been loaded into the world. If *file-version* is **:none**, checks to see that the patch file has never been loaded. If the check fails, it causes an error. Typically, this form is used in a patch file to ensure that a patch to another system has (or has not) been made.

Developing Portable Common Lisp Programs

Introduction to the Common Lisp Developer

The Common Lisp Developer is a tool which assists in developing and testing Common Lisp programs. It is useful when you are planning to port a program to other Common Lisp implementations.

When you are planning to port a program, it is often useful to aim at making that program work in the "Least Common Denominator" of Common Lisp. The Genera environment tends to be highly fault-tolerant and therefore not always predictive of other systems which might be a target of your port. The Common Lisp Developer adds a new dimension to the Genera development environment, by enabling you to define certain programs in an environment that replaces Genera's fault-tolerant interpretation of Common Lisp with a stricter interpretation, which is more predictive of the "Least Common Denominator" of Common Lisp.

The aim of the Common Lisp Developer is to permit you to develop code for any conforming implementation of Common Lisp, as described in *Common Lisp: The Language* (also known as *CLtL*) by Guy L. Steele Jr.. (Note that the Common Lisp Developer currently does not assist in developing code intended to run in an ANSI Common Lisp implementation.)

Because the Common Lisp Developer's interpretation of *CLtL* is more strict than the Genera interpretation of *CLtL*, there is much higher likelihood that programs developed in the Common Lisp Developer will port easily to other Common Lisp environments.

In summary, Symbolics offers two dialects of Common Lisp: a conservative interpretation (the Common Lisp Developer's CLtL syntax) and a liberal interpretation (Genera's Common-Lisp syntax). When developing programs to ship outside the Symbolics environment, the conservative approach will help you plan for the harsh realities of life outside. The CLtL syntax is a good donor environment. But for programs you do not plan to port, or for programs that were written elsewhere, the fault-tolerant environment (Common-Lisp syntax) is better. The Common-Lisp syntax is a good recipient environment. Because of the conflicting requirements of donor and recipient environments, it is tricky for a single environment to satisfy both needs; this is why Symbolics provides two environments.

For detailed information about differences between the two dialects of Common Lisp, see the section "Technical Details: Differences Between CL and CLtL".

For historical information about where Common Lisp Developer came from, see the section "Relationship of Common Lisp Developer and Cloe".

Nature of the Common Lisp Developer

The Common Lisp Developer is not a program like Zmail or Frame-Up; it is not tied to a particular window. Instead, it is a substrate which is accessible in various different ways from existing programs.

Unlike some code analysis tools which employ only static techniques to determine conformance, the Common Lisp Developer uses a mixture of static and dynamic techniques. For example, you can get both *compile-time* diagnostics about wrong numbers of arguments and *runtime* checking of declarations. (The variable **cltl-i:*enforce-type-declarations*** controls this behavior. See the variable **cltl-i:*enforce-type-declarations***.)

Working in the Common Lisp Developer can be characterized as being like working in Genera with blinders on. You have access to all the Genera tools, but you are kept focused on the business at hand: developing portable programs. The Common Lisp Developer keeps you from accidentally using Symbolics features which probably will not be available in your delivery environment.

How the Common Lisp Developer Works

The Common Lisp Developer provides a new syntax called CLtL, with a set of associated packages.

Both the CLtL and Common-Lisp syntaxes have a lisp package, but the packages are different. Where the symbols have the same meaning, they are shared between the two packages. For example, CLtL's **car** is the same symbol as Common-Lisp's car. However, **cltl:if** is different than Common-Lisp's **if**, because the CLtL version does not have the extended-else body offered in the normal Genera Common-Lisp implementation. The reason for this should be obvious: the extended-else body is useful, but it is not available everywhere. Programs which depend on its presence are not likely to port easily. The guiding principle of the Common Lisp Developer is "Better to find out now than later."

Since both CLtL and Common-Lisp have a package called lisp, we will sometimes speak of a package universe. This is the currently active global mapping from names of packages to package objects. So, for example, if you enter lisp:if in a Lisp Listener, you will get the sym- bol cltl:if if you are in CLtL syntax or cl:if if you are in Common- Lisp syntax.

For more information about packages, see the section "How Package Universes Work".

How to Access the Common Lisp Developer

First, you must statically decide which environment your code will live in. For example, you cannot type

```
(defun double (x) (declare (fixnum x)) (* x 2))
```

to a Lisp Listener and then later decide you want to run the program under the Common Lisp Developer. The rules are basically the same as they are for the separation between Zetalisp and Common-Lisp. Another way to view the CLtL syntax (or the Common Lisp Developer as a whole) is as a third dialect that has been added alongside the original two.

You can access the developer in any of these three ways:

- Create files with an attribute list that specifies the CLtL syntax. The syntax line should contain the following:

  ```
  -*- Syntax: CLtL; ... -*-
  ```

  You can use the Set Lisp Syntax (`m-X`) command in the editor (when you are in Lisp mode) to enable this syntax.

- Enable CLtL syntax globally by using the Set Lisp Context command when typing to the command processor in a Lisp Listener.

- Name individual symbols in the CLtL package universe without entering the CLtL universe by using the escape sequence ":::" in a symbol. For example, when typing to a Lisp Listener, regardless of whether you are in Common-Lisp, Zetalisp, or CLtL syntax, the symbol **cltl:::lisp:if** denotes **if** in CLtL's package universe. This technique can also be used to escape back to Common-Lisp or Zetalisp from the CLtL syntax, but be aware that using this syntax indiscriminately blurs important modularity boundaries and can lead to trouble when it comes time to port.

CLtL-Only and CLtL Syntax

Common Lisp Developer provides two syntaxes, which are almost the same. The two syntaxes are needed because of a small but important problem which comes up a lot in porting. The following example illustrates the problem.

Claim	Nothing in Common Lisp states that the following expression is non-conforming: `(defun choose-em (x) (tv:menu-choose x))`
Realization	Things explicitly permitted or forbidden by *CLtL* are not the only source of portability headaches. There is no *a priori* way to distinguish whether the program **choose-em** is relying on Genera functionality to make **tv:menu-choose** available, or whether the user in good faith plans to port only to implemen-

tations which provide **tv:menu-choose**, or whether the user in fact plans to develop **tv:menu-choose** for implementations which do not have such a package and/or function.

Therefore, for some purposes we want to be able to use the CLtL environment only to "get at things which are specified by CLtL". For other purposes, we want to use the CLtL environment to lock out those things which are not specified. The CLtL syntax differs from the CLtL-Only syntax in only one way: the package universe of CLtL-Only does not contain the initial Genera packages such as TV, DW, ZWEI, and so on.

Interactively, it may be useful to get to these packages. Short of setting the Lisp Context back to Common-Lisp, you can escape from the CLtL-only universe and gain access to **tv:make-window**, for example, by using **cl:::tv:make-window**. Of course code that calls **tv:make-window** is not likely to be portable. (For suggestions about designing code that needs to use some non-portable features, see the section "Developing Code for a non-Symbolics Lisp with Extensions".)

Another pragmatic reason why you might like to get to these packages is that you may prefer not to see the Debugger type every symbol from the Genera universe as **zl:::tv:this** or **zl:::dw::that**. In files, it is almost always better to lock out the Genera packages and to ask for specific features by name as you need them to keep a tight reign on your modularity, and so you'll best be able to anticipate porting costs.

We recommend that you always use

```
-*- Syntax: CLtL-Only; ... -*-
```

in files so that you do not accidentally become dependent on Genera functionality without having realized it. Interactively, you might want to use

```
Set Lisp Context CLtL
```

rather than

```
Set Lisp Context CLtL-Only
```

because it gives you slightly more convenient interactive access to functions in Genera that you might want to call when debugging. Note that these are only recommendations, and you might prefer a different workstyle.

Using #+ and #- to Distinguish the Common Lisp Developer

You can use **#+CLTL** and **#-CLTL** to identify the Common Lisp Developer environment. Since this is intended to be a fairly conservative simulation, we expect that other vendors might offer this same feature if they also offer a similarly conservative interpretation of CLtL. You can use **#+(and Symbolics CLtL)** in the rare case when you need to distinguish the particular idiosyncrasies of the Symbolics Common Lisp Developer environment.

For some uses, you can also use **#+Genera** and **#-Genera**, given that the Genera feature is not present in the CLtL syntax.

As should be obvious, the variable **cltl:*features*** is not **eq** to ***features***. The former controls the #+ and #- macros when in CLtL syntax, the latter when in Zetalisp or Common-Lisp syntax. (For historical reasons, the Zetalisp and Common-Lisp syntaxes cannot be distinguished with #+ and #-).

Avoiding a common pitfall: For most purposes, it is appropriate for an application to simply do **(push** *key* ***features*)** without regard to what environment it is loading into. Since the two environments are largely separate, this will cause ***features*** to get set correctly when you load things into Common-Lisp and **cltl:*features*** when you load things into CLtL. In general, your goal should be to put something on a features list only to announce to other programs that some interesting feature is accessible to them. Except in very unusual cases, loading something into CLtL does not make things available to Common-Lisp, or vice versa, so it is not necessary to push a feature onto both features lists in most cases.

Developing Code for a non-Symbolics Lisp with Extensions

Sometimes you develop code to run in an environment which is extended in some way beyond straight *CLtL* but not in the same way as Genera.

For example, suppose you want to develop code for a Lisp which offers a menu choice facility that has an interface like:

(menu:choose *objects prompt*)

where *objects* is an uninterpreted list of objects, not an item list as in Genera, and where *prompt* is just a string. We recommend that you isolate the part of your application which is not covered by CLtL into a single file or a small set of files. When you are ready to port the program, you can then focus on these files. Such a file might contain definitions like:

```
;-*- Mode: LISP; Syntax: CLtL; Package: AARDVARK -*-

(defun choose-one-of (objects prompt)
  #+symbolics
  (zl:::dw:menu-choose
    (mapcar #'(lambda (x)
                (list (zl:::dw:present-to-string x) :value x))
            objects)
    :prompt prompt)
  #+acme
  (menu:choose objects prompt)
  #-(or acme symbolics)
  (error "implementation does not support choose-one-of."))
```

The idea is to isolate the non-portable references (for example, to **dw:menu-choose** or to **menu:choose**) in one file, and to use portable references (for example, **aardvark::choose-one-of**) everywhere else in the application. In general this methodology can help keep porting issues under control, although not all problems can be decomposed this simply.

In some cases you need to make an entire package available, with the idea that on other machines you will provide equivalent functionality in some other way. The function **si:import-genera-package** is useful for this case.

si:import-genera-package *genera-package-name* &optional *target-name target-nicknames target-syntax-name* *Function*

Imports *genera-package-name* into the current package universe, or the package universe given by *target-syntax-name*. Within that universe the name chosen will be the same as *genera-package-name* unless overridden by *target-name* and *target-nicknames*.

Since this function exists outside the CLTL package universe, and it is typically used when you are in the CLTL package universe, you usually call it as follows:

```
(zl:::si:import-genera-package "TV")
```

Technical Details: Differences Between CL and CLtL

- In CLtL syntax, the package named LISP denotes CLTL, and CL denotes Genera's CL package. In CLtL-Only syntax, LISP denotes CLtL and there is no CL package.

- *CLtL* says that some functions require a certain number of arguments, but Genera (that is, the Common-Lisp syntax) allows some of those arguments to be optional. Also, in some cases, Genera allows additional (`&OPTIONAL` or `&KEY`) arguments that are not sanctioned by *CLtL*. In many cases, the CLTL package shadows the symbols in question and provides a function that requires the same argument convention as is specified in *CLtL*.

- The definitions of **cltl:lambda** and **cltl:defun** handle `&REST` arguments correctly (by copying stack-allocated lists into the heap). See *CLtL*, p59 and p67. There are two variables that control rest list copying: **cltl-i:*copy-&rest-lists-in-heap*** and **cltl-i:*copy-&rest-lists***. For related information, see the section "Performance Considerations of Common Lisp Developer".

- The **functionp** function returns true for symbols only if they have a global function definition; that is, if (**fboundp** *symbol*) would return true. The **cltl:functionp** function returns true for any symbol argument. See *CLtL*, p76.

- The **funcall** and **apply** functions allow you to call an array as a function. The **cltl:funcall** and **cltl:apply** functions allow you to call only those objects for which **cltl:functionp** would return true. See *CLtL*, p76 and p107.

- The **if** special form allows more than three arguments. The **cltl:if** special form requires two or three arguments, erring if additional arguments are supplied. See *CLtL*, p115.

- The **loop** facility allows control of iteration with keywords. The **cltl:loop** macro does not allow atoms in the loop body because no standard interpretation of such keywords has been established. See *CLtL*, p121.

- The default package to **:use** in the functions **in-package** and **make-package** is `"CL"`. The default package to **:use** in the functions **cltl:in-package** and **cltl:make-package** is the package which the Common Lisp Developer calls **lisp** and Genera calls **cltl**. See *CLtL*, p183.

- The CL implementation of many functions allows them to accept some types of arguments that are not guaranteed to be accepted by *CLtL*. The CLTL package shadows such symbols and provides definitions which are more compatible with a strict reading of *CLtL* (that is, errors are signalled at runtime when the stricter interpretation has been violated).

  ° The functions **export**, **unexport**, **import**, **shadowing-import**, and **shadow** allow strings in place of symbols in some cases where *CLtL* says that symbols are required. See *CLtL*, p186.

  ° The functions **rename-package**, **intern**, **find-symbol**, and **unintern** allow package names in place of packages in places where *CLtL* does not define the behavior. See *CLtL*, pp184-185.

  ° The functions **string=**, **string<**, **string>**, **string<=**, **string>=**, **zl-user:string//=**, **string-equal**, **string-lessp**, **string-greaterp**, **string-not-greaterp**, **string-not-lessp**, **string-not-equal**, **string-trim**, **string-right-trim**, **string-left-trim**, **string-upcase**, **string-downcase**, and **string-capitalize** are defined by *CLtL* to accept strings and symbols, but the default CL implementation allows character arguments as well. See *CLtL*, pp299-303.

- The functions **char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-equal** are not compatible with the descriptions as stated in *CLtL* because they do not ignore the bits information of the characters they are comparing. The functions **cltl:char-equal**, **cltl:char-not-equal**, **cltl:char-lessp**, **cltl:char-greaterp**, **cltl:char-not-greaterp**, and **cltl:char-not-equal** strictly observe the *CLtL* description, even though the Common Lisp designers generally seem to believe that the *CLtL* description is a design mistake. See *CLtL*, p239.

- The symbols **prin1**, **evalhook**, and **applyhook** have both function and variable definitions, though *CLtL* defines only the function definition. **cltl:prin1**, **cltl:evalhook**, and **cltl:applyhook** are shadowed symbols in order to allow only the function definition to be accessible. See *CLtL*, p323 and p383.

- The **open** function and **with-open-file** macro allow many non-standard options. The **cltl:open** function and **cltl:with-open-file** macro allow only those options which are portable. See *CLtL*, p418-422.

- The macros **trace** and **untrace** accept complicated argument patterns which are not defined by Common Lisp. The **cltl:trace** and **cltl:untrace** macros signal an error if their arguments are non-atomic. See *CLtL*, p440.

  Note that although this checking for **trace** and **untrace** may be helpful for code which calls trace, it is probably an inconvenience in interactive code.

  The option variable **cltl-i:*extended-trace-enable*** (which defaults to **nil**) may be set to **t** to make **cltl:trace** behave like **trace**. Most users will probably want to consider enabling this option in an init file.

- The functions **cltl:apropos** and **cltl:apropos-list** take only the arguments defined in *CLtL*. See *CLtL*, p443.

- Uses of type declarations in **declare** are enforced at runtime in most variable binding situations (**cltl:lambda**, **cltl:defun**, **cltl:let**, and so on). For related information, see the variable **cltl-i:*enforce-type-declarations*** and see the section "Performance Considerations of Common Lisp Developer".

- The function **cltl:ash** requires that its first argument be an integer. The function **ash** accepts either an integer or a float as a first argument.

- The **cltl:flet** macro does not permit declarations by default. The variable **cltl-i:*allow-declarations-in-flet*** controls this; its default is **nil**.

- The symbol **cltl:ignore** is unlike the symbol **ignore** in that it is only defined as a function. Neither is **cltl:ignore** treated magically as a variable name, nor is it a predefined function.

- The symbol **cltl:lambda** is unlike the symbol **lambda** in that it is recognized only by the **function** special form. **cltl:lambda** is not a predefined macro.

- The function **cltl:documentation** accepts as a second argument only the standard documentation keywords: **variable**, **function**, **structure**, **type**, and **setf**.

- The functions **cltl:adjust-array** and **cltl:vector-push-extend** will signal an error if the given array was not created by specifying **:adjustable t**. For related information, see the variable **cltl-i:*adjustable-arrays-being-recorded*** and see the section "Performance Considerations of Common Lisp Developer".

- The following functions permit their pathname arguments to be symbols, whereas their CL counterparts would signal an error in that case: **cltl:pathname**, **cltl:truename**, **cltl:merge-pathnames**, **cltl:pathname-host**, **cltl:pathname-device**, **cltl:pathname-directory**, **cltl:pathname-name**, **cltl:pathname-type**, **cltl:pathname-version**, **cltl:namestring**, **cltl:file-namestring**, **cltl:directory-namestring**, **cltl:host-namestring**, **cltl:enough-namestring**, **cltl:directory**, **cltl:load**, and **cltl:compile-file**.

- The variable **cltl:*features*** contains significantly fewer symbols than does does ***features***, and some of them are different. For example, CLtL has a **:cltl** feature which CL does not. CL has a **:genera** feature which CLtL does not. Since this list is coordinated with #+ and #-, you can read-conditionalize code for the Common Lisp Developer.

- The functions **cltl:software-type**, **cltl:software-version**, **cltl:machine-type**, **cltl:machine-version**, **cltl:machine-instance**, **cltl:lisp-implementation-type**, and **cltl:lisp-implementation-version** return different values than their CL counterparts.

- The function **cltl:describe** returns no values. The function **describe** returns its argument.

- The function **cltl:make-echo-stream** is implemented.

- **cltl:make-concatenated-stream** returns a stream which does not treat an EOF in one of the given streams as a token break.

- **list-all-packages** and **do-all-symbols** work locally with respect to the CLTL package universe when in CLtL-Only (but not CLtL) syntax. (The decision of which action to take is made dynamically at runtime.)

Performance Considerations of Common Lisp Developer

The goal of the Common Lisp Developer is to provide you with correct code; speed of the correct code under the Developer was not a primary goal. That is not to say that the Developer is unbearably slow, but in some cases speed has been traded for additional checking.

In some large applications we have run, the typical loss in speed of using CLtL over using the Genera's Common-Lisp dialect is about 20 percent. However, this number could vary widely depending on what language features you use. In some cases you may see negligible loss in speed, while in others you may see a higher loss.

In cases where an error-check was likely to result in a noticeable speed cost, we have tried to provide you a way to selectively disable the checking. That way, you can disable checks that you decide are not worth the performance penalty.

The primary places where speed was traded for error checking are documented below:

- Declaration checking

  For example:

```
(defun f (x) (declare (float x)) x)
(f 1)

ERROR: The value of X in F, 1, was of the wrong type.
       The function expected a floating point number.
```

The variable **cltl-i:*enforce-type-declarations*** can be used to control which declarations are being checked.

Implementation Note: Declarations are currently checked only at binding time (and at re-assignment time in a **do**). Declarations are not currently re-tested when a **setq** is done.

Note also that the checking is dynamic, not lexical. In some cases, this can make for much more intelligible diagnostics than you might get by other means:

```
(DO ((I (- MOST-POSITIVE-FIXNUM 4) (+ I 1)))
    (NIL)
  (DECLARE (FIXNUM I))
  (PRINT I))
2147483643
2147483644
2147483645
2147483646
2147483647
Error: The value of I in SI:*EVAL, 2147483648, was of the wrong type.
       The function expected a fixnum.
```

- **Rest list consing**

  For example:

  ```
  (DEFUN F (&REST X) X)
  (F 'A 'B 'C) => (A B C)
  ```

  By default, rest list copying is done even in cases where you might wish it could be lexically determined that it is not needed. There are two variables that control rest list copying: **cltl-i:*copy-&rest-lists-in-heap*** and **cltl-i:*copy-&rest-lists***.

- **Checking for adjusting arrays**

  Common Lisp Developer checks for violations of attempts to adjust arrays that are created with **:adjustable nil** (or with no **:adjustable** option).

  In Genera's Common-Lisp implementation of arrays, all arrays are adjustable regardless of the use of **:adjustable**. (This is conforming under a strict reading of *CLtL*, however some have argued that this is not the standard interpretation.)

The CLtL implementation takes the alternate point of view — that no array should be adjustable unless **:adjustable t** was used. The Common Lisp Developer's information about adjustable arrays is recorded in a table. Any array not recorded in the table is assumed to be non-adjustable (unless this recording is disabled, in which case all arrays are reverted to being assumed adjustable). Since there are both speed and space concerns to this technique, the feature can be disabled by using the variable **cltl-i:*adjustable-arrays-being-recorded***, whose default is **t**.

Variables That Affect Common Lisp Developer Behavior

cltl-i:*adjustable-arrays-being-recorded* *Variable*

Controls whether or not Common Lisp Developer checks for violations of attempts to adjust arrays that are created with **:adjustable nil** (or with no **:adjustable** option).

The variable **cltl-i:*adjustable-arrays-being-recorded***, enables you to disable this feature. The default value of this variable is **t**. You can set it to **nil** to disable this feature.

Note that if you set this variable to **nil**, we recommend that you do not later set this variable back to **t**. If you do set it back to **t**, any arrays created with **:adjustable t** during the time while the variable was **nil** will not appear to be adjustable.

This decision is made dynamically at runtime.

cltl-i:*allow-declarations-in-flet* *Variable*

This variable, used by Common Lisp Developer, controls whether **cltl:flet** permits declarations by default. The default is **nil**.

This decision is made at semantic resolution (that is, compile or macroexpand) time.

cltl-i:*copy-&rest-lists* *Variable*

This variable, used by Common Lisp Developer, controls whether `&REST` lists have indefinite or dynamic extent. Its default is true.

If this variable is true, `&REST` lists always have indefinite extent (as specified in *CLtL*).

If this variable is false, `&REST` lists always have dynamic extent (as done in Zetalisp).

This decision is made at semantic resolution (that is, compile or macroexpand) time.

cltl-i:*copy-&rest-lists-in-heap* *Variable*

This variable, used by Common Lisp Developer, controls whether &REST lists are permitted to share structure with the list used in the call even when those lists are already in the heap. Its default is true.

If this variable is true, &REST lists are not permitted to share structure with the list used in the call even when those lists are already in the heap. (This can happen only when &REST is used.) If false, sharing might or might still occur. The interesting test case is:

```
(let ((l '(a b c)))
  (not (eq (apply #'(cltl:lambda (&rest x) x) l) l)))
```

If this variable is true, the above expression returns true. If this variable is false, then the above expression might return either true or false.

This decision is made at semantic resolution (that is, compile or macroexpand) time.

cltl-i:*enforce-type-declarations* *Variable*

Common Lisp Developer does runtime checking of declarations. This variable is a list of the types which are checked at runtime by Common Lisp Developer.

The decision to enforce these declarations is made at semantic resolution (that is, compile or macroexpand) time, but the enforcement itself occurs at runtime.

Tuning an Application in the Common Lisp Developer

You should generally not try to tune an application in the Common Lisp Developer (or any development environment). Tuning should be done for each target implementation by running the program within that implementation. Issues of varying paging performance, memory configuration, data formats, compiler optimizations, and so on make it impossible for the Common Lisp Developer to simulate the application's native performance.

Moreover, the error checking offered by the Common Lisp Developer may skew your intuitions about your program's speed even further. For example, the Developer will (by default) check the validity of declarations at runtime, slowing things down somewhat in the hope of detecting bugs. Yet when your program is delivered to its target environment, the same declaration will increase, not decrease, the speed of your program. The reason the program will run faster is that it will compile out checking; this is the reason that it is important to get this checking at development time, and the reason that the Developer prefers to let your program run a little slower just to assure that the checking really gets done.

Relationship of Common Lisp Developer and Cloe

The Cloe Developer and the Common Lisp Developer are similar in purpose. They differ primarily in their intended target and in the array of features offered.

The target implementation of the Common Lisp Developer is very broad; code developed in it will typically run in any conforming Common Lisp implementation.

The target implementation of the Cloe Developer is the Cloe Runtime lisp, a software-only delivery option sold by Symbolics which runs on many 386-based machines under either MS-DOS or UNIX.

A key difference is that any good development environment for code to be ported is constrained to provide you only the least common denominator of the two systems you are working on. The Cloe dialect of Lisp is a superset of Common Lisp, so the least common denominator of Cloe and Genera is a richer environment than the least common denominator of Common Lisp and Genera. As such, the Cloe Developer offers you more power.

Another key difference is that both the Cloe development environment and the Cloe delivery environment are Symbolics products. This means that we can keep the features of the two carefully aligned to minimize porting problems. This also means that additional tools can be provided which specifically anticipate (and help you work around) standard kinds of problems which we know you will encounter. For example, the migration tools of the Cloe Developer will take an application which is defined in SCT and automatically copy its source files, journal files, and so on to the MS-DOS or UNIX environment and set up everything so that you can simply compile and load the system on the 386 without any special effort.

Symbolics provides you with two important software delivery options. The specific delivery option you choose should be driven by the specific technical and delivery needs of your application. You can develop in the Common Lisp Developer and deliver in an arbitrary Common Lisp, or you can develop in the Cloe Developer and deliver specifically in Cloe. The former offers you a broader base of targets but loses the ability to take advantage of specific information about the target to make life easy for you in some ways. The latter offers you a more limited number of target implementations, but takes advantage of the constraint by exploiting additional features known about those delivery options.

And, of course, since the Cloe Runtime system is a Common Lisp, you could develop in the Common Lisp Developer and use the Cloe Runtime system as one of several delivery vehicles. In that case, of course, the least common denominator rule keeps you from being able to take advantage of Cloe's extended features (except by conditionalized escapes), but that may still be satisfactory for some applications. You should not ignore Cloe as an option just because you are constrained to deliver in other vendors Lisps as well.

The Compiler

Introduction to the Compiler

The purpose of the Symbolics Lisp compiler is to convert Lisp functions into programs in the Symbolics computer's instruction set. Compiled functions run more quickly and take up less storage than interpreted code. They are executed directly by the machine. The compiler checks for errors and issues warnings regarding faulty syntax, typographical errors, and undeclared variables. Because the compiler does all this checking, as well as the fact that compiling code does not lose any run-time checking, most users debug their programs in compiled form rather than debugging them in interpreted form and compiling them after they work.

How to Invoke the Compiler

You can invoke the compiler in several ways.

- Use one of several Zmacs commands to compile regions of Lisp code in an editor buffer to your Lisp environment. Some of the most common commands are Compile Region (`m-X`) (`c-sh-C`), Compile Changed Definitions of Buffer (`m-X`) (`m-sh-C`), and Compile Buffer (`m-X`). See the section "Compiling Lisp Programs in Zmacs".
- Call the function **compile** to compile an interpreted function in the Lisp environment. Compiling an interpreted function in a Dynamic Lisp Listener converts the function into a compiled code object in memory. Programmers occasionally compile interpreted functions to examine the code generated by the compiler. To examine a compiled function in symbolic form, use the **disassemble** function.
- Use **compile-file** and related functions, Compile File (`m-X`), or Compile File at the Command Processor prompt to translate source files into compiled code files.

- Invoke **compile-system** or type Compile System at the Command Processor prompt to compile and load large programs, usually consisting of many files.

Caveats on Using the Compiler

Circular structures cannot be dumped by the **bin** file dumper.

Circular structures may not be constants in compiled functions.

Users should not name arguments or variables **self**. The compiler has a special way of dealing with variables named **self**, because it uses that name when compiling generic functions for structures and flavors.

Structure of the Compiler

The Lisp compiler is actually composed of three distinct pieces of software:

- The stream compiler

- The function compiler

- The **bin** (binary) file dumper

The stream compiler accepts a stream of top-level Lisp forms and processes them. These forms are usually read from a stream of characters, which can be either a file or part or all of an editor buffer. The stream compiler passes forms recognized as function definitions through the function compiler. Certain other forms are also processed specially: See the section "How the Stream Compiler Handles Top-level Forms". Stream compiler output can be sent either to the Symbolics computer's virtual memory or to a file (via the **bin** file dumper) for later loading.

The function compiler takes a Lisp function and translates it from Lisp expressions into machine instructions. Its job includes expanding macros, performing optimizations, recognizing special forms, and recognizing calls to functions that have corresponding machine instructions. The function compiler is available to use by itself as the **compile** function; it is also called by the stream compiler.

The **bin** file dumper accepts a stream of Lisp forms and machine-instruction function definitions (compiled function objects) and writes them into a file in a compact form understood by the loading function (**zl:load**). The **bin** file dumper is available for use by itself as the **sys:dump-forms-to-file** function; it is also called by the stream compiler.

Different combinations of these compilers are available:

- The function compiler can be used by itself (via the **compile** function).

- The **bin** file dumper can be used by itself (via the **sys:dump-forms-to-file** function).

- The stream compiler can be used with the function compiler (`c-sh-C` or related Zmacs commands).

- All three compilers can be used (via **compile-file**, **compile-system**, or the Command Processor's Compile System command).

The following diagram shows the relationship of the different compilers to one another.

```
                a stream
                    ↓
             STREAM COMPILER
              ↓           ↓
function definitions      other forms
(such as defuns)          (such as defvars)
       ↓                         |
FUNCTION COMPILER                |
       ↓                         |
compiled function                |
objects                          |
       |                         |
       |                         |
       |----->----\ /----<-------|
       |           |             |
       |           |             |
       |           |             |
       |           |             |
       |    BIN FILE DUMPER      |
       |           ↓             |
       |    compiled code file   |
       |           ↓             |
       |         LOAD            |
       |___________|_____________|
                   |
                  EVAL
                   ↓
             virtual memory
```

The Genera tools you use to invoke compilation determine the path through the diagram. For example, suppose you run the **compile-file** function on a Lisp source file. The function calls the stream compiler, which in turn calls the function compiler on any function definitions in the file. The function compiler passes the resulting compiled function objects to the **bin** file dumper. Some forms are passed directly to the **bin** file dumper (middle of the diagram) without being processed through the function compiler. All output from the **bin** file dumper is sent to a compiled code file. Loading that file creates the effect of compiling the source code directly to virtual memory.

For example, rather than compiling the source file, read it into an editor buffer and compile the entire buffer via the Zmacs command Compile Buffer (m-X); the output from the stream compiler and function compiler is evaluated immediately. The point is that while these two methods of compilation operate completely differently, the effect is the same once the results are in virtual memory.

How the Stream Compiler Handles Top-level Forms

The stream compiler accepts a stream of top-level Lisp forms and processes them. These forms are usually read from a stream of characters, which can be either a file or part or all of an editor buffer. The stream compiler categorizes these forms according to the table below and processes each according to its category. It calls the function compiler to translate a form that defines a function into a compiled function object containing compiled instructions. Certain other categories of forms are also processed specially, as documented in Table 1.

The stream compiler remembers certain "declarations" for the duration of the compilation. For example, when it compiles a macro definition, it saves the macro definition for use in processing subsequent top-level forms and function bodies. This permits a macro definition different from the one installed in the Symbolics computer's virtual memory to be used during compilation. Other kinds of "declarations" are also saved; most of these are documented in Table 1. The duration of the compilation during which these "declarations" are saved is usually a single invocation of the stream compiler, but when a system is being compiled (a program declared via **defsystem**) the declarations are in effect for the entire compilation, regardless of how many files in the system are compiled.

Stream compiler output can be sent either to the Symbolics computer virtual memory or to a file (via the **bin** file dumper) for later loading. This output can be regarded as a stream of forms that are evaluated either immediately, during the compilation, or later, when the **bin** file is loaded, depending on the type of compilation.

Table 1. Lisp Forms that Require Special Processing by the Compiler.

----

1. DEFINITIONS

Function Definitions, such as **(defun** *function-spec arguments body*...**)**, **(defselect**...**)**, and **(defmethod**...**)**

> The stream compiler calls the function compiler to translate the function definition into a compiled function object. The result is to define the *function-spec* to be the compiled function object. See the function **fdefine**.

Macro Definitions, such as **(defmacro...)**

> The stream compiler saves the definition of the macro for the duration of the compilation, and calls the function compiler to translate the function definition into a compiled function object. The result is to define the *function-spec* to be a macro whose expander function is the compiled function object. See the function **fdefine**.

Substitutable Function Definitions, such as **(defsubst...)**

> The stream compiler saves the definition of the substitutable function for the duration of the compilation, and calls the function compiler to translate the function definition into a compiled function object. The result is to define the *function-spec* to be the compiled function object. See the function **fdefine**.

Variable Definitions, such as **(defvar...)**, **(defparameter...)**, **(zl:defconst...)**, **(defconstant...)**, and **(defvar-standard...)**

> The stream compiler saves the declaration of the variable as a **special** variable for the duration of the compilation. It passes the form through as the compiler's output.

Generalized Function Definitions: **(def...)** and **(deff...)**

> The stream compiler processes each subform of **def** after the initial function spec as a top-level form.
>
> The stream compiler passes a **deff** form through as its output and remembers that it defines a function.

Other Definitions, such as **(defstruct...)**, **(defflavor...)**, **(defpackage...)**, and **(defsystem...)**

> The processing of each type of definition is idiosyncratic. The behavior of the stream compiler for these definition types is defined using the extension mechanisms discussed in this table, principally macro expansion.

2. COMPILER-SPECIFIC FORMS

(progn *form form*...**)**

> Each *form* is processed as a top-level form. Any macro that expands into multiple top-level forms uses **progn** to arrange for the stream compiler to process all of the forms. See the section "Macros Expanding into Many Forms".

(eval-when (*time time*...**)** *form form*...**)**

> Each *form* is processed under the control of the list of *times*. If **load** is one of the *times*, the stream compiler processes each *form* as a top-level form. If **compile** is one of the *times*, each *form* is evaluated during the compilation.

(compiler-let ((*var val***)...)** *form*...**)**

> Each *form* is processed as a top-level form, with the specified bindings of **special** variables in effect.

(*function args*...**)** where the symbol *function* has a **compiler:top-level-form** property.

> The value of the property must be a function of one argument. This function controls the behavior of the stream compiler.

3. DECLARATIONS

(declare *form form*...**)**

> The stream compiler considers each *form*. If it invokes **special** or **unspecial**, the compiler handles it as if it had appeared at top level. Otherwise, the compiler simply evaluates *form*.

Use of **declare** in this way is considered to be an obsolete Maclisp-compatibility feature. Declaring special variables in a top-level **declare** form is not advisable because this hides the variables from the intepreter, which uses **special** declarations in the same way as the compiler. It is preferable to declare **special** variables with an appropriate special form (such as **defvar**) that is understood by both the compiler and the interpreter, or by using **special** as a top-level form without enclosing it in **declare**, or by including a (**declare** (**special** ...)) form inside the body of each function that uses the variable.

Forms to be evaluated at compile time should be specified with **eval-when** rather than **declare**. The stream compiler recognizes a top-level (**declare** *form1 form2*...) as equivalent to (**eval-when** (**compile**) *form1 form2*...) and evaluates *form1*, *form2*, and so on; if the car of *form* is **special** or **unspecial**, then that form is equivalent to (**eval-when** (**compile load**) *form*). Forms appearing within a top-level **declare** should be valid top-level forms. Typical special forms that might appear are **special**, **unspecial**, ***expr**, ***lexpr**, and ***fexpr**.

(zl:local-declare (*declaration declaration***...)** *form form***...)**

The stream compiler processes the *forms* as top-level forms, with the specified *declarations* in effect. **zl:local-declare** is considered to be an obsolete feature; use **declare** inside function bodies instead.

(zl:special *variable variable***...)** and **(zl:unspecial** *variable variable***...)**

The stream compiler saves the declaration for the duration of the compilation and outputs the form unchanged.

4. OTHER FORMS

Macro Invocations

The stream compiler expands each top-level form that invokes a macro before further considering that form. Thus macro expansion can be used to extend the behavior of the stream compiler. Many definition forms are implemented by macros that expand into simpler definitions and other forms. For example, the expansion of such a macro might look like

```
(progn
  (record-source-file-name 'name 'type)
  (eval-when (compile)
    things to do at compile time)
  (defun ...))
```

For additional examples, use **mexp** to examine the expansion of **defvar**, **defsubst**, and **defstruct** forms.

Ordinary Forms

If the stream compiler does not recognize a form, it simply outputs the form unchanged.

Forms Protected From the Compiler

To prevent the stream compiler from recognizing a form, if for some reason it is necessary to pass the form unchanged through the compiler, the safest way is to conceal it inside an **eval** form. For example, the following form prevents the **foo** function from being converted into a compiled function object.

```
(eval (quote (defun foo (x) ...)))
```

Ignored Forms

The stream compiler ignores atoms (both variables and constants), **(quote** *x***)**, and **(zl:comment...)**. It outputs no form when one of these appears in its input.

For Maclisp compatibility a number of top-level declaration forms are provided, including **zl:special**, **zl:unspecial**, **zl:*expr**, **zl:*lexpr**, and **zl:*fexpr**.

----

special &rest *symbols* *Special Form*

Declares each of the *symbols* to be "special" for the Lisp system (for example, the interpreter and the compiler). Provided for Maclisp compatibility. Note: **defvar** is usually preferred over **special**.

zl:unspecial &rest *symbols* *Special Form*

Removes any "special" declarations of the *symbols* for the Lisp system (for example, the interpreter and the compiler). Provided for Maclisp compatibility.

Controlling the Evaluation of Top-level Forms

Sometimes you want to override the stream compiler's default behavior. For example, you might want a form to be put into the compiled code file (compiled, of course), or not; evaluated within the compiler, or not; or evaluated if the file is read directly into Lisp, or not. To tell the stream compiler exactly what to do with a form, use the general **eval-when** special form.

eval-when *times-list* &body *forms* *Function*

Allows you to tell the compiler exactly when the *body* forms should be evaluated. *times-list* can contain one or more of the symbols **load**, **compile**, or **eval**, or can be **nil**.

The interpreter evaluates the *body* forms only if the *times-list* contains the symbol **eval**; otherwise **eval-when** has no effect in the interpreter.

If symbol is present	*Then forms are*

load	Written into the compiled code file to be evaluated when the compiled code file is loaded, with the exception that **defun** forms put the compiled definition into the compiled code file.
compile	Evaluated in the compiler.
eval	Ignored by the compiler, but evaluated when read into the interpreter (because **eval-when** is defined as a special form there).

Example 1: Normally, top-level special forms such as **defprop** are evaluated at load time. If some macro expansion depends on the existence of some property, for example, *constant-value*, the definition of that property must be wrapped inside an **(eval-when (compile) ...)** so that the property is available at compile (macro expansion) time.

```
(eval-when (compile load eval)
  (defprop three 3 constant-value))
```

Example 2: **eval-when** should be used around **defconstant**s of complex expressions. This is because the compiler does not maintain an environment acceptable to **eval** containing **defconstant**s

```
(eval-when (compile load eval)
  (defconstant name expr))
```

In other words, if you are sure that (1) evaluating the *expr* in the global environment gives the correct results, and (2) that no harm is done by changing the current environment to have the (possibly new) value of *name*, then you can use the global environment as a substitute for the compilation environment.

In addition to **eval-when**, the **compiler:top-level-form** property provides another means for overriding the default behavior of the stream compiler.

compiler:top-level-form *Property*

Provides a way to extend the behavior of the stream compiler when it encounters a top-level form that looks like **(***function args***...)** and the symbol *function* has a **compiler:top-level-form** property. The value of the property must be a function of one argument. The compiler, rather than behaving in its normal fashion, calls the function with the original form as its argument. Whatever the function returns is dumped as the form to be evaluated at load time. You can have the function evaluate the form at compile time simply by calling **eval**. Note that the form returned by the function does *not* go back through the compiler's top-level form processing. This means that the returned form, which has been dumped to a compiled code file, cannot contain function definitions that you expect to be compiled.

Function Compiler

The function compiler takes a Lisp function and translates it from Lisp expressions into compiled functions. Compiled functions are represented in Lisp by compiled function objects, which contain machine code as well as various other information. The printed representation of the object is as follows:

```
#<DTP-COMPILED-FUNCTION name address>
```

When dealing with function bodies the function compiler performs the following operations on a form in this order:

1. Looks for compiler declarations (expands macros far enough to determine if they are declarations or not)
2. Performs style checking, unless you explicitly inhibit it.
3. Performs optimizations, if so requested, trying to optimize body forms from the inside out.
4. Runs transformations.
5. Expands macros.

If the case of a regular function, the entire process is repeated on the function's arguments. A special form, on the other hand, compiles its subforms, or not, depending on the syntax of the particular special form. When all the processing is done, the function compiler generates machine instructions.

Circular structures may not be constants in compiled functions.

bin File Dumper

The **bin** (binary) file dumper accepts a stream of Lisp forms and/or machine-instruction function definitions from the function compiler and writes them in a compact form into a compiled code file.

It is also possible to make a compiled code file containing data, rather than a compiled program. Call the **bin** file dumper by itself via the **sys:dump-forms-to-file** function. See the section "Putting Data in Compiled Code Files".

By loading the compiled code file (using the function **zl:load**, the Zmacs command Load File (`m-X`), or the Command Processor command Load File), the objects represented in the file are created in your Lisp world.

Compiler Tools and Their Differences

Tools for Compiling Code from the Editor Into Your World

You can use several Zmacs commands to compile code in an editor buffer to your world. Users generally compile routines to memory as soon as they write them, debugging them before proceeding with more complex routines. The most common command for incremental compiling is Compile Region (`m-X`), or `c-sh-C`.

c-sh-C Compile Region
Compile Region (m-X)

Compiles the region, or if no region is defined, the current definition.

Because recompiling routines as you edit them can be quite time-consuming, Zmacs provides two commands for compiling only those routines that have changed since they were last compiled: Compile Changed Definitions (m-X) and Compile Changed Definitions of Buffer (m-X), or m-sh-C. These commands obviate the need to remember which routines have changed in your buffer or buffers. Alternatively, you can recompile the entire buffer.

Compile Changed Definitions (m-X)

Compiles any definitions that have changed in any of the current buffers. With a numeric argument, it prompts individually about whether to compile particular changed definitions (the default compiles all changed definitions).

Compile Changed Definitions of Buffer (m-X)
m-sh-C

Compiles any definitions that have changed in the current buffer. With a numeric argument, it prompts individually about whether to compile particular changed definitions. The default is to compile all changed definitions.

Compile Buffer (m-X)

Compiles the entire buffer. With a numeric argument, it compiles from point to the end of the buffer. (This is useful for resuming compilation after a prior Compile Buffer has failed.)

Tools for Compiling Files

Compiling a source file, using the Zmacs command Compile File (m-X), the Command Processor command Compile File, or the function **compile-file**, saves the output in a binary file (called a compiled code file). You can compile a file and also load the resulting file by using **compile-file** with the **:load** keyword set to **t**, or you can load the file separately into your Lisp world by using **load** or Load File (m-X).

Compile File **Command**

Compile File *pathname keywords*

Compile the file(s) designated in *pathname.*

pathname	The pathname of the file to compile. The default is the usual file default.
keywords	:Binary File, :Compiler, :Load, :More Processing, :Output Destination, :Query, :Silently
:Binary File	{*pathname*} The file into which to put the output. The default is *pathname*.bin for a 3600-family machine, and *pathname*.ibin for an Ivory-based machine.
:Compiler	{Lisp, Pascal, Prolog, Fortran, Use-Canonical-Type} The compiler to use. The default is Use-Canonical-Type.
:Load	{Yes, No, Ask} Whether to load the file after compiling. The default is No. The mentioned default is Yes.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Silently	{Yes, No} Whether to display the pathname of the file being compiled. The default is Yes.
:Query	{Yes, No, Ask} Whether to ask for confirmation before compiling each file. The default is No. The mentioned default is Yes.

compile-file *input-file* &key *:output-file :package :load (:set-default-pathname* ***compile-file-set-default-pathname****)* *Function*

The file *input-file* is given to the compiler, and the output of the compiler is written to a file whose name is *input-file* with a canonical file type of **:bin** or **:ibin**, under Genera, ".b" under CLOE Runtime.

You can supply the **:output-file** or ".fas" keyword to specify where the output is written. The **:package** keyword indicates the package with respect to which the *input-file* is compiled. You can load the file after compiling it by supplying **:load t**.

The purpose of **compile-file** is to take a file and produce a translated version that does the same thing as the original except that the functions are compiled.

compile-file reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the compiled code file, which when loaded reproduces the effect of that source form.

Thus, if the source contains a **(defun ...)** form at top level, when the compiled code file is loaded, the function is defined as a compiled function. If, on the other hand, the source file contains a form that is not of a type known specially to the stream compiler, then that form (encoded in binary format) is output "directly" into the compiled code file, so that when that file is loaded that form is evaluated. For example, if the source file contains **(setq x 3)**, then the compiler places in the compiled code file instructions to set **x** to **3** at load time. (For a more general form, the compiled code file would contain instructions to recreate the list structure of a form and then call **eval** on it.)

compile-file returns the pathname of the **:output-file**, which you can pass to **:load** to load the compiled code file.

Compatibility Note: **:package**, **:load**, and (**:set-default-pathname *compile-file-set-default-pathname***) are Symbolics extensions to Common Lisp, not available in CLOE.

Compile File (m-X)

Compiles a file, offering to save it first (if it has an associated buffer that has been modified). It prompts for a file name in the minibuffer, using the file associated with the current buffer as the default. It does not load the file.

File Types of Lisp Source and Compiled Code Files

The results of compilation are written to a file of canonical type **:bin** or **:ibin**. The type **:bin** is for files compiled for 3600-family machines, and **:ibin** is for files compiled for Ivory-based machines. The actual file types for compiled code files are host-dependent, as are those of the Lisp source files. The following table shows the file types of both input and output files for various hosts.

Host type	*File type of source file*	*File type of compiled code file*
3600-family Symbolics	lisp	bin
Ivory-based Symbolics	lisp	ibin
Multics	lisp	bin
TOPS-20	LISP, LSP	BIN
UNIX	l, lisp	bn, bin
VAX/VMS	LSP	BIN

Tools for Compiling Single Functions

Compiled functions are Lisp objects that contain programs in the machine instruction set. Compiling an interpreted function by calling the function compiler on a function spec, converts it into a compiled function and changes the definition of the function spec to be that compiled function. Most users do not compile functions directly, but rather compile files or regions of code in a Zmacs buffer.

compile *function-spec* &optional *lambda-exp* *Function*

Gets the function definition from either of its arguments. If the lambda expression *lambda-exp* is supplied, **compile** uses **lambda-exp** and converts it into a compiled function object. If, on the other hand, *lambda-exp* is **nil**, **compile** gets the function definition of **function-spec**, which is either a function specification or **nil**. If **nil**, **compile** returns the compiled function object without storing it anywhere. If *function-spec* is not **nil**, **compile** changes *function-spec*'s definition to be the compiled function object; the returned value is *function-spec*.

Consider this example:

```
(setq foo (compile nil '(lambda (x) (* x (- x 1)))))

(funcall foo 8) => 56
```

Consider this example:

```
(fboundp 'compiler-test) nil
(compile 'compiler-test '(lambda (x) x))
(fboundp 'compiler-test) t
(compiler-test 259) 259
```

For more information, see the function **fdefine**.

uncompile *function-spec* *Function*

If *function-spec* is not defined as an interpreted function and it has a **:interpreted-definition** property in its debugging-info, **uncompile** restores the function cell from the value of the property. (Otherwise, **uncompile** does nothing and returns **"Not compiled"**.) This "undoes" the effect of **compile**.

See the function **undefun**.

Although all these methods call the compiler and produce compiled function objects, they are not equivalent. For example, using **compile-file** to compile a source file of canonical type **:lisp** converts it into a binary file, with a canonical file type of **:bin**. Compiling the source file has no effect on your Lisp environment. Compiling a top-level form in an editor buffer, using a command like Compile Region (c-sh-C) or Compile Buffer (m-X), creates a compiled function object in memory but does not write an object code file on disk. Compiling a top-level form in an editor buffer does cause some side effects on the Lisp environment.

The most essential difference, however, between compiling a source file and compiling the same code in an editor buffer is this: When you compile a file, most function specs are not defined and most forms (except those within **eval-when (compile)** forms) are not evaluated at compile time. Instead the compiler puts instructions into the binary file that causes evaluation to occur at load time.

Loading a compiled code file does not differ substantially from loading its associated source file, except that the functions defined in the binary file are defined as compiled functions instead of interpreted functions. When you load a source file that contains **defun** forms, you define the function specs named in the forms to be those functions.

Sometimes you might want to put things in the compiled code file that are not meant merely to be translated into binary form. Top-level macro definitions fall into this category. The macros must actually get defined within the compiler in order for the compiler to be able to expand them at compile time. Compiler declarations also fall into this category.

Compiler Warnings Database

Compiler warnings are kept in an internal database. Several functions, Command Processor commands, and Zmacs commands 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 can also be saved to a file or printed out as well as stored in the database. If the value of the special variable **compiler:suppress-compiler-warnings** is not **nil**, warnings are not printed, although they are still stored in the database.

Save Compiler Warnings Command

Save Compiler Warnings *pathname* *files-whose-warnings-to-save*

Save compiler warnings of the files *files-whose-warnings-to-save* to the specified *pathname*. *files-whose-warnings-to-save* can be All to save all warnings, or it can be a list of one or more pathnames. Among the pathnames can be the special token No File to catch warnings for no particular file.

The database has a printed representation. The command Show Compiler Warnings or the function **print-compiler-warnings** produces this printed representation from the database, and **compiler:load-compile-warnings** updates the database from a saved printed representation.

Show Compiler Warnings Command

Show Compiler Warnings *pathname keywords*

Display compiler warnings for the files specified by *pathnames*.

pathname	{*pathnames(s)*, All, No File} The compiled files whose warnings to show. All shows all compiler warnings for the compilation. No File shows the warnings for no particular file. The default is All.
keywords	:More Processing, :Output Destination

:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

print-compiler-warnings &optional *files* (*stream* **zl:standard-output**) *file-node-message function-node-message anonymous-function-node-message* *Function*

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*, which you can use to send the results to a file.

compiler:load-compiler-warnings *file* &optional (*flush-old-warnings* **t**) *Function*

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 by invoking **compile-system** with the **:batch** option. The default type for such files is CWARNS; for example, a file might be named FOO.CWARNS.

Several Zmacs commands manipulate the compiler warnings database.

Compiler Warnings (m-X)

Creates the compiler warnings buffer (called *Compiler-Warnings-1*) if it does not exist, puts all outstanding compiler warnings in that buffer, and switches to that buffer. You can view the compiler warnings by scrolling around and doing text searches through them using Edit Compiler Warnings (m-X).

Edit Compiler Warnings (m-X)

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 uses of c-. 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 previous configuration.

Edit File Warnings (m-X)

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.

Load Compiler Warnings (m-X)

Loads a file containing compiler warning messages into the warnings database. It prompts for the name of a file that contains the printed representation of compiler warnings. It always replaces any warnings already in the database.

Controlling Compiler Warnings

compiler:*compiler-warnings-to-core-action* *Variable*

Determines what the compiler does if there are compiler warnings when you compile functions in core. If the function already exists in core (that is, if you are redefining the function), then this variable is examined to determine the action to take. Possible values are

:define	Defines the function in core despite the warnings. This is the default.
:do-not-define	Does not define the function in core if there are warnings.
:query	Asks you what to do.

:warn Warns you that the redefinition is occuring.

si:*duplicate-declarations-warnings* *Variable*

Controls issuing a warning if duplicate declarations are found for a symbol. Its default is **t**, so warnings are issued. In some cases, particularly where code is generated automatically, you might not want to see these warnings. In those cases, you can set **si:*duplicate-declarations-warnings*** to **nil**.

compiler:invisible-references *variables* &body *body* *Special Form*

Allows references to *variables* in a form to remain invisible to the compiler for the purposes of variable reference warnings.

For example, suppose you have a macro **my-dotimes**, which is defined like this:

```
(defmacro my-dotimes ((var countform &optional resultform) &body forms)
  (let ((countvar (gensym)))
    `(prog ((,var 0)
            (,countvar ,countform))
        loop
           (unless (< ,var ,countvar)
             (return ,resultform))
           ,@forms
             (incf ,var)
           (go loop))))
```

If you use this in a function like this:

```
(defun silly-function ()
  (my-dotimes (i 5)
    (print 3)))
```

You would like to get a compiler warning to the effect that you have not used the variable **i** in the body of your function. However, since there are uses of the variable in the macro expansion, the compiler does not warn you. If you use **compiler:invisible-references**, like this:

```
(defmacro my-dotimes ((var countform &optional resultform) &body forms)
  (let ((countvar (gensym)))
    `(prog ((,var 0)
            (,countvar ,countform))
        loop
           (unless (compiler:invisible-references (,var)
                     (< ,var ,countvar))
             (return ,resultform))
           ,@forms
           (compiler:invisible-references (,var)
             (incf ,var))
           (go loop))))
```

the compiler warns you when you compile **silly-function** that the variable **i** was never used.

Compiler Style Warnings

The compiler performs style checking on all forms. This means that the Lisp compiler produces compiler warnings when it sees programs that are invalid Lisp or that may produce errors at runtime. You can add to the checks that the compiler makes in several ways.

- Your macros can can call the **warn** function to warn of problematic usage.
- You can use **compiler:make-obsolete** to declare something obsolete.
- You can define *style checkers* by means of the function-spec **compiler:style-checker**. A style checker is a Lisp function associated with a symbol. When the compiler compiles an s-expression with that symbol in the functional position **car**, it calls all of the style checkers for the symbol with an argument of the form. These style checkers can examine the form and call **warn** if they detect something wrong.

compiler:style-checker

Defines a style checker. Note: **compiler:style-checker** is not a function but rather, a function-spec. A style checker is a Lisp function associated with a symbol. When the compiler compiles an S-expression with that symbol in the functional position **car**, it calls all of the style checkers for the symbol with an argument of the *form*. These style checkers can examine the *form* and call **warn** if they detect something wrong. *checker-name* is the name of your style checker function, and *symbol* is the symbol that you want to check. *arg1* and *arg2* are optional arguments to your style checker function.

You define a style checker as follows:

```
(defun (compiler:style-checker style-checker-name
                               function-symbol)
       (form)
   body-that-looks-at-the-form...
   )
```

You can have multiple style checkers on a single function symbol. For example, assume that you define function to take a first argument that must be a number, and which is often a constant.

```
(defun stylish-function (number &rest other-args)
    )
```

You might write:

```
(defun (compiler:style-checker first-arg-must-be-numeric
                               stylish-function)
       (form)
  (destructuring-bind (ignore number &rest ignore) form
    (when (and (compiler:constant-form-p number)
               (not (numberp
                      (compiler:constant-evaluator number))))
      (warn "The first argument ~S to ~S is not a number."
            number 'stylish-function))))
```

In the example, the function **compiler:constant-form-p** simply checks if the form is treated as a constant by the compiler; the function **compiler:constant-evaluator** returns the value of a constant. You have to be very careful about how you examine arguments. The **form** in the example code is uncompiled list structure. If the caller is passing a variable as an argument

```
(stylish-function foo)
```

then the form will contain the symbol **foo** as the second element. **foo** is not a constant, so you cannot tell what its runtime value is at compile time.

The pre-Genera 7.0 way of style checking using property lists is also supported, but you cannot use both the new and the old technology on the same checked function. In the old way, style checking is implemented by the **compiler:style-checker** property on a symbol; the value of the property is called on all forms whose **car** is that symbol, except those immediately enclosed in **inhibit-style-warnings**. Obsolete function warnings are also performed by means of the style-checking mechanism.

inhibit-style-warnings *form* *Function*

Prevents the compiler from performing style-checking on the top level of *form*; style-checking will still be done on the arguments of *form*.

The following code warns you about the obsolete function **zl:explode**, since **inhibit-style-warnings** applies only to the top level of the form inside it, in this case, to the **setq**.
Generate warning:

```
(inhibit-style-warnings (setq bar (explode foo)))
```

The following code, on the other hand, does *not* warn that **explode** is an obsolete function:
Do not generate warning:

```
(setq bar (inhibit-style-warnings (explode foo)))
```

If an optimizer needs to return a form with nested "bad-style" forms, there should be an explicit **inhibit-style-warnings** wrapped around the nested forms.

By setting the compile-time value of **inhibit-style-warning-switch** you can enable or disable some of the warning messages of the compiler. The compile-time value of **obsolete-function-warning-switch** enables or disables obsolete-function warnings in particular.

compiler:make-obsolete *spec reason* &optional *(type* **'defun***)* *Special Form*

Declares a function, flavor, or structure to be obsolete; code that calls an obsolete definition generates a compiler warning. It is useful for marking as obsolete some Zetalisp functions that exist in Common Lisp but should not be used in new programs, or for reminding users that some function is being phased out.

spec is the definition to be made obsolete and is not evaluated. *reason* is evaluated and is the warning or explanation to be printed when the obsolete definition is called. *type*, the optional third argument, is the definition-type of the object declared obsolete and is not evaluated. Its default value is **defun** when no type is specified. **compiler:make-obsolete** recognizes four definition-types: **defun**, **defflavor**, **zl:defstruct**, and **defvar**.

compiler:make-obsolete with a third argument of **zl:defstruct** makes the structure obsolete as well as all of its accessor functions. **compiler:make-obsolete** with a third argument of **defflavor** makes obsolete both the flavor and its outside accessible instance variables.

An attempt to create a new flavor with an obsolete flavor as an included or component flavor generates a compiler warning. Likewise, creating a new structure with an obsolete structure as an included structure also generates a warning.

compiler:make-message-obsolete *message-name format-string* *Function*

Allows you to generate compiler warnings about obsolete message names. The first argument, *message-name*, is the obsolete message name. The second argument, *format-string*, is the warning to be printed. If the string contains the *~S* format directive, it will be replaced by the object that was sent the message.

Example:

```
(compiler:make-message-obsolete :clear-screen
        "You have sent the message :CLEAR-SCREEN to the object ~S.
        This name is obsolete.  The new name for this message is
        :CLEAR-WINDOW.  Please update your code.")
```

Function-referenced-but-never-defined Warnings

Normally, the compiler notices whenever any function *x* calls any other function *y*; it takes note of all these uses, and then warns you at the end of the compilation if function *y* was called but was neither defined nor declared (by **compiler:function-defined**).

The compiler uses a set of variables and functions to keep track of which functions have been defined and which have been referenced. These are the basis for the messages "FOO was defined but never referenced" that occur during compiling.

sys:file-local-declarations *Variable*

Stores global declarations valid for the entire compilation. Since it can become fairly large, it is implemented as a hash table (or **nil**). The symbol being declared is the key, and the value is a property list of declarations and values. The default value is **nil**.

compiler:functions-defined *Variable*

A hash table of all functions defined or **nil**, if none has been defined yet.

compiler:functions-referenced *Variable*

A hash table of functions referenced but not defined. Each entry is an alist of (<generic-pathname> . <by-whom>). In this way warnings can be put into the appropriate file when this variable is processed at the end of a compilation.

compiler:function-defined *fspec* *Function*

Tells the compiler that the function *fspec* has been defined (by putting it into the hash table in **compiler:functions-defined**).

zl:*expr, **zl:*lexpr**, and **zl:*fexpr** are the Maclisp equivalents of **compiler:function-defined**.

zl:*expr &rest *functions* *Function*

Declares each function spec in the list of *functions* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, which appears at the end of the compilation. Provided for Maclisp compatibility.

zl:*lexpr &rest *functions* *Function*

Declares each function spec in the list of *functions* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined that is printed at the end of the compilation. Provided for Maclisp compatibility.

zl:*fexpr &rest *functions* *Function*

Declares each function spec in the list of *functions* to be the name of a special form. In addition it prevents these names from appearing in the list of functions referenced but not defined that is printed at the end of the compilation. Provided for Maclisp compatibility.

compiler:file-declare *thing declaration value* *Function*

Enters a declaration in the table **sys:file-local-declarations** for the remaining extent of the compilation environment.

```
(compiler:file-declare 'foo 'special t)
```

compiler:file-declaration *thing declaration* *Function*

Looks up a declaration in the table **sys:file-local-declarations**. It returns the declaration when *thing* is a declaration of type *declaration* and **nil** otherwise.

compiler:function-referenced *what* &optional (*by* **compiler:default-warning-function**) *Function*

Useful for requesting compiler warnings in certain esoteric cases. For example, sometimes the compiler has no way of telling that a certain function is being used. Suppose that instead of *x*'s containing any forms that call *y*, *x* simply stores *y* away in a data structure somewhere, and someplace else in the program that data structure is accessed and **funcall** is done on it. In this case the compiler cannot see that this is going to happen; the result is that it cannot note the function usage and hence cannot create a warning message. In order to make such warnings happen, you can explicitly call the function **compiler:function-referenced** at compile-time.

what is a symbol that is being used as a function. *by* can be any function spec. **compiler:function-referenced** must be called at compile time while a compilation is in progress. It tells the compiler that the function *what* is referenced by *by*. When the compilation is finished, if the function *what* has not been defined, the compiler issues a warning to the effect that *by* referred to the function *what*, which was never defined.

Overriding Variable-defined-but-never-referenced Warnings

Sometimes functions take arguments that they deliberately do not use. Normally the compiler warns you if your program binds a variable that it never references. In order to disable this warning for variables that you know you are not going to use, you can do one of several things.

- You can **declare** the variable to be ignored:

  ```
  (declare (ignore fraz-size))
  ```

- You can name the variables **ignore** or **ignored**. The compiler does not complain if a variable of one of these names is not used. Furthermore, you can have more than one variable in a lambda-list that has one of these names.

- You can simply use the variable for effect (ignoring its value) at the front of the function. This has the advantage that **arglist** will return a more meaningful argument list for the function, rather than returning something with **ignore**s in it. Example:

```
(defun the-function (list fraz-name fraz-size)
  fraz-size       ; This argument is not used.
  ...)
```

- You can use the variable as an argument to the **ignore** function.

```
(defun the-function (list fraz-name fraz-size)
  (ignore fraz-size)
  ...)
```

Compiler Switches

The compile-time values of the following variables, called "compiler switches", affect the operation of the compiler. Use **compiler-let** to bind compiler switches.

compiler:obsolete-function-warning-switch *Variable*

The compile-time value of this variable affects the operation of the compiler. If this variable is non-**nil**, the compiler tries to warn you whenever an obsolete function, such as **zl:maknam** or **zl:samepnamep**, is used. The default value is **t**.

compiler:open-code-map-switch *Variable*

The compile-time value of this variable affects the operation of the compiler. If this variable is non-**nil**, the compiler attempts to produce inline code for the mapping functions (**mapc**, **mapcar**, and so on, but not **zl:mapatoms**) if the function being mapped is an anonymous lambda-expression. Setting this switch to **nil** makes the compiled code smaller. Setting this switch to **t** makes the compiled code larger but faster. The default value is **t**.

zl:all-special-switch *Variable*

The compile-time value of this variable affects the operation of the compiler. If this variable is non-**nil**, the compiler regards all variables as special, regardless of how they were declared. The default is **nil**.

compiler:inhibit-style-warnings-switch *Variable*

The compile-time value of this variable affects the operation of the compiler. If this variable is non-**nil**, all compiler style-checking is turned off. Style checking is used to issue obsolete function warnings and other sorts of warnings. The default value is **nil**.

compiler:*inhibit-keyword-argument-warnings* *Variable*

Controls whether the compiler checks the keyword arguments of a function against the keyword arguments accepted by the called function. The values can be:

nil	This is the default. The compiler checks keyword arguments supplied in a function call against the keyword arguments accepted by the called function. As with checking the number of arguments in a function call, this checking does not work if the function call is earlier in the file or group of files than the definition of the called function. If there is an **arglist** declaration, it is used in place of the actual lambda-list to determine what keywords are accepted, since often the declared lambda-list contains **&key** but the actual lambda-list contains just **&rest**.
t	Disables this checking. This can be useful if you have a lot of declared arglists that are malformed.

compiler:compiler-verbose *Variable*

The compile-time value of this variable affects the operation of the compiler. The compiler displays a message (using **zl:standard-output**) each time it starts compiling a function when the value of **compiler:compiler-verbose** is **t**. The default value is **nil**.

compiler:*enable-frame-splitting* *Variable*

This variable controls the compiler's action when it encounters a function which is too large to be directly supported by the machine architecture. The possible values are:

nil	Attempt to compile the function anyway. For borderline functions, this may work, and if it does will result in better code being generated.
t	Heuristically split the function into smaller functions which can be supported by the architecture. The resulting code may not be as efficient.
:warn	Same as **t**, but when the compiler splits a function, issue a warning. This can be useful to identify functions which may not compile as efficiently as you would want.

To use this variable, either set it globally, or wrap defining forms in a **compiler-let**.

Example:

```
(compiler-let ((compiler:*enable-frame-splitting* :warn))
  (defun large-function ...)
  ) ;;End of COMPILER-LET
```

si:*compiled-function-constant-mode* *Variable*

Controls how constants are localized with compiled functions in normal compiled-function creation. Its value can be one of the following:

:share	This is the default. Compiled function constants are copied and shared to be immediately after the compiled function in memory.
:copy	Compiled function constants are copied to be immediately following the compiled function in memory. No attempt is made to share constants. In some cases this may result in faster loading of compiled functions and a larger working set for the resulting functions.
:unlocalized	Compiled function constants are not copied. Thus circular structures and **eq**-ness of constants are preserved. However, the working set of running functions loaded in this manner is guaranteed to be larger, since the constants are guaranteed to be on separate pages. Additionally, garbage collection overhead wil be higher for dynamic constants, and IDS files may be larger.

Note that the only constants which are currently copied are lists, numbers, strings, and simple arrays.

Copying of compiled functions and this variable may be changed in a future release.

Compiler Source-Level Optimizers

An *optimizer* is a function that converts a form into another form that is more efficiently executed. An optimizer can be used to transform code into an equivalent but more efficient form that can be compiled better. For example, **(eq** *obj* **nil)** is transformed into **(null** *obj***)**, which can be compiled better.

Do not use optimizers to define new language features, because they take effect only in the compiler; the interpreter (that is, the evaluator) does not know about optimizers. So an optimizer should not change the effect of a form; it should produce another form that does the same thing, possibly faster or with less memory. If you want to actually change the form to do something else, you should use macros.

The compiler treats (optimized or transformed) forms returned by compiler optimizers as if they were wrapped in an **inhibit-style-warnings** form. For example, the expression:

```
    (eql x 3)
```

is optimized into the expression:

```
    (eq x 3)
```

In general, it is a bad idea to compare numbers with **eq**, since the implementation of numbers is such that some numbers can be compared with **eq** and some can't. A style checker keeps the user from writing **(eq x 3)**. The optimizer is allowed to do this without warning on the assumption that the optimizer always generates "correct" code.

Note: **inhibit-style-warnings** only affects the top-level form inside it. If an optimizer needs to return a form with nested "bad-style" forms, there should be an explicit **inhibit-style-warnings** wrapped around the nested forms.

compiler:add-optimizer *target-function optimizer-name* &rest *optimized-into* *Function*

Puts *optimizer-name* on *target-function*'s optimizers list if it is not there already. *optimizer-name* is the name of an optimization function, and *target-function* is the name of the function calls that are to be processed. Neither is evaluated.

(**compiler:add-optimizer** *target-function optimizer-name optimize-into-1 optimize-into-2...*) also remembers *optimize-into-1*, and so on, as names of functions that can be called in place of *target-function* as a result of the optimization.

Files That Maclisp Must Compile

Certain programs are intended to be run both in Maclisp and in Symbolics Common Lisp. Their source files need some special conventions. For example, all **special** declarations must be enclosed in top-level **declare** forms, so that the Maclisp compiler sees them. The main issue is that many Symbolics Common Lisp functions and special forms do not exist in Maclisp.

The **#Q** sharp-sign reader macro causes the object that follows it to be visible only when compiling for Symbolics Common Lisp. The sharp-sign reader macro **#M** causes the following object to be visible only when compiling for Maclisp. These work both on subexpressions of the objects in the file, and at top level in the file. To conditionalize top-level objects, however, it is better to put the macros **zl:if-for-lispm** and **zl:if-for-maclisp** around them. (You can only put these around a single object.) The **#Q** sharp-sign reader macro cannot do this, since it can be used to conditionalize any Lisp object, not just a top-level form.

To allow a file to detect what environment it is being compiled in, the following macros are provided:

zl:if-for-lispm &rest *forms* *Function*

Seen at the top level of the compiler, *forms* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Symbolics Common

Lisp. If the Symbolics Common Lisp interpreter sees this it evaluates *forms* (the macro expands into *forms*).

zl:if-for-maclisp &rest *forms* *Function*

Seen at the top level of the compiler, *forms* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Maclisp (for example, if the compiler is COMPLR). If the Symbolics Common Lisp interpreter sees this it ignores it (the macro expands into **nil**).

zl:if-for-maclisp-else-lispm *maclisp-form lispm-form* *Function*

When **(if-for-maclisp-else-lispm** *form1 form2***)** is seen at the top level of the compiler, *form1* is passed to the compiler top level if the output of the compiler is a compiled code file intended for Maclisp; otherwise *form2* is passed to the compiler top level.

zl:if-in-lispm &rest *forms* *Function*

In Symbolics Common Lisp, **(if-in-lispm** *forms***)** causes *forms* to be evaluated; in Maclisp, *forms* is ignored.

zl:if-in-maclisp &rest *forms* *Function*

In Maclisp, **(if-in-maclisp** *forms***)** causes *forms* to be evaluated; in Symbolics Common Lisp, *forms* is ignored.

When you have two definitions of one function, one conditionalized for one machine and one for the other, put them next to each other in the source file with the second **(defun)** indented by one space, and the editor will put both function definitions on the screen when you ask to edit that function.

In order to make sure that those macros are defined when reading the file into the Maclisp compiler, you must make sure the file starts with a prelude, which should look like:

```
(declare (cond ((not (status feature lispm))
                (load '|AI: LISPM2; CONDIT|))))
```

This does nothing when you compile the program on Symbolics computers. If you compile it with the Maclisp compiler, it loads definitions of the above macros, so that they will be available to your program. The form **(status feature lispm)** is generally useful in other ways; it evaluates to **t** when evaluated on Symbolics computers and to **nil** when evaluated in Maclisp.

Putting Data in Compiled Code Files

A compiled code file can contain data rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in a printed representation of that same data structure. Also, certain data structures, such as arrays, do not have a convenient printed representation as text, but can be saved in compiled code files.

In compiled programs, the constants are saved in the compiled code file in this way. The compiler optimizes by making constants that are **equal** become **eq** when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a compiled code file is loaded, objects that were **eq** when the file was written are still **eq**; this does not normally happen with text files.

The following types of objects can be represented in compiled code files:

- Symbols
- Numbers of all kinds
- Lists
- Characters
- Arrays of all kinds (including strings)
- Instances, such as hash tables
- Instance of user-defined CLOS classes
- Compiled function objects

When an instance is put (dumped) into a compiled code file, it is sent a **:fasd-form** message, which must return a Lisp form that, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply dumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain **eq**; the **:fasd-form** message is sent only the first time a particular instance is encountered during writing of a compiled code file. If the instance does not accept the **:fasd-form** message, it cannot be dumped.

If you need to dump CLOS instances, see the generic function **clos:make-load-form**.

sys:dump-forms-to-file *filename forms* &optional *file-attribute-list* *Function*

Writes data to a file in binary form by invoking the **bin** file dumper. *forms* is a list of Lisp forms, each of which is dumped in sequence. It dumps the forms, not their results. The forms are evaluated when you load the file. For more information, see the section "Putting Data in Compiled Code Files". (If you need to dump CLOS instances, see the generic function **clos:make-load-form**.)

For example, suppose **a** is a variable bound to any Lisp object, such as a list or array. The following example creates a compiled code file that recreates the variable **a** with the same value:

```
(sys:dump-forms-to-file "f:>foo>aval"
        (list '(setq a ',a)))
```

For the purposes of understanding what this function does, you can consider that it is the same as the following:

```
(defun sys:dump-forms-to-file (file forms)
   (with-open-file (s file ':direction ':output)
      (dolist (f forms)
        (print f s))))
```

The actual definition (which is more complicated) writes a binary file in a more easily parsed format so it will load faster. It can also dump arrays, which you cannot write to a Lisp source file.

file-attribute-list supplies an optional attribute list for the resulting compiled code file. It has basically the same result when loading the binary file as the file attribute list does for **compile-file**. Its most important application is for controlling the package that the file is loaded into.

```
(sys:dump-forms-to-file "foo" forms-list '(:package "user"))
```

sys:dump-forms-to-file always puts a package attribute into the binary file it writes. If you do not specify the *file-attribute-list* argument, or if *file-attribute-list* does not contain a **:package** attribute, the function uses the **cl-user** or **zl-user** package, depending on the context. This is to ensure that package prefixes on symbols are always interpreted when they are loaded as they were intended when the file was dumped.

The *file-attribute-list* argument can be used to store useful information (such as "headers" for special data structures) in the file's attribute list. The information can then be retrieved from the attribute list with **fs:pathname-attribute-list**, without reading the rest of the file.

sys:dump-forms-to-file checks the variable **si:*bin-dump-no-list-sharing***:

si:*bin-dump-no-list-sharing*
Controls whether or not **sys:dump-forms-to-file** attempts to determine what data is shared and preserve that sharing.

Files dumped with **sys:dump-forms-to-file** that contain only data (no code), can be loaded into both 3600-family and Ivory machines.

clos:make-load-form *object* *Generic Function*

Provides a way to use an instance of a user-defined CLOS class (that is, an instance whose metaclass is **clos:standard-class** or **clos:structure-class**) as a constant in a program compiled with **compile-file**. Users can define a method for **clos:make-load-form** that describes how an equivalent object can be reconstructed when the compiled-code file is loaded.

compile-file calls **clos:make-load-form** on an object needed at load time, if the object's metaclass is **clos:standard-class**. **compile-file** will call **clos:make-load-form** only once for any given object (compared with **eq**) within a single file. If **clos:make-load-form** is called and no user-defined method is applicable, an error is signaled.

The argument *object* is an object needed at load-time.

clos:make-load-form returns two values. The first value, called the "creation form", is a form that, when evaluated at load time, should return an object that is equivalent to *object*.

The second value, called the "initialization form", is a form that, when evaluated at load time, should perform further initialization of the object. The value returned by the initialization form is ignored. If the **clos:make-load-form** method returns only one value, the initialization form is **nil**, which has no effect. If the object used as the argument to **clos:make-load-form** appears as a constant in the initialization form, at load time it will be replaced by the equivalent object constructed by the creation form; this is how the further initialization gains access to the object.

Both the creation form and the initialization form can contain references to instances of user-defined CLOS classes. However, there must not be any circular dependencies in creation forms. An example of a circular dependency is when the creation form for the object X contains a reference to the object Y, and the creation form for the object Y contains a reference to the object X. A simpler example would be when the creation form for the object X contains a reference to X itself. Initialization forms are not subject to any restriction against circular dependencies, which is the entire reason that initialization forms exist. See the example of circular data structures below.

The creation form for an object is always evaluated before the initialization form for that object. When either the creation form or the initialization form references other objects of user-defined types that have not been referenced earlier in the **compile-file**, the compiler collects all of the creation and initialization forms. Each initialization form is evaluated as soon as possible after its creation form, as determined by data flow. If the initialization form for an object does not reference any other objects of user-defined types that have not been referenced earlier in the **compile-file**, the initialization form is evaluated immediately after the creation form. If a creation or initialization form F references other objects of user-defined types that have not been referenced earlier in the **compile-file**, the creation forms for those other objects are evaluated before F, and the initialization forms for those other objects are also evaluated before F whenever they do not depend on the object created or initialized by F. Where the above rules do not uniquely determine an order of evaluation, which of the possible orders of evaluation is chosen is unspecified.

While these creation and initialization forms are being evaluated, the objects are possibly in an uninitialized state, analogous to the state of an object between the time it has been created and it has been processed fully by **clos:initialize-instance**. Programmers writing methods for **clos:make-load-form** must take care in manipulating objects not to depend on slots that have not yet been initialized.

Examples:

```
;; Example 1
(defclass my-class ()
   ((a :initarg :a :reader my-a)
    (b :initarg :b :reader my-b)
    (c :accessor my-c)))

(defmethod shared-initialize ((self my-class) ignore &rest ignore)
  (unless (slot-boundp self 'c)
    (setf (my-c self) (some-computation (my-a self) (my-b self)))))

(defmethod make-load-form ((self my-class))
  `(make-instance ',(class-name (class-of self))
                  :a ',(my-a self) :b ',(my-b self)))
```

In this example, an equivalent instance of **my-class** is reconstructed by using the values of two of its slots. The value of the third slot is derived from those two values.

Another way to write the last form in the above example is to use **clos:make-load-form-saving-slots**:

```
(defmethod make-load-form ((self my-class))
   (make-load-form-saving-slots self '(a b)))
```

```
;; Example 2
(defclass my-frob ()
   ((name :initarg :name :reader my-name)))
(defmethod make-load-form ((self my-frob))
  `(find-my-frob ',(my-name self) :if-does-not-exist :create))
```

In this example, instances of **my-frob** are "interned" in some way. An equivalent instance is reconstructed by using the value of the name slot as a key for searching existing objects. In this case the programmer has chosen to create a new object if no existing object is found; an alternative would be to signal an error in that case.

```
;; Example 3
(defclass tree-with-parent ()
   ((parent :accessor tree-parent)
    (children :initarg :children)))
(defmethod make-load-form ((x tree-with-parent))
  (values
    ;; creation form
    `(make-instance ',(class-of x)
                    :children ',(slot-value x 'children))
    ;; initialization form
    `(setf (tree-parent ',x) ',(slot-value x 'parent))))
```

In this example, the data structure to be dumped is circular, because each parent has a list of its children and each child has a reference back to its parent. Suppose **clos:make-load-form** is called on one object in such a structure. The creation form creates an equivalent object and fills in the children slot, which forces creation of equivalent objects for all of its children, grandchildren, and so on. At this point none of the parent slots have been filled in. The initialization form fills in the parent slot, which forces creation of an equivalent object for the parent if it was not already created. Thus the entire tree is recreated at load time. At compile time, **clos:make-load-form** is called once for each object in the tree. All the creation forms are evaluated, in unspecified order, and then all of the initialization forms are evaluated, also in unspecified order.

clos:make-load-form-saving-slots *object* &optional *save-slots* *Function*

Used in the bodies of methods for **clos:make-load-form**. The argument *object* is an object needed at load-time. The argument *save-slots* is a list of the names of the slots to preserve; it defaults to all of the local slots.

clos:make-load-form-saving-slots returns forms that construct an equivalent object using **clos:make-instance** and **setf** of **clos:slot-value** for slots with values, or **clos:slot-makunbound** for slots without values, or other functions of equivalent effect.

clos:make-load-form-saving-slots returns two values, thus it can deal with circular structures. **clos:make-load-form-saving-slots** works for instances of user-defined classes; that is, instances whose metaclass is **clos:standard-class** or **clos:structure-class.**

See the generic function **clos:make-load-form**.

si:*bin-dump-no-list-sharing* *Variable*

Controls whether or not **sys:dump-forms-to-file** attempts to determine what data is shared and preserve that sharing. The default is **nil**, meaning that shared structure is preserved. If you set **si:*bin-dump-no-list-sharing*** to **t**, **sys:dump-forms-to-file** dumps faster but does not share sublists. This means that the file created may be larger than if sharing were preserved, and when you reload the file, sharing optimizations that were present in your original environment are not there.

Metering a Program's Performance

Metering Interface

Metering is the process of measuring the performance of a program, usually with the goal of determining where performance can be improved. Genera offers tools for metering different aspects of performance, such as time, paging, and consing. The Metering Interface is a uniform interface which makes it convenient to use the various metering tools.

Note that you can use the Metering Interface to meter "foreign" programs (such as C, FORTRAN, or Pascal programs) in the same way you use it to meter Lisp programs.

This chapter also documents several macros that are useful for metering short forms. See the section "Macros for Metering the Execution Time of Forms".

Overview of the Metering Interface

This section gives an overview of how to use the Metering Interface.

The Metering system is not part of the default world; it is loaded separately. To begin, load the system:

```
Load System Metering
```

Now you can select the Metering Interface:

```
SELECT %
```

The name % was chosen because it is related to metering; you might be seeking the percentage of time spent in one function, or some other percentage.

You will notice that when you first select the Metering Interface, it takes a little while for the Metering Interface to do some preparatory work. The progress note says "Computing Fudge Factors". We suggest that you wait for this to finish before typing, using the mouse, or doing any other activity that would interfere with this computation. See the section "Computing Fudge Factors".

Metering involves a sequence of steps. Here we briefly describe each step, and refer to the section that describes the step in further detail.

1. Specifying what to meter

   You can meter the performance of a Lisp form, a portion of a function, or one or more functions running within a process. For example, to execute and meter a form, click on [Meter Form] and enter the form. You could also meter one or more functions running within a process by clicking on [Meter in Process]. See the section "Specifying What to Meter".

2. Choosing the type of metering

   The Metering Interface prompts you for a metering type. The metering type controls how the data is collected and presented. The choices are: Function Call, Page Fault, Call Tree, Statistical Function Call, Statistical Call Tree, and Statistical Program Counter. See the section "Choosing a Metering Type".

3. Specifying metering parameters

   This is an optional step. The keyword options to Meter Form and Meter in Process allow you to control various aspects of the metering run. For example, you might decide to meter the code within a **without-interrupts** form, or to run the code a number of times and meter only the results of the last time it executes. When doing a Page Fault metering run, you can flush all pages first. The keywords available depend on which metering command you give, and the type of metering. See the section "Meter Form Command". See the section "Meter in Process Command".

4. Running and metering the code

   When using Meter Form, once you choose the type of metering and press RETURN, the form is immediately executed and simultaneously metered.

   When using Meter in Process you can meter one or more functions whenever they are naturally executed within a given process, instead of executing them immediately. This is useful for metering a function that normally runs within a process such as Zmail or a network process; the metering results are more representative of the usual environment of the function than they would be if you called the function explicitly.

   When you use Meter Form or Meter in Process, the result is called a *metering run*, which contains the data collected. The Metering Interface saves a history of metering runs in the top pane, which makes it convenient to show the data of a metering run later, or to repeat the metering run. The "current" metering run is the run whose results are now being displayed. The current metering run appears in bold face in the Metering History. See the section "Running and Metering the Code".

5. Customizing the display of metering results

   This is an optional step. The display in the bottom pane shows the results of the metering run. The results appear in columns under headers that describe the data. Each column is an *output field*. An output field shows a kind of data, such as consing, page faults, or time spent in a function. Output fields are divided into subfields; each subfield shows one aspect of the information.

   You can click Middle on a column header for information describing the data in that output field. You can also tailor this display to request more or less information by removing output fields from the display or adding them to the display. See the section "Customizing the Display of Metering Results".

   Often the data displayed is only a summary of the data available. You can get expanded information on a particular portion of the data by clicking Middle on it. See the section "Expanding Metering Data".

6. Interpreting the results of the metering run

   In this step you analyze the metering results and try to identify where the performance of your program can be improved. See the section "Interpreting the Results of a Metering Run".

7. Saving the results of a metering run (in hardcopy or a buffer)

   The results of a metering run are saved in your Lisp world until you explicitly delete the metering run or cold boot. However, sometimes it is useful to save the results more permanently, either by printing them or by sending the results to an editor buffer and then using a Zmacs command to write the results to a file. To do this, use Show Metering Run and give the :Output Destination option. See the section "Show Metering Run Command".

Usually when you are metering a program, you go through the cycle of metering steps several times. You might choose other metering types to collect information on different aspects of performance. You might modify the program on the basis of the metering results, and then meter the program again.

When you begin using the Metering Interface, you might make use of the Metering Help facilities: See the section "Getting Help in the Metering Interface". The Metering Interface enables you to use the mouse to give many of the metering commands: See the section "Using the Mouse in the Metering Interface".

When you do a metering run of a new metering type, it takes some time for the Metering Interface to do the necessary compilation and set-up work. However, future metering runs of the same metering type will not have this start-up delay.

How to Use the Metering Interface

Getting Help in the Metering Interface

Here are some suggestions for getting help within the Metering Interface. Most of these suggestions are applicable in most other contexts of the Symbolics Genera environment.

- What metering commands are available? Press the HELP key for a list.
- What does a metering command do? Enter Help *command-name* to see the documentation for a given metering command. You can also click Middle on any of the commands visible in the command menu.
- What operations can you do on a metering run? Click Right on a metering run displayed in the History of Metering Runs pane for a menu of operations.
- What operations can you do on an output field? Click Right on an output field displayed above the Metering Results pane for a menu of operations.
- What operations can you do on a node of a call tree display? Click Right on a node displayed in the Metering Results pane for a menu of operations.
- What is the information presented under a output field or subfield? Click Middle on an output field or subfield to describe its contents.

Specifying What to Meter

The first decision is whether you want to meter within the scope of a form or within a process.

When you use Meter Form, the form is executed in the Metering Interface process, and simultaneously metered. Metering will occur only within that form. You can meter everything occurring within that form, or specify functions or portions of functions to meter within that form. See the section "Meter Form Command".

Sometimes you want to meter a function whenever it is normally called within a process. You don't want to use Meter Form, because that would execute and meter the function immediately. Meter in Process allows you to meter one or more functions within a process, without explicitly calling those functions. You can meter everything occurring within the process, or specify functions or portions of functions to meter within the process. See the section "Meter in Process Command".

For both Meter Form and Meter in Process, you will be prompted for "What to meter", which allows you to further specify the code to be metered. The choices are:

Everything	Meter everything within the form/process.
Only When Enabled	Meter only the code which is surrounded by a **mi:with-metering-enabled** form.
Within Functions	Meter only within the functions specified. You will be prompted for :Metered functions, and you should enter the functions of interest.

Only When Enabled is used to meter only a portion of code. First you edit one or more functions of interest to wrap **mi:with-metering-enabled** around the desired portion or portions of code and compile the changed function (or functions). Then you can use Meter Form or Meter in Process and specify Only When Enabled. This starts a metering run that will meter only the code in the dynamic scope of **mi:with-metering-enabled**. See the section "Controlling Metering Within Lisp Code".

Choosing a Metering Type

This section describes each metering type, and then gives some general guidelines and suggestions about choosing the right metering type for different purposes.

Function Call	Collects data on every function entry and exit. The display indexes the data by function. This display shows the number of times each function was called, the total amount of time spent in each function, the total amount of consing that took place in each function, and information on any page faults that occurred in each function.
Call Tree	Collects the same data as Function Call, but the display indexes the data by the stack trace. This describes the entire calling sequence of functions that occurred. Each function is a node of

	the tree; the callees of a function are displayed below the function and indented. You can selectively conceal or display nodes of the tree.
Page Fault	Collects data related to the paging system. The display indexes the data by page fault. The display shows how much time was spent in each page fault, what kind of page fault occurred, the virtual address and physical page where the page fault occurred, and the function and/or Lisp object whose reference caused the page fault.
Statistical Function Call	Collects and displays the same kind of data as does Function Call. The difference is that Statistical Function Call does not collect data on every function entry and exit; instead, it periodically samples the process being metered.
Statistical Call Tree	Collects and displays the same kind of data as does Call Tree. The difference is that Statistical Call Tree does not collect data on every function entry and exit; instead, it periodically samples the process being metered.
Statistical Program Counter	This metering type incorporates the PC Metering Tools into the Metering Interface. It collects and displays only exclusive time, and indexes it by function. It automatically executes the form a number of times, gradually zooming in on the functions where most of the time is spent.

Where to Start?

Function Call and Call Tree are the typical places to start metering. These metering types collect the same kind of data, but they display it differently. Since Function Call indexes the data by function, you can see the total amount of time spent in each function, the total number of times a function was called, and totals for paging and consing during each function. However, the Function Call display does not inform you of the calling sequence. In contrast, Call Tree indexes data by the stack trace, which informs you of the calling sequence. However, if a function was called in more than one place in the call tree, the information on that function is not merged together to show you the total number of times the function was called, total paging, total consing within that function, and so on.

Metering Long Runs

You will notice immediately that when you execute and meter code, it takes much longer than running the code normally (without metering). For a long run, it might be prohibitively time-consuming to meter every function entry and exit.

You can use the Statistical Function Call or Statistical Call Tree metering types to periodically sample the process being metered instead of collected data on every function entry and exit; this enables you to identify performance problems in runs that are too long to meter completely.

Often it is useful to use one of the statistical metering types to get a rough idea of where the performance problems are, and then narrow the scope of the metering to focus on those problems. You can edit your program to use **mi:with-metering-enabled** to specify exactly what code should be metered. You can then use Function Call or Call Tree metering types (and supply the :Only When Enabled keyword as Yes); much less data will be collected, and the metering run will go faster. You can also use **mi:with-metering-enabled** with the statistical metering types.

Function Call metering is significantly faster than Call Tree metering. Function Call and Call Tree metering take much longer than their statistical counterparts, but the data is deterministic, whereas the result of statistical metering is statistical data.

Metering for Paging Performance

The Function Call and Call Tree metering types give information on paging, including the number of page faults and the time spent in the paging system. Often that information is exactly what you are looking for, and there is no need to use Page Fault type of metering.

Page Fault metering collects and displays more detailed information on page faults, and the activities of the paging system. It is intended for people already familiar with paging systems. Page Fault metering shows what function or object took the page fault. It also shows information about fetching that occurred, which is useful for programmers who control the prefetch count by using the **:swap-recommendations** option to **make-area**.

Integration of PC Metering into the Metering Interface

The Statistical Program Counter metering type integrates the PC Metering tools into the Metering Interface. PC Metering was available prior to the Metering Interface, which was introduced in Genera 7.2. Probably for most purposes the Metering Interface will collect and display the desired kinds of data.

The Statistical Program Counter metering is useful only when the form you are metering has strictly repeatable results. (You cannot use this type of metering for Meter in Process, only for Meter Form metering runs.) The form is executed a number of times. It collects and displays the percentage of exclusive time spent in functions; this information is indexed by function.

In Statistical Program Counter metering, the sampling is supported by microcode. This means it can meter code within a **without-interrupts** special form. In contrast, in Statistical Function Call and Statistical Call Tree metering, the sampling is done from another process, so it cannot take place within **without-interrupts**. Also, the data will show time spent in escape functions, which is not shown in the other metering types.

Running and Metering the Code

When using Meter Form, once you have entered the arguments and pressed RETURN, the metering run begins. The form is executed and the desired kinds of data are collected during the execution. A metering run takes significantly longer than running the code without metering, because metering collects a lot of data. When the metering run finishes, the results are displayed in the Metering Results pane.

When using Meter in Process, the function being metered is not executed by the Metering Interface. Instead, the Metering Interface meters that function within some other process. For example, you might want to meter a function that normally runs within a network process; you would not want to call that function explicitly, but rather meter it whenever it normally is called. Once metering is started, whenever the function is called within that process, it is metered. In Meter in Process you specify explicitly when the metering should start and stop. Whenever you stop the metering, the desired kinds of data are collected into a metering run, which is displayed in the Metering Results pane.

Customizing the Display of Metering Results

The default display of metering results is a summary of the data collected. Each column is an *output field*. An output field shows a kind of data, such as consing, page faults, or time spent in a function. Output fields are divided into subfields; each subfield shows one aspect of the information.

You can request more detail by adding output fields or subfields to the display, or by expanding some piece of data already shown. You can request less detail by deleting output fields or subfields from the display. If desired, you can also set the default output fields for a given type of run, and cause other runs of that type to be displayed using the new defaults.

You can get information on the display itself, such as finding out what units are being displayed. You can use the Metering History to show the results of a previous metering run.

In many cases, you can give the following commands by using a mouse gesture. See the section "Using the Mouse in the Metering Interface".

Getting Information on the Display Itself

You can describe the meaning of a major output field or a subfield by positioning the mouse over the field and clicking Middle.

Describe Output Field Command
: Describes the meaning of the data displayed in a given output field. You can do this by clicking Middle on an output field.

Displaying Information not Currently Visible

You can request expanded data by clicking Middle on a piece of data. You can scroll the various window panes by positioning the mouse on the scroll bar and using the normal scrolling commands.

Expand Field Command
: Expands the data identified by the output field (column) and the function (row), for a given metering run. You can do this by clicking Middle on the piece of data you want to expand.

Deleting Output Fields from the Display

You can delete a major field or a subfield from the display by positioning the mouse on the field and clicking sh-Middle.

Delete Output Field Command
: Deletes an output field from the display of the metering run. You can do this by clicking sh-Middle on the output field you want to delete.

Delete Output Subfield Command
: Deletes an output subfield from the display of the metering run. You can do this by clicking sh-Middle on the output subfield you want to delete.

Adding Output Fields to the Display

Add Output Field Command
: Adds a new field to the display of the metering run. You can do this by clicking c-m-Left on a metering run.

Add Output Subfield Command
: Adds a new subfield to the display of a metering run. You can do this by clicking c-m-Left on an output subfield.

Freezing the Display while Adding or Deleting Fields

Lock Results Display Command
: Prevents updating of the display of metering results until the Unlock Results Display command is given. Useful when you are adding or deleting several output fields.

Unlock Results Display Command
: Reenables the updating of the display of metering results. Use this after you have used Lock Results Display and finished customizing the output fields.

Changing the Defaults for Displaying

Once you have added or deleted fields from a metering run, you might want to cause all future metering runs of that metering type to display the same fields that this run displays. To do so, use Set Default Output Fields for Type. That sets the default output fields for displaying runs of that metering type, but it only affects future metering runs. You can use Set Output Fields of Run from Defaults to cause an existing metering run to use the new defaults.

Set Default Output Fields for Type Command
: Sets the defaults for displaying future metering runs to be the same as the fields displayed for the given metering run. This only affects the display of metering runs of the same metering type as this run.

Set Output Fields of Run From Defaults
: Sets the output fields of the given metering-run to the defaults. This is useful when you have changed the defaults and you want a metering run to use the new defaults.

Using the Metering History

You can position the mouse over a metering run in the Metering History. Then you can click Left to display the results of the run, or Middle to describe the run, or Right for other alternatives.

Describe Metering Run Command
: Describes a metering run, including the date and time of the run, what code was metered, and the metering parameters that were used. You can do this by clicking Middle on a metering run.

Show Metering Run Command
: Displays the results of a metering run. You can do this by clicking Left on a metering run.

Delete Metering Run Command
: Deletes a metering run from the Metering History. You can do this by clicking sh-Middle on a metering run.

Re-Meter Command
: Repeats a metering run, selecting the type of metering and the code to meter from the specified metering run. You can do this by clicking s-Middle on a metering run.

Changing Other Aspects of the Display

Move Output Field Command
: Moves an output field to another position in the display of metering results. You can do this by clicking c-m-Middle on an output field.

Set Display Options Command
: Enables you to specify how the data of a metering run should be sorted and filtered; this is very useful for specifying which kind of data you are particularly interested in seeing. You can do this by clicking on [Set Display Options] in the menu.

Set Indentation Depth Command
: Specifies how many levels not to indent for displaying a call tree metering run. You can do this by clicking s-m-Middle on a displayed node, to start indentation after that node.

For information on customizing the display of a call tree metering run: See the section "Exploring a Call Tree".

Computing Fudge Factors

When you first select the Metering Interface, some initialization work goes on. The progress note says "Computing fudge factors." The goal of this computation is to measure the overhead of some of the metering tools, so the metering results do not reflect any of this overhead. The "fudge factors" are based on the hardware and software configuration of your machine.

We recommend that you wait until this process has completed before you type anything or move the mouse around. It is important for the machine to be otherwise idle, while the fudge factors are being calculated.

The computation happens more than once. If the results are roughly similar, then the Metering facility records the variations and uses these numbers to estimate the reliability and accuracy of the fudge factors. It then uses these estimates in the various Error fields in the interface. (See the section "Error Output Subfields in Metering Results".) On the other hand, if the results vary significantly, the computation is believed to have failed. This can happen if you move the mouse rapidly during the computation, for example, or if something else requires action on the part of the machine, such as unusually heavy network traffic. It might also happen if your machine has special hardware which the Metering facility did not anticipate, and for which the Metering facility cannot compensate.

When the fudge factors have not been calculated accurately, if you use Call Tree or Function Call metering, the metering results you obtain will not be accurate, because the Metering Interface cannot accurately subtract metering overhead from the results. Incorrect fudge factors can result in negative times for short functions, for example. If you decide to use the Metering Interface after the computation has failed, it is very important to display the Error information (such as the Error% field), so you can see how reliable the results are. (Even when the computation succeeds, these numbers are still valuable.)

The Metering Interface informs you if the computation has failed, and it provides some numbers which describe more about how it failed. The Metering Interface then prompts you for what to do. The choices are:

Retry once	Make one more attempt to do the computation, and prompt again if it fails.
Retry	Continue retrying the computation until the measurements are consistent.
Ignore	Use the values computed so far, even though they are possibly inconsistent.

We advise retrying the computation. To make it more likely to succeed, you might try moving your mouse off the screen, make sure the garbage collector is off, or wait for network traffic to die down.

If this is not possible, you can use Ignore to proceed past this stage. However, we recommend against using Function Call or Call Tree metering unless the fudge factors have been computed correctly.

Before using metering types Function Call or Call Tree you can recompute the fudge factors by evaluating the following form:

```
(progn
 (setq metering:*function-entry-fudge-factor-1* 0)
 (metering:enable-metering-utility))
```

Using the Mouse in the Metering Interface

It is often convenient to use the mouse to give commands in the Metering Interface. This section summarizes the available mouse gestures. The mouse gestures are arranged in patterns so it should be easy to remember how to use them. For example, clicking Middle on something describes that thing.

To take action on a metering run, you can click on a metering run in the Metering History. To indicate the current metering run, you can click on the header of the Metering Results pane (where the names of the output fields appear).

Mouse Gesture	*Action*
Left	Says "do it". When used on a metering run, it displays the results of the run. When used on a metering command in the menu, it prompts you for the arguments to the command. This mouse gesture can mean different things in different contexts; it usually enables you to do the most commonly done action on the highlighted thing. When used on a node in a call tree display, it offers to hide or show the children (whichever is appropriate).
Middle	Gives a description. Can be used on a metering run, a metering command in the menu, an output field or subfield, or a piece of data (to expand the data).
Right	Gives a menu of commands that can be given on the highlighted thing. Can be used on a metering run, an output field or subfield, or a visible node in a call tree display.
sh-Middle	Deletes the highlighted thing. Can be used to delete a metering run, to delete an output field or subfield, or to hide a node in a call tree display.
c-m-Left	Adds an output field or subfield. When used on a metering run, it adds an output field. When used on an output field, it adds an output subfield.
c-m-Middle	Moves an output field or subfield.
s-Middle	On a metering run, re-meters the run.

s-Left	On a node in a call tree display, shows all the node descendants.
s-m-Left	On a node in a call tree display, hoists the node. On a node that has already been hoisted, dehoists the node.
s-m-Middle	On a node in a call tree display, starts the indentation after that node.

The Metering Interface also offers the following command keyboard accelerators, which are based on similar accelerators in other parts of Genera:

c-sh-D	Describes the current metering run. (This is based on c-sh-D, which means describe in Zmacs and the input editor.)
c-m-R	Re-meters the current metering run. (This is based on c-m-R, which means reinvoke in the Debugger.)
c-m-U	Dehoist current node. This takes a numeric arg. An integer greater than 0 tells how many levels to dehoist. A numeric argument of 0 Dehoists all the way. The default is 1 level. (This is based on c-m-U in Zmacs and the input editor, which means "up one level of list structure".)

Interpreting the Results of a Metering Run

In general, interpreting metering results is a skill that requires practice and familiarity with the code being metered. We suggest that you do metering with specific, limited questions in mind, rather than metering with too great a scope and being overwhelmed with data, much of which is not relevant. Another approach is to start with a general question in mind, and use the metering results to help you limit the scope of future metering runs, thus enabling you to focus on the important aspects of your program's performance.

Be aware that the default display of metering results shows many fields that might be of interest, but for any given metering run, some of those fields may not be of interest. You might find it useful to delete fields from the display in order to focus on the fields that are relevant to the question being asked. On the other hand, you can also add other fields to the display.

This chapter describes the important concepts that apply to interpreting metering results. We do not document here what each of the fields of data means. We suggest that you use the online documentation available within the context of the Metering Interface. To find out what a field of data means, position the mouse over a field or subfield and click Middle to describe it.

Inclusive and Exclusive Time in Metering

When interpreting metering results, it is important to understand the meaning of *inclusive time* and *exclusive time*.

Inclusive time	The amount of time spent in function, *including* time spent in any functions that this function called.
Exclusive time	The amount of time spent in function, *excluding* time spent in any functions that this function called.

The terms "inclusive" and "exclusive" are also applied to other aspects of performance, such as consing or page faults. "Inclusive" always means that any callees of the function are included in the data, whereas "exclusive" means that the callees are excluded from the data.

For an illustration of inclusive versus exclusive time, suppose you meter the form **(format t "~&Hello, world.")** and specify the Function Call type of metering. The first function in the display is **format**. The inclusive time of format is very large; in fact it is equal to the amount of time spent in the run. However, the exclusive time of **format** is very small, because most of the time spent in **format** was actually spent in functions called by **format**.

The inclusive time of a function is the sum of the inclusive times of its callees and the exclusive time in itself.

Process Time and Time Metering Output Fields

This section describes how time spent in other processes affects metering results, and describes the difference between the output fields labeled "Time" and "Process Time".

When you meter a form and do not use **without-interrupts**, the scheduler will probably cause other processes to run, interleaved with the process in which you are metering. You will sometimes see the function **process::run-process-dispatcher** in the metering results; this function indicates that a process preemption has occurred. The Exclusive Time of that function shows how much time was spent in other processes.

The Metering Interface collects two kinds of data on time. The Process Time output field (whether Inclusive or Exclusive) includes only the time during which the process of interest was running. The Time output field (whether Inclusive or Exclusive) includes all the time spent during the metering run; that is, both the time when the process of interest was running, and the time when other processes were running. You can think of the Time output field as the result of using a stopwatch or wall clock to time your code.

The Process Time output field is usually more valuable information than the Time output field. Another interesting piece of information is the %Root subfield of Process Time, which shows the proportion of time spent in the process with respect to time spent in the metering run. This gives you an idea of the proportion of computing power that was allotted to your process during the metering run.

The metering results on paging time, number of page faults, and consing are collected on a per-process basis, so they do not reflect anything that occurred during other processes.

Metering with :Without Interrups

The Meter Form command offers the :Without Interrupts keyword, which enables you to execute and meter the form from start to finish without allowing the scheduler to give control to other processes. (The form is executed inside a **process:without-preemption** form.) Note that this can be dangerous; it causes both your program and the metering code to run without interruption. Unless the program is short, this might take a very long time and use up a lot of space on the machine.

When you do metering and any part of the function being metering uses **without-interrupts** (or other forms that disable preemption), the metering code itself is also within the scope of that **without-interrupts**. Usually the metering code takes significantly longer than the user code being metered; this has one side-effect you might notice. For a function that uses **without-interrupts**, scheduler preemptions are more likely to occur when you are running metering the function than when you run your function without metering it. This happens because of the way the scheduler works; whenever a **without-interrupts** is exited, the scheduler immediately checks to see if other processes are waiting to run, and if so, it preempts the current process. Since the **without-interrupts** surrounds both the user code and the metering code (which often takes significantly longer than your code), there is more time for other processes to be ready to run. Thus preemptions are likely to occur immediately upon exiting the **without-interrupts** form during metering.

An additional side-effect of using **without-interrupts** is that the processor must do additional work if a preemption has been requested. If preemption has been requested and **sys:inhibit-scheduling-flag** is **t**, then the machine chekcs at every unbind to see if **sys:inhibit-scheduling-flag** has been cleared. On the 3600-family machines, this check is not noticeable. On Ivory machines, this check is implemented by an expensive trap handler, and can affect metering times.

Exploring a Call Tree

For Call Tree and Statistical Call Tree types of metering, the output field labeled Function contains the "call tree" of the functions. This describes the entire calling sequence of functions that occurred. The callees of a function are displayed below the function and indented.

Each function in a call tree display is called a *node*. The first function is called the *root node*; this is the top-level function you metered. The callees of a function are known as the "children" of that node. The descendants of a node include all of its children, all of their children, and so on.

Usually the call tree is not presented in entirety because that would be too long and hard to decipher; instead a heuristic determines which nodes should be displayed. The Metering Interface offers several ways to explore a call tree display. You can open a node (show all of its children) or close it (hide all of its children).

The symbols in the call tree have the following meanings:

↓ This node is completely opened; all of its children are shown.

→ This node is not opened at all; none of its children are shown.

↔ This node is partially open; some of its children are shown.

• This is a terminal or "leaf" node; it has no children.

In many cases, you can give the following commands by using a mouse gesture. See the section "Using the Mouse in the Metering Interface".

Commands for exposing nodes further

Show Node Children Command
: Adds all the children of a node to a call tree metering display. You can do this by clicking Left on a node with undisplayed children.

Show All Node Descendants Command
: Adds all the descendants of a node to a call tree metering display. You can do this by clicking `s`-Left on a node.

Commands for concealing nodes or portions of nodes

Hide All But Path to This Node Command
: Customizes a call tree metering display to show only the path to the given node, by removing functions from the display that do not lead directly to this node.

Hide Node Children Command
: Removes all the children of a node from a call tree metering display. You can do this by clicking `sh`-Left on a node which is partially or completely open.

Hide Node Command
: Removes a node and all of its descendants from a call tree metering display. You can do this by clicking `sh`-Middle on a node.

Commands for changing the root node

Hoist Node Command
: Changes a call tree metering display to focus on a certain node as if it were the root node, and removes all functions from the display which are not descendants of this node. You can do this by clicking `s-m`-Left on a node.

Dehoist Command
: After you have hoisted a node, you can use Dehoist to restore the display to a different root node that is no longer displayed. You can do this by clicking `s-m`-Left on a node that has been hoisted.

When you hoist a node, it is often useful to add the /Root subfield to one or more

fields of interest. For example, the /Root subfield of Exclusive Time output field shows the fraction of exclusive time spent in a given function, with respect to the new root (as opposed to /Run, the fraction of time in a given function with respect to the whole run).

Command for altering the indentation

Sometimes the indentation is so great that the names of the functions are pushed off the right edge of the display. There are two solutions to this problem. First, you can scroll the window horizontally by using the usual scrolling commands. Second, you can use the Set Indentation Depth command to specify how many levels of the tree should be displayed without indentation; the following levels will be indented.

Set Indentation Depth Command

> Specifies how many levels not to indent for displaying a call tree metering run. You can do this by clicking s-m-Middle on a displayed node, to start indentation after that node.

The Set Display Option command is very valuable for filtering the results of a call tree metering run. For information, see the section "Setting the Display Options".

Different Views of the Same Metering Data

The Metering Interface can often give you different views of the same data. For example, Inclusive Time is a field that can express its data in several views; each view is expressed by a subfield of the Inclusive Time field. Some of the views include:

Total	The total time spent inclusively in the function.
/Run	A bar graph showing the proportion of time spent inclusively in this function with respect to time spent in the whole run.
%Run	Same as /Run, but expressed as a numerical percentage.
Avg	Average amount of time spent inclusively in this function, per call.
RAvg	"Reasonable Average" amount of time spent inclusively in this function, per call. This is the average of samples that fell in the main node of the histogram, only. The partitioning of a histogram into nodes is done heuristically, at runtime, and is therefore imperfect, at best. RAvg is a good first cut at filtering out noise, but closer inspection of the full contents of the histogram is sometimes necessary, if RAvg depends on false nodes.

Notice that some of the views describe the relationship between two kinds of data. For example, /Run shows the proportion of time spent inclusively in this function with respect to time spent in the whole run.

The default display shows some of these subfields. You can choose to add subfields to the display, or delete subfields from the display. See the section "Customizing the Display of Metering Results".

Often the metering results displays a summary of the collected data, and additional data is available to you. You can position the mouse over a piece of data, and click Middle to expand it. See the section "Expanding Metering Data".

Error Output Subfields in Metering Results

The Metering Interface tries to keep track of the reliability of the results of various output fields. To display this information, use the Add Output Subfield command (or click `c-m-Left` on an output field) to add one or more the the following fields:

Error	The total probable error of this field. The total value of this field is probably accurate within +/- the error.
Avg Error	The probable per-call error of this value. The average value is accurate only to within +/- the average error.
Error%	The percentage error of this value. The value itself is accurate only to within +/- this percentage.
Error/	Bar graph of the percentage error of this value.

The Error% and Error/ fields are probably the most useful, because they give a clear, visual clue as to the reliability of the metering results.

Setting the Display Options

The Metering Interface collects a huge amount of data, and it must make some decisions on how to present the data. The Metering Interface makes some decisions by default, but it also enables you to specify the criteria in which you are particularly interested.

You can use the Set Display Options command to specify how the data should be sorted and filtered most usefully for your purposes. The decisions on what criteria to use for sorting and filtering has a great effect on what results you see on the screen. As you learn more about the performance of your program, you can continue to change the display options to answer different questions.

When you use the Set Display Options command, you are prompted with all the available choices for sorting and filtering. These choices include the major output fields that are collected for the metering type, and other criteria, such as function name. You will also see the default sorting and filtering criteria, which gives you an idea of how the data you are seeing was chosen to be displayed.

By default, the Metering Interface uses the same criterion for both sorting and filtering the results. In some cases, you might wish to sort on one criterion (such as consing), and filter the data on another criterion (such as Inclusive Process Time).

For Call Tree metering runs, the display options are quite complex, so we describe them in detail here.

By default, Call Tree metering runs do not display every function called. These runs are filtered according to five criteria:

`Filter the output by:` *the category or kind of data by which to filter.*

```
Node Threshold with respect to caller %:  None 80
Node Threshold with respect to total %:  None 20
Maximum Tree Depth:  None integer
Match Functions:  None strings
```

The last four criteria are all with respect to the filtering category. For example, if the filtering category is by Total Inclusive Process Time, then the Metering Interface displays any node which took 80 percent of the Process Time of its caller, or which took 20 percent of the total Process Time of the metering run. (These are the meanings of Node Threshold with respect to caller, and with respect to total.) The Metering Interface OR's together the Node Thresholds, so if a node meets one threshold, it is displayed.

Sometimes you do a Call Tree metering run, and the functions in which you are particularly interested do not appear at all in the results. You can specify None for the two Node Threshold criteria to ensure that all function calls are shown in the display, and then use the Hide Node command to conceal nodes or branches of little or no interest.

The Maximum Tree Depth controls how many levels of the tree should be shown. If the Maximum Tree Depth is None, then all levels of the call tree that meet the Node Thresholds are shown. If it is an integer such as 5, then no more than 5 levels of the tree are shown. The Metering Interface AND's together the Maximum Tree Depth with the Node Thresholds, which means that to be displayed, a function must meet one of the Node Thresholds, and must not exceed the Maximum Tree Depth. The Maximum Tree Depth is also AND'ed with the Match Functions.

Finally, the Match Functions criterion enables you to specify one or more functions of particular interest. The Metering Interface displays only functions that "match" the specified Match Functions; this is a substring match, so "append" would match **string-append**, **append**, **sage::make-appendix**, and so on. To specify more than one Match Function, separate them by commas. (If strings are separated with spaces, then they are interpreted as one string with embedded spaces.) These Match Functions are OR'd, so if a function matches any of the Match Functions, then it is displayed.

The Metering Interface OR's the Match Functions criterion with the Node Thresholds, which means that to be displayed, a node's function must match the Match Function or the node must meet the Node Threshold.

If you want to see only those functions that match the Match Functions, then set both Node Thresholds to 100. (Note that if you set Node Threshold to None, then Match Functions will have no effect, because all nodes will be displayed.)

Note that in a call tree, if any function is shown, then the calling sequence leading to that function call is also shown. In other words, the filtering criteria do not eliminate the calling sequence leading up to a function call. The filtering criteria simply choose which function calls (and their calling sequences) are displayed.

Overview of How Metering Works

This section briefly summarizes how metering works, which should help you understand what the results mean. The metering substrate is the implementation underlying the Metering Interface.

Background on Function Call and Call Tree Metering

When you start metering something, the metering substrate sets up a trap which is entered when the code to be metered begins to execute. When this trap is entered, the metering substrate notes the time when the the function begins to run; it also begins to collect data on paging and consing. When the code being metered finishes, the trap is exited and the metering substrate notes the time when it ended (and other data).

You will notice that it takes longer to execute and meter a form than it does to execute the form without metering it. However, it is important to note that the metering substrate subtracts all of its own overhead from the metering results. That is, the metering results (time, page faults, paging system, and consing) correctly exclude any work done by the metering substrate itself.

Results Collected on a Per-process Basis

Note that during the metering, the scheduler might switch processes from the process in which the metered code is running to some other process. When this happens, the metering substrate "turns off" the collection of data on page faults, paging system, and consing. Thus the page faults, paging system, and consing results are collected on a per-process basis, and they correctly exclude any page faults, paging, or consing done within a different process. However, this is not the case for the data on time: See the section "Process Time and Time Metering Output Fields".

Metering Overhead is Excluded when Possible

One overall design goal of the metering tools was to subtract out overhead only if it could be done accurately, and when this is not possible, to document the possible anomalies in the metering results. The alternative would be to attempt to estimate the overhead, which might yield incorrect results, without the user being aware that the results were inaccurate. One example of the metering tools not subtracting out overhead is in sequence breaks.
See the section "The Effects of Sequence Breaks on Metering Results".
See the section "Metering Percentages Greater Than 100%".
See the section "Metering Overhead When :Within Functions is Used".

Background on Statistical Program Counter Metering

The Statistical Program Counter metering type (also called PC Metering) is done at the microcode level, and it works differently than the other metering types. For details: See the section "PC Metering".

PC metering divides up compiled functions into "buckets" by their locations in memory. It repeatedly executes the form provided to Meter Form, sampling the PC at a high rate. It increments the count of a bucket each time a PC falls within that range. If the number of samples in a bucket is greater than a given percentage (the *resolution percentage*) of the total number of samples, it will rerun your form and "zoom in" on this particular bucket. It "zooms in" by ignoring all PC's outside of the bucket of interest, and therefore is able to use progressively finer and finer resolution buckets. When a bucket contains a single function the "zooming" stops.

The resolution percentage controls how many buckets the metering interface "searches" (it will skip all buckets that take up less than the resolution percentage of the total), and consequently how many times it must repeat your form. The finer (or smaller) the resolution, the more times it will have to repeat your form in order to investigate more buckets.

Metering Percentages Greater Than 100%

For the output fields that give information in a ratio or percentage, such as the /Run subfield of Inclusive or Exclusive Time, sometimes the result is greater than 100 percent. In theory, a result greater than 100 percent should never happen. However, it can happen when the resolution of the the numerator is not the same as the resolution of the denominator. For example, the paging time is expressed in units of 1024 microseconds.

One goal of the Metering Interface is to give the user the most complete and undoctored information possible. That is, the Metering Interface chooses not to round the percentage down to 100 percent, but rather to give the actual data to the user, who can then interpret them.

When a ratio greater than 100 percent occurs, the bar graph displays are filled in with a darker stipple.

Metering Overhead When :Within Functions is Used

When you use the :Within function keyword to Meter Form or Meter in Process, the metered functions are encapsulated, and the encapsulations show up in the metering results. You will see functions that are obviously part of the metering facility.

These encapsulations usually take only about 4-500 microseconds, so they are usually insignificant compared to the other data. With the default filtering, they are almost never visible in the display. However, when you are metering code that takes less than a couple of milliseconds, the overhead spent in these encapsulations becomes significant and they appear in the display. In a call tree they are usually top level nodes, and so you can easily ignore them by hoisting the real top level nodes of interest. In function call metering there is currently no way to eliminate these functions automatically.

Metering Results Are Not Usually Repeatable

Note that although you can repeat a metering run, the results themselves are usually not repeatable. For example, paging performance depends on what pages are currently in virtual memory, and this is constantly changing. The metering results depend on all kinds of events that might be occurring in Genera, such as sequence breaks, incremental garbage collection, notifications, network services, and other processes. (See the section "The Effects of Sequence Breaks on Metering Results".) In addition, variations in user code itself, such as caching, often change the metering results from one run to the next.

There are techniques for looking below the surface of the metering results, to determine how reliable the results are. Sometimes it is useful to meter the same thing several times in several different ways. If some aspect of the data seems out of the ordinary or suspicious, you can look at a histogram to see whether all of the data points are clustered together, or whether a few data points are at one extreme. You can do this by expanding the displayed data, or (when available) adding the output subfields Dist or WDist. See the section "Expanding Metering Data".See the section "Distribution of Metering Data".

Expanding Metering Data

Often the metering results display a summary of the collected data, and additional data is available to you. You can position the mouse over a piece of data, and click Middle to expand it.

For example, in a Function Call metering run, the column Inclusive Time shows the *total* amount of time spent inclusively in the function, which is a sum of the inclusive time for each call of the function. The function might have been called hundreds or thousands of times. Click Middle on one of those pieces of data to get more information on the Inclusive Time. You will see information such as:

Lowest data point
Highest data point
Average
Standard deviation
Histogram of the data points

Usually you picture a histogram as having the majority of the data points gathered around one main peak. However, sometimes the data points are gathered around more than one recognizable peak; there might be an underflow peak (below the main peak) and/or an overflow peak (above the main peak). When the data points are gathered around more than one peak, the histogram is *multi-modal*. For multi-modal histograms, the display shows more than one histogram, in order to focus on each of the peaks. Thus there is always one histogram showing the main peak, and there might be one or two more histograms, showing the underflow and overflow peaks, if any.

For a graphic example: See the section "Distribution of Metering Data".

Distribution of Metering Data

Some output fields collect information about the distribution of the data points. This information is available in the "Dist" and "WDist" output subfields, which are usually not part of the default display, but can be added with the Add Output Subfield command

The "Dist" output subfield stands for "Distribution" and "WDist" means "Weighted Distribution". Each shows a small graphic representation of the data points. The middle of the graph is the average; the left-hand edge is 0, and the right-hand edge is twice the average. If there is data whose value is greater than twice the average, a gap appears afer the right-hand edge, and a smaller horizontal bar appears to its right; this represents the data whose values are greater than twice the average.

Dist	The height of each bar is related to the call count. That is, for the inclusive time output field, if several calls to a function fall within the same range of time, the height of each bar is controlled by the number of function calls within that category.
WDist	The height of each bar shows how much weight that data point contributed to the average. That is, for the inclusive time output field, several calls to a function might fall into the same range of time; the height of that bar is controlled both by the product of the number of calls in that range, and the (average) amount of time the calls in that range took.

Below we generate a metering run that has a wide distribution of data, show how the Dist and WDist fields appear on the screen, and discuss their significance.

Generating the Metering Run

We generated these results by Meter Form, where the arguments were:

`CALL-TREE Metering Run`
Created: `1/31/90 17:40:12`
Form: `(LOOP FOR I DOWNFROM 1000 TO 0 DO (FROB (FLOOR I 40)))`
What was metered: `Everything`
Count: `1`
Process: `Metering Interface 1`
Without Interrupts: `Yes`

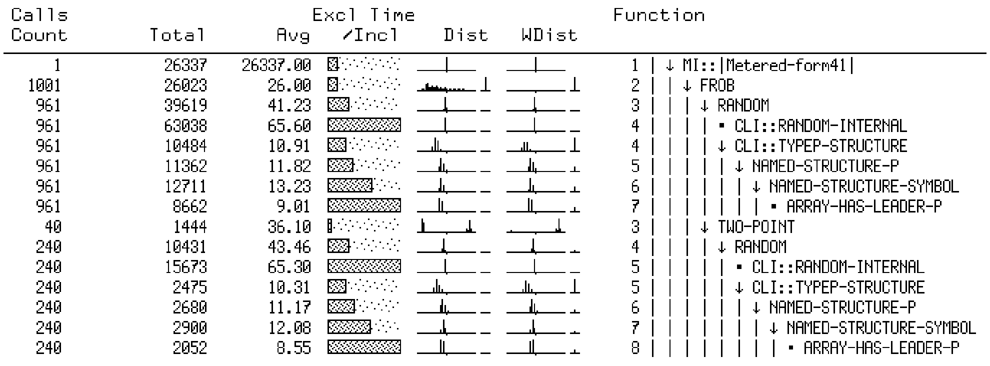
We defined **frob** as follows:

```
(defun frob (n)
  (if (plusp n)
      (let ((limit (random n)))
        (loop repeat limit))
      (two-point)))
```

```
(defun two-point ()
  (let ((n (random 2)))
    (if (oddp n)
        (loop repeat 10 doing (random 2)))))
```

The purpose of this metering run was to show a widespread distribution of data. The functions **frob** and **two-point** have no other purpose.

Visual Appearance of Dist and WDist Fields



```
Calls                      Excl Time                          Function
Count        Total      Avg   /Incl    Dist   WDist
   1         26337  26337.00                                  1 | ↓ MI::|Metered-form41|
1001         26023     26.00                                  2 | | ↓ FROB
 961         39619     41.23                                  3 | | | ↓ RANDOM
 961         63038     65.60                                  4 | | | | ▪ CLI::RANDOM-INTERNAL
 961         10484     10.91                                  4 | | | | ↓ CLI::TYPEP-STRUCTURE
 961         11362     11.82                                  5 | | | | | ↓ NAMED-STRUCTURE-P
 961         12711     13.23                                  6 | | | | | | ↓ NAMED-STRUCTURE-SYMBOL
 961          8662      9.01                                  7 | | | | | | | ▪ ARRAY-HAS-LEADER-P
  40          1444     36.10                                  3 | | | ↓ TWO-POINT
 240         10431     43.46                                  4 | | | | ↓ RANDOM
 240         15673     65.30                                  5 | | | | | ▪ CLI::RANDOM-INTERNAL
 240          2475     10.31                                  5 | | | | | ↓ CLI::TYPEP-STRUCTURE
 240          2680     11.17                                  6 | | | | | | ↓ NAMED-STRUCTURE-P
 240          2900     12.08                                  7 | | | | | | | ↓ NAMED-STRUCTURE-SYMBOL
 240          2052      8.55                                  8 | | | | | | | | ▪ ARRAY-HAS-LEADER-P
```

Figure 4. Dist and WDist Metering Output Subfields

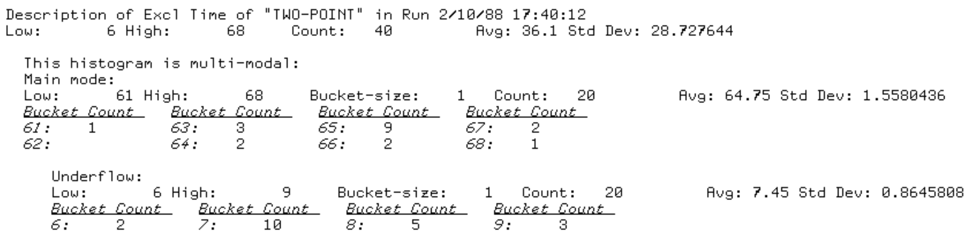
The leftmost edge of the horizontal bar means 0. The small tick that descends from the middle of the horizontal bar of each Dist and WDist entry marks where the middle, or the average value is. The rightmost edge of the horizontal bar is twice the average value.

In some cases you will notice that past the rightmost edge of the bar there is a gap followed by another, smaller horizontal bar. This gap indicates that there are data points whose value is greater than twice the average value. Figure 3 shows such gaps in all of the functions except for two. In some cases so few data points occur there that they don't show up as a vertical bar. Still, the fact that there is a bar beyond the right edge indicates that some amount of data is there.

Using Dist to Understand the Average

Notice that the first function was called once. Its inclusive time is 100% of the run. Since it was called exactly once, there is only one data point. With only one data point, the Dist field clearly shows that all the data (one value) is clustered at the middle of the average.

The other functions were all called many times, so there are many data points. Here the Dist field shows more useful information. Look at the Dist field for the function **two-point**. The data points are clustered around two different values. That is, there is a peak somewhere below the average and another peak above the average. There is no data at all appearing at the average. (This is entirely due to the definition of **two-point**.) Here the Dist tells you that the average was calculated based on two distinct behaviors. In cases such as these, probably the average itself is unimportant, but the average of each separate peak is important. You can get that information by expanding the data (clicking Middle on the row).



```
Description of Excl Time of "TWO-POINT" in Run 2/10/88 17:40:12
Low:       6 High:      68     Count:   40          Avg: 36.1 Std Dev: 28.727644

  This histogram is multi-modal:
  Main mode:
  Low:      61 High:      68     Bucket-size:    1   Count:   20          Avg: 64.75 Std Dev: 1.5580436
  Bucket Count     Bucket Count     Bucket Count     Bucket Count
  61:    1         63:    3         65:    9         67:    2
  62:              64:    2         66:    2         68:    1

     Underflow:
     Low:       6 High:       9     Bucket-size:    1   Count:   20          Avg: 7.45 Std Dev: 0.8645808
     Bucket Count     Bucket Count     Bucket Count     Bucket Count
     6:     2         7:     10        8:     5         9:     3
```

Figure 5. Expanded Data for two-point

The expanded data in Figure 4 confirms what we learned from the Dist output field; the data points are clustered around two different areas. The histogram is multi-modal (with two modes). The average of the 40 calls was 36.1. However the average of 20 calls was 64.75, and the average of the other 20 calls was 7.45.

Using WDist to Understand the Effect of Data Past the Gap

The Dist field shows where all the data points occurred. You can think of this as a seesaw where the average is the fulcrum. The seesaw is balanced on the average. If there is a gap and another horizontal bar to the far right, then the seesaw is longer on that end. The one thing that the Dist field cannot show you is how much longer that side is.

The data appearing past the gap affects the average. Its effect is based on two things: the number of data points there, and their values. The Dist field shows the number of data points there, but gives you no information about what their values. In other words, you don't know how long the gap is, or how much longer that side of the seesaw is.

The WDist field shows you the weighted distribution. It gives you an idea of how great an effect on the average the data points have. When computing an average from many data points, a small number of data points that have a high value have great impact on the average. On the other hand, a large number of data points with low values have a small impact on the average.

Look at the Dist and WDist fields for **cli::typep-function**. The Dist field shows that the great majority of data occurred well below the average. There is a gap, so some amount of data happened beyond twice the average. Since there is no vertical bar beyond the gap, very few data points occurred there. However, the WDist shows that the weighted value of those few data points was very large. In other words, a very few data points occurred quite far past twice the average. The right side of the Dist seesaw is much longer than the left.

The Effects of Sequence Breaks on Metering Results

Symbolics computers periodically take a *sequence break*, an asynchronous interrupt. During sequence breaks some mouse-tracking, I/O interrupts, and disk events might occur. Also, during the sequence break control might switch to the scheduler, which then checks to see whether other processes are waiting to be activated. The value of the variable **si:*default-sequence-break-interval*** controls how many sequence breaks occur before the scheduler is activated.

Although sequence breaks can occur even during **without-interrupts**, control is never switched to the scheduler inside a **without-interrupts**. However, there is some amount of overhead due to the sequence break itself.

The metering tools cannot measure exactly the overhead due to sequence breaks, so that overhead shows up in the metering results. The effects of sequence breaks are quite easy to recognize in metering results. For example, if you meter a function hundreds of times, you would expect the inclusive times of the function for all class to be very similar. However, you might notice that one call takes 250 microseconds or so (this varies from one machine to another) longer than the other calls; this extra time is probably due to a sequence break.

Sequence breaks are not something you should try to control; they happen in the normal operation of the machine. However, it is good to be able to recognize a sequence break, so you won't be concerned when you notice this anomaly in the metering results.

The Effects of Paging on Metering Results

When you are metering something, the metering code and data significantly increase the working set of your program. Thus, paging occurs more frequently during metering than it would otherwise occur.

If a page fault occurs during the execution of a function, the effect on the metering results is very large. You might notice this in a Call Tree metering run; a function might be called ten times, and its inclusive time for nine of those calls is approximately equal, but the inclusive time spent in one of the calls is far greater. If you notice this, look in the PFs or PS output field for that function call; probably a page fault occurred during it.

Since the Function Call metering runs display data indexed by function, the output fields show the total time spent in each function, not the time for each individual call of a given function. If you notice that a page fault occurred during a Function Call metering run, you can usually observe the effect of paging by expanding a piece of data (by clicking Middle on it) in the Inclusive Time field. This shows a histogram of the inclusive time in each of the calls to that function. You might notice one data point at the high extreme, which is probably the function call during which the page fault occurred. See the section "Expanding Metering Data".

We mention this for general background. If your goal in metering is to reduce paging time, then the extreme data points that occur due to paging represent information that is directly helpful to your goal. However, the goal might be to increase efficiency of a program in aspects other than paging. In that case, you would probably ignore the extreme data points caused by paging. If your goal is more general (simply to improve performance, however it can best be done), you would probably try to weigh the effects of paging to determine whether it is worthwhile to spend effort in reducing paging time. In this case the histogram would be useful because it probably gives you some idea how often page faults are occurring by the number of data points are at the high extreme.

Page Fault Output Fields

Several of the metering types have two output fields related to paging: PFs (number of page faults) and PS Time (paging system time).

Usually, the time spent in the paging system is more valuable information than the number of page faults. The performance of your program is affected by the time spent in the paging system; you can compare the proportion of time spent paging versus time spent in the function itself, to get an idea how significant the effects of paging are on the function.

The data under PS Time is measured in microseconds, but each individual sample is quantized in units of 1024 microseconds. This quantization is achieved by using a clock that "ticks" every 1024 microseconds. This is not equivalent to rounding each individual sample to units of 1024 microseconds. If we assume a uniform distribution of entries into the paging system with respect to this 1024 microsecond clock, then the sum of a large number of samples as a measure of "Total Time spent in the Paging System" becomes progressively more accurate. For a small number of samples it is important to remember that the possible error is plus-or-minus 1024 microseconds per sample.

It is possible to have zero page faults but still spend a small amount of time in the paging system. This might indicate a map miss (that is, the paging system experiences a miss in the hardware cache that changes a virtual address to a physical address; it is necessary to page in that cache) or some other background activity in the paging system.

The different types of page faults take different amounts of time. The longer times can be several hundred times as long as the shorter times. A reasonable time for a page fault that uses the disk is 15 or more milliseconds. An entry into the paging system that does not use the disk might be satisfied in less than a millisecond.

Interpreting Results of Meter in Process

Metering is implemented by noting when a function is entered and when it is exited. When using Meter Form, the form is always executed from start to finish, and the metering data is collected in entirety.

In contrast, when you use Meter in Process, you stop and start the metering explicitly. This means there is the possibility of starting or stopping metering in the middle of the execution of a function being metered. This is not necessarily bad, but it does have an effect on the data.

Consider what happens if you are doing a Call Tree metering run and you start the metering in the middle of a call tree. The function at the top of the tree (the root) was entered before metering was started, so the metering tools cannot collect data starting at that level of the tree. Instead, the callees of that function are metered, and they appear to be roots in the metering results. (A root function has level 1 in a call tree.) Thus the callees of the real root function appear as disconnected roots in the display, because the real root function was not metered from the beginning of its execution.

Now consider what happens if you are doing a Function Call metering run and you stop the metering before a function completes its execution. If that function was called only once, its Call Count is zero; this informs you that it was entered but not exited. However, if the function was called more than once, data has already been collected and will appear in the total time spent in the function, and the other fields. In this case the Call Count will be some integer which represents how many times the function was executed completely (both entered and exited) during metering. In this situation there is no way of knowing that the function was entered one additional time without being exited.

In summary, the metering tools collect data only for functions that are both entered and exited during metering. Sometimes you can control the starting and stopping of Meter in Process runs carefully, with the goal of not starting or stopping in the middle. In other cases, the metering results contain the information you want, even if the data is incomplete.

The Effects of the Sample Size in Statistical Metering

For Statistical Function Call and Statistical Call Tree metering, the results are more valuable as the number of samples increases. For example, if you meter a function that runs so quickly that only one sample takes place, the metering results will not be at all representative of the function's performance.

We suggest that you describe a metering run (click Middle on a metering run) to find out how many times sampling occurred during the metering. In Statistical Function Call metering you can also find out how many times a particular function was sampled (that is, how many times that function was being executed during the sampling) by expanding a portion of data (click Left on a piece of data).

Note that Statistical Program Counter metering runs your function again and again, in order to achieve a representative sample. This is not done by Statistical Function Call or Statistical Call Tree metering.

Statistical Metering of Code That Uses without-interrupts

The metering types Statistical Function Call and Statistical Call Tree do not accept the keyword :Without Interrupts. These metering types work by sampling the function periodically. Sampling cannot take place when code is running inside a **without-interrupts** form, because the scheduler will not interrupt that code in order to allow the metering process to sample the code. For example, if you wrap a **without-interrupts** form around a function and then try to meter that function with either the Statistical Function Call or the Statistical Call Tree metering type, the result will be no data. This holds for any portion of the code which is within the scope of **without-interrupts**.

The Statistical Program Counter metering type is done at the microcode level, so it can meter code within the a **without-interrupts** form.

Controlling Metering Within Lisp Code

mi:with-metering-enabled &body *body* *Special Form*

Specifies where metering should be enabled. All code in the dynamic scope of the *body* will be metered when you use Meter Form or Meter in Process, and specify :Only When Enabled. Alternatively, you can create a metering run by using **mi:with-new-metering-run** instead of the commands in the Metering Interface.

For example, you might want to exclude from metering any code that does preliminary set-up work, or performs a transition from one state to another, or cleans up afterward. The following example enables metering for two portions of the code:

```
(mi:with-new-metering-run (:metering-type :call-tree)
  (setup-code)
  (mi:with-metering-enabled
    (first-step))
  (transition-code)
  (mi:with-metering-enabled
    (second-step))
  (cleanup-code))
```

mi:with-new-metering-run *((*&key *:metering-type :name :process :without-interrupts)* &body *body)* *Special Form*

Creates a new metering run without using the Metering Interface; note that you need to use the Metering Interface to view the results. Use this special form in conjunction with **mi:with-metering-enabled**. All code of the *body* is executed, and any code within a **mi:with-metering-enabled** form is metered. The result is a metering run, which is placed in the Metering History pane of the Metering Interface. This metering run is not necessarily current, so to display it you should click Left on the metering run.

:metering-type	One of the following: **:function-call**, **:call-tree**, **:page-fault**, **:statistical-function-call**, **:statistical-call-tree**. The default is **:function-call**. See the section "Choosing a Metering Type".
:name	A string used to identify this metering run. There is no default for *:name.*
:process	The process in which to execute and meter the body. The default is the Metering Interface process.
:without-interrupts	If **t**, the code within **mi:with-metering-enabled** is executed inside a **without-interrupts** form. This means that no other process can interrupt the execution of the metering run. This

should be used with caution, because it can be dangerous for any code that does a lot of consing or takes a long time. If **nil**, the body is executed normally, and the results may show time spent in other processes. (Note that the functions running in other processes are not shown, but the time spent in them is shown). The default is **nil**. If the metering type is **:statistical-function-call** or **:statistical-call-tree**, you should not supply *:without-interrupts* as **t** because no sampling would take place. See the section "Statistical Metering of Code That Uses **without-interrupts**".

See the special form **mi:with-metering-enabled**.

Dictionary of Commands in the Metering Interface

Add Output Field Command

Add Output Field *metering-run new-field before-field*

Adds a new field to the display of the metering run. The available fields depend on the type of metering.

metering-run	A metering run. You can click on a metering run in the Metering History.
new-field	A field of data not already being displayed.
before-field	States where to place the new field in the display; the new field is placed immediately to the left of the *before-field*.

You can do this by clicking c-m-Left on a metering run. After entering the field to be added, a bar will appear somewhere within the output field. You can move that bar horizontally until you have it where you want the subfield to be placed, and then click Left to add the field to that position.

This command is available only within the Metering Interface.

Add Output Subfield Command

Add Output Subfield *metering-run new-subfield before-field*

Adds a new subfield to the display of the metering run. The available subfields depend on the type of metering.

metering-run	A metering run. You can click on a metering run in the Metering History.
new-subfield	A subfield of data not already being displayed.

before-field	States where to place the new field in the display; the new field is placed immediately to the left of the *before-field*.

You can do this by clicking c-m-Left on an output subfield. After entering the subfield to be added, a bar will appear somewhere within the output field. You can move that bar horizontally until you have it where you want the subfield to be placed, and then click Left to add the subfield to that position.

This command is available only within the Metering Interface.

Dehoist Command

Dehoist *metering-run call-tree-node keyword*

After you have hoisted a node, you can use Dehoist to change the display to use a different root node which is no longer displayed. The default is to restore the display to use the previous root node.

keyword	{:number of levels}
:number of levels	An integer or "All the way". The default is the integer that would restore the display to its previous root. All the way means to restore the display to reinstate the top-level function as the root.

You can do this by clicking s-m-Left on a node that has been hoisted.

This command is available only within the Metering Interface, and only for call tree displays.

Delete Output Field Command

Delete Output Field *metering-run output-field*

Deletes an output field from the display of the metering run.

metering-run	A metering run. You can click on a metering run in the Metering History.
output-field	An output field, which is one of the column headers. You can type in the name of an output field, or click on one in the display.

You can do this by clicking sh-Middle on the output field you want to delete.

This command is available only within the Metering Interface.

Delete Output Subfield Command

Delete Output Subfield *metering-run output-subfield*

Deletes an output subfield from the display of the metering run.

metering-run	A metering run. You can click on a metering run in the Metering History.
output-subfield	An output subfield, which is one of the column sub-headers. You can type in the name of an output subfield, or click on one in the display.

You can do this by clicking sh-Middle on the output subfield you want to delete. You can achieve the same effect by positioning the mouse over the

This command is available only within the Metering Interface.

Delete Metering Run Command

Delete Metering Run *metering-run*

Deletes a metering run from the Metering History.

metering-run	A metering run. You can click on a metering run in the Metering History.

You can do this by clicking sh-Middle on a metering run.

This command is available only within the Metering Interface.

Describe Metering Run Command

Describe Metering Run *metering-run*

Describes a metering run, including the date and time of the run, what code was metered, and the metering parameters that were used.

metering-run	A metering run. You can click on a metering run in the Metering History.

You can do this by clicking Middle on a metering run.

This command is available only within the Metering Interface.

Describe Output Field Command

Describe Output Field *output-field*

Describes the meaning of the data displayed in the *output-field*.

output-field	An output field or subfield, which is one of the column headers. You can type in the name of a field, or click on one in the display.

You can do this by clicking Middle on an output field.

This command is available only within the Metering Interface.

Expand Field Command

Expand Field *metering-run output-field function*

Expands the data identified by *output-field* (a column) and *function* (a row), for the given *metering-run.*

metering-run	A metering run. You can click on a metering run in the Metering History.
output-field	An output field, which is one of the column headers. You can type in the name of an output field, or click on in the display. This identifies the column of interest.
function	The function spec of the function for which data should be expanded. You can type in the function spec, or click on one in the display. This identifies the row of interest.

You can do this by clicking Middle on the piece of data you want to expand.

This command is available only within the Metering Interface.

Help Command in Metering Interface

Help *command-name*

Displays the documentation about the Metering Interface command.

To get a list of the Metering Interface commands, press the `HELP` key.

You can do this by clicking Middle on a command that appears in the metering command menu.

Hide All But Path to This Node Command

Hide All But Path to This Node *call-tree-node*

Customizes a call tree metering display to show only the path to the given node, by removing functions from the display that do not lead directly to this node. This does not remove any descendants of this node from the display.

You can achieve the same effect by positioning the mouse over a node, clicking Right, and choosing this command. This command is available only within the Metering Interface, and only for call tree displays.

Hide Node Children Command

Hide Node Children *call-tree-node*

Removes all the children of a node from a call tree metering display.

You can do this by clicking `sh`-Left on a node which is partially or completely open.

This command is available only within the Metering Interface, and only for call tree displays.

Hoist Node Command

Hoist Node *metering-run call-tree-node*

Changes a call tree metering display to focus on a certain node as if it were the root node. Removes all functions from the display which are not descendants of this node. When you hoist a node, it is often useful to add the /Root subfield to one or more fields of interest. For example, the /Root subfield of Exclusive Time output field shows the fraction of exclusive time spent in a given function, with respect to the new root (as opposed to /Run, the fraction of time in a given function with respect to the whole run).

You can do this by clicking `s-m`-Left on a node.

This command is available only within the Metering Interface, and only for call tree displays.

Lock Results Display Command

Lock Results Display

This is useful when customizing the display of metering results. You will notice that normally when you add or remove output fields, the Metering Interface immediately updates the display. This can be cumbersome. When you know you will be adding or deleting one output field after another, you can use this command to prevent the updating of the display. After you have finished specifying what output fields should be displayed, use Unlock Results Display to update the display of metering results.

This command is available only within the Metering Interface.

Meter Form Command

Meter Form *form metering-type what-to-meter keywords*

Immediately executes and simultaneously meters the *form* and displays the results. This command is available only within the Metering Interface.

form	Any Lisp form
metering-type	{Function Call, Call Tree, Page Fault, Statistical Function Call, Statistical Call Tree, Statistical Program Counter.} See the section "Choosing a Metering Type".
what-to-meter	{Everything, Only when Enabled, Functions.}

Everything	Meter everything within the form/process.
Only when Enabled	Meter only the code which is surrounded by a **mi:with-metering-enabled** form.
Within Functions	Meter only within the functions specified. You will be prompted for :Metered functions, and you should enter the functions of interest. See the section "Metering Overhead When :Within Functions is Used".

keywords — The keywords allow you to specify parameters that control the metering run. The keywords vary according to the *metering-type*.

All types of metering accept these keywords:

:Count	{*integer*} Execute the form this many times, and collect data only on the last run of the code. The default is 1. Note that often the first time a form is executed is not a representative run, for a variety of reasons. For example, sometimes some compilation occurs during the first execution of a form. Another example is paging; probably significantly more paging is necessary the first time a form is executed than the subsequent times. Often using this keyword is useful for metering a more representative run.
:Name	{*name*} A name to be used when printing and describing this run. This name will appear in the Metering History window pane.

The following keyword is accepted by Function Call, Call Tree, Page Fault, and Statistical Program Counter:

:Without Interrupts

{Yes No} Yes executes the form inside a **process:without-preemption** form. This means that no other process can interrupt the execution of the metering run. This should be used with caution, because it can be dangerous for any code that does a lot of consing or takes a long time. When No, the form is executed normally, and the results may show time spent in other processes. (Note that the functions running in other processes are not metered or displayed, but the time spent in them is shown). The default is No. Using :Without Interrupts is useful for preventing irrelevant data from being collected and displayed, but it does not usually make the environment more representative (unless the code is typically executed within a **process:without-preemption** form).

Note that if you specify both :Only When Enabled and :Without Interrupts as Yes, only the code within the **mi:with-metering-enabled** form is surrounded by **process:without-preemption**.

Using :Without Interrupts is particularly useful for the Statistical Program Counter metering type, because it usually yields more repeatable results. When doing metering by sampling (instead of metering constantly), the results are more valuable when all of the runs are similar. If you are metering a form which has very different results each time it is run, the results of metering by sampling will be only a rough approximation of the characteristics of all the sampled runs, and may not be a good approximation of any given run.

The following keyword is accepted by Page Fault:

:Initially Flush All Pages

{Yes No} If Yes, all pages are flushed from virtual memory prior to the metering run. The default is No. This is useful when trying to set up the metering to occur in an environment in which the virtual memory does not contain the pages of interest; this might be representative of the first time a form is executed.

The following keyword is accepted by Statistical Program Counter:

:Resolution Percentage

{*float*} The default is 0.5 percent. The resolution percentage controls how many buckets the metering interface "searches" (it will skip all buckets that take up less than the resolution percentage of the total), and consequently how many times it must repeat your form. The finer (or smaller) the resolution, the more times it will have to repeat your form in order to investigate more buckets. For more information: See the section "Overview of How Metering Works".

Meter in Process Command

Meter in Process *process metering-type what-to-meter keywords*

This command is useful when you want to meter some code that normally runs within a process. For example, you might want to meter a function that normally runs within Zmail. You don't want to use Meter Form, because that would execute and meter the function immediately; instead, you want the function to be metered whenever it is normally called. Meter in Process allows you to meter one or more functions within a process, without explicitly calling those functions.

This command offers greater control over when the metering is started and stopped than does the Meter Form command. By default, the metering starts immediately after you finish entering the Meter in Process command. To stop the metering, you should select the Metering Interface. Either press END or answer YES to the displayed question, which is "Do you want to stop metering now?"

This command is available only within the Metering Interface.

process	The process in which to meter.
metering-type	{Function Call, Call Tree, Page Fault, Statistical Function Call, Statistical Call Tree.} Note that you cannot use the Statistical Program Counter metering type with Meter in Process. See the section "Choosing a Metering Type".
what-to-meter	{Everything, Only when Enabled, Functions.}

Everything	Meter everything within the process.
Only when Enabled	Meter only the code which is surrounded by a **mi:with-metering-enabled** form.
Within Functions	Meter only within the functions specified. You will be prompted for :Metered functions, and you should enter the functions of interest. See the section "Metering Overhead When :Within Functions is Used".

keywords	The keywords allow you to specify parameters that control the metering run. The keywords vary according to the *metering-type*.

All types of metering accept these keywords:

:Name	{*name*} A name to be used when printing and describing this run. This name will appear in the Metering History window pane.
:Only When Enabled	{Yes No} Yes means to meter only those portions of the code that occur within the dynamic scope of a **mi:with-metering-enabled** form. No means to meter the specified function specs or the whole process. The default is No.
:Start and stop	{Until End Chosen, Function Keys} Specifies the way in which metering is started and stopped. The default is Until End Chosen.

Until End Chosen means that metering is started immediately after the command is entered, and it is stopped when the user presses `END` in the Metering Interface.

Function Keys means that metering is started when the user presses `FUNCTION (` and stopped when the user presses `FUNCTION )`. This allows you asynchronous control over when metering is started and stopped. See below for information on how :Start and stop interacts with :Mode lock p.

:Mode lock p	{Yes No} Specifies whether the `MODE LOCK` key controls whether metering is on or off. Yes means that metering is turned on only when the `MODE LOCK` key is pressed. No means that the `MODE LOCK` key is not used to start and stop metering. The default is No. Note that `MODE LOCK` does not give an asynchronous signal to start or stop metering; instead, it gives a synchronous signal. This means that it might take a moment for the Metering Interface to poll for the status of `MODE LOCK`, so its effect is not immediate. See below for information on how :Start and stop interacts with :Mode lock p.

The following keyword is accepted by Page Fault:

:Initially Flush All Pages

{*yes no*} If Yes, all pages are flushed from virtual memory prior to the metering run. The default is No. This is useful when trying to set up the metering to occur in an environment in which the virtual memory does not contain the pages of interest; this might be representative of the first time a form is executed.

Interaction between :Mode lock p and :Start and stop

Usually when users specify :Mode lock p as Yes, they specify :Start and stop as Until End Chosen. That way you cause metering to occur by pressing `MODE LOCK`; you cause it to stop occurring by releasing `MODE LOCK`; and you finally end the metering run entirely and display the data by selecting the Metering Interface and pressing `END`.

If you specify :Start and Stop as Function Keys and :Mode lock p as Yes, then metering is started only when you have pressed `FUNCTION (` **and** the `MODE LOCK` key is pressed. In other words, each of the keywords states how metering is started, so you must meet both requirements in order to start the metering. You can stop metering from occurring by releasing the `MODE LOCK` key, and cause metering to start again by pressing `MODE LOCK` again; you finally end the metering run entirely and display the data by entering `FUNCTION )`.

Move Output Field Command

Move Output Field metering-run output-field before-field

Moves the specified *output-field* to the left of *before-field* in the display of metering results.

You can do this by clicking c-m-Middle on an output field.

This command is available only within the Metering Interface.

Re-Meter Command

Re-Meter *metering-run*

Repeats a metering run, selecting the type of metering and the code to meter from the specified metering run. You can then change the metering parameters, or start metering with the same parameters.

metering-run	A metering run. You can click on a metering run in the Metering History.

You can do this by clicking on [Re Meter] in the Metering Interface menu, or by clicking s-Middle on a metering run.

This command is available only within the Metering Interface.

Set Default Output Fields for Type Command

Set Default Output Fields for Type *metering-run*

Sets the defaults for displaying future metering runs of a certain metering type to be the same as the output fields displayed for the given *metering-run*. Any metering runs you do from now on will use these defaults. You can cause an existing metering run to use these defaults for display by using Set Output Fields of Run from Defaults on that metering run.

metering-run	A metering run. You can click on a metering run in the Metering History.

This command is available only within the Metering Interface.

Set Display Options Command

Set Display Options *metering-run*

Enables you to specify how the data of a metering run should be displayed, including how the data should be sorted. The display options depend on the metering type of the run.

metering-run	A metering run. You can click on a metering run in the Metering History.

You can achieve the same effect by clicking on [Set Display Options] in the Metering Interface menu. This command is available only within the Metering Interface.

The Metering Interface collects a huge amount of data, and it must make some decisions on how to present the data. The Metering Interface makes some decisions by default, but it also enables you to specify the criteria in which you are particularly interested.

You can use the Set Display Options command to specify how the data should be sorted and filtered most usefully for your purposes. The decisions on what criteria to use for sorting and filtering has a great effect on what results you see on the screen. As you learn more about the performance of your program, you can continue to change the display options to answer different questions.

When you use the Set Display Options command, you are prompted with all the available choices for sorting and filtering. These choices include the major output fields that are collected for the metering type, and other criteria, such as function name. You will also see the default sorting and filtering criteria, which gives you an idea of how the data you are seeing was chosen to be displayed.

By default, the Metering Interface uses the same criterion for both sorting and filtering the results. In some cases, you might wish to sort on one criterion (such as consing), and filter the data on another criterion (such as Inclusive Process Time).

For Call Tree metering runs, the display options are quite complex, so we describe them in detail here.

By default, Call Tree metering runs do not display every function called. These runs are filtered according to five criteria:

`Filter the output by:` *the category or kind of data by which to filter.*

`Node Threshold with respect to caller %:  None` **80**
`Node Threshold with respect to total %:  None` **20**
`Maximum Tree Depth:` **None** *integer*
`Match Functions:` **None** *strings*

The last four criteria are all with respect to the filtering category. For example, if the filtering category is by Total Inclusive Process Time, then the Metering Interface displays any node which took 80 percent of the Process Time of its caller, or which took 20 percent of the total Process Time of the metering run. (These are the meanings of Node Threshold with respect to caller, and with respect to total.) The Metering Interface OR's together the Node Thresholds, so if a node meets one threshold, it is displayed.

Sometimes you do a Call Tree metering run, and the functions in which you are particularly interested do not appear at all in the results. You can specify None for the two Node Threshold criteria to ensure that all function calls are shown in the display, and then use the Hide Node command to conceal nodes or branches of little or no interest.

The Maximum Tree Depth controls how many levels of the tree should be shown. If the Maximum Tree Depth is None, then all levels of the call tree that meet the Node Thresholds are shown. If it is an integer such as 5, then no more than 5 levels of the tree are shown. The Metering Interface AND's together the Maximum Tree Depth with the Node Thresholds, which means that to be displayed, a function must meet one of the Node Thresholds, and must not exceed the Maximum Tree Depth. The Maximum Tree Depth is also AND'ed with the Match Functions.

Finally, the Match Functions criterion enables you to specify one or more functions of particular interest. The Metering Interface displays only functions that "match" the specified Match Functions; this is a substring match, so "append" would match **string-append**, **append**, **sage::make-appendix**, and so on. To specify more than one Match Function, separate them by commas. (If strings are separated with spaces, then they are interpreted as one string with embedded spaces.) These Match Functions are OR'd, so if a function matches any of the Match Functions, then it is displayed.

The Metering Interface OR's the Match Functions criterion with the Node Thresholds, which means that to be displayed, a node's function must match the Match Function or the node must meet the Node Threshold.

If you want to see only those functions that match the Match Functions, then set both Node Thresholds to 100. (Note that if you set Node Threshold to None, then Match Functions will have no effect, because all nodes will be displayed.)

Note that in a call tree, if any function is shown, then the calling sequence leading to that function call is also shown. In other words, the filtering criteria do not eliminate the calling sequence leading up to a function call. The filtering criteria simply choose which function calls (and their calling sequences) are displayed.

Set Indentation Depth Command

Set Indentation Depth *metering-run integer*

Specifies how many levels to display without indenting, when displaying a Call Tree metering run. The levels after *integer* are indented. This helps you customize the display to focus on an area of interest in the call tree, which might be many levels deep in the tree.

metering-run	A metering run. You can click on a metering run in the Metering History.
integer	Number of levels not to indent in the display. Indenting starts at the level after *integer*.

You can do this by clicking s-m-Middle on a displayed node, to start indentation after that node.

This command is available only within the Metering Interface, and only for call tree displays.

Set Output Fields of Run From Defaults

Set Output Fields of Run from Defaults *metering-run*

Sets the output fields of the given *metering-run* to the defaults. When you next display the metering run, the output fields will be displayed according to the defaults for metering runs of this type. This is useful when you have changed the defaults and you want a metering run to use the new defaults.

metering-run	A metering run. You can click on a metering run in the Metering History.

This command is available only within the Metering Interface.

Show All Node Descendants Command

Show All Node Descendants *call-tree-node*

Adds all the descendants of a node to a call tree metering display.

You can do this by clicking s-Left on a node.

This command is available only within the Metering Interface.

Show Node Children Command

Show Node Children *call-tree-node*

Adds all the children of a node to a call tree metering display.

You can do this by clicking Left on a node with undisplayed children.

This command is available only within the Metering Interface.

Show Metering Run Command

Show Metering Run *metering-run keywords*

Displays the results of the metering run in the Metering Results window, or if :Output Destination is specified, sends the results to that destination.

metering-run	A metering run. You can click on a metering run in the Metering History.
keywords	{:Output Destination}

:Output Destination
{buffer window printer} Sends the metering results to the specified buffer, window, or printer. Neither of the other two usual output destinations (files and streams) are supported.

However, you can send the output to a file by first sending it to a buffer and then saving that buffer to a file.

You can achieve the same effect by clicking on [Show Metering Run] in the Metering Interface menu, or by clicking Left on a metering run.

This command is available only within the Metering Interface.

Unlock Results Display Command

Unlock Results Display

Use this to update the display of metering results, after you have locked the display by using Lock Results Display. You can achieve the same effect by clicking on the phrase **Unlock Results Display** which appears in the Metering Results pane when the display is locked.

This command is available only within the Metering Interface.

Macros for Metering the Execution Time of Forms

Sometimes a programmer wants a simple measure of how long a Lisp form takes to execute. It might not be worthwhile setting up the Metering Interface if only a quick test is desired, or if the amount of data collected by the Metering Interface is not needed. Probably the first alternative to come to mind is the Common Lisp **time** function: See the special operator **time**.

Often, **time** is not adequate for simple metering. Since the behavior of the form varies depending on the state of the machine, one sample isn't enough. To understand the behavior of a form, it is useful to execute the form many times, and to see a histogram of the values so you can see the effects of "noise", bimodal behavior, or extreme data points. For an example: See the section "Distribution of Metering Data".

Here we document several macros that give you more flexibility and accuracy in metering the time of short forms. They are similar to **time** in the respect that they take a single form and return some simple metering information. They are more precise and informative than **time** in the measurement of time itself, although they provide less information than **time** with regard to the storage system, sequence-breaks, and consing.

The metering macros address the problems with using **time**. They enable you to meter a form by executing it many times and computing the average execution time. They simultaneously measure the metering overhead, which gives you an indication of the accuracy of the results.

Here we summarize the metering macros:

metering:with-part-of-form-measured
Executes the *form* many times, and meters the subform that is surrounded by **metering:form-to-measure**.

metering:with-form-measured
Executes the *form* many times, and meters the whole form.

metering:define-metering-function
Returns a compiled function which can be used to meter the *form* more than once. Useful when you know in advance that you will be metering a form repeatedly.

metering:measure-time-of-form
Has the same effect as **metering:define-metering-function** in that it uses a compiled function to meter the *form*, but instead of returning the metering function, it runs it once to meter the form. The metering function is not saved for further use.

Probably **metering:define-metering-function** is the most generally useful of the

group. It enables you to meter the form more than once. However, if you want to execute a form and meter only a portion of it, use **metering:metering-with-part-of-form-measured**.

Here we document each of the metering macros:

metering:with-part-of-form-measured *(*&key *(:no-ints* **'t***) :verbose :values (:time-limit* **1***) :count-limit)* &body *form* *Macro*

Executes the *form* many times, and meters the subform that is surrounded by **metering:form-to-measure**. Note that **metering:form-to-measure** does not return the value of the subform it surrounds.

If you want to meter the whole form, the macro **metering:with-form-measured** is more convenient: See the macro **metering:with-form-measured**.

This tells you how many microseconds (on average) were needed to evaluate the form or subform. It also measures the overhead of the metering code.

By default, the average time and the average overhead are printed out in a mouse-sensitive way. You can click on these averages to display the histogram of values that were used to compute them.

The keywords *:time-limit* and *:count-limit* can be used to control how many times the form is evaluated. The *:verbose* and *:values* keywords control the output of this macro. See the section "Keyword Options for Metering Macros".

This form is most useful in compiled code. When they are used in interpreted code, the results are primarily a measurement of the interpreter, and not the form.

See the section "Output of the Metering Macros".

metering:with-form-measured *(*&key *(:no-ints* **'t***) :verbose :values (:time-limit* **1***) :count-limit)* &body *form* *Macro*

Executes the *form* many times, and meters the whole form. This is an abbreviation for the most common case of **metering:with-part-of-form-measured**, in which the entire *form* is metered. If you want to meter a subform within a form, use **metering:with-part-of-form-measured**. See the macro **metering:with-part-of-form-measured**.

This tells you how many microseconds (on average) were needed to evaluate the form or subform. It also measures the overhead of the metering code.

By default, the average time and the average overhead are printed out in a mouse-sensitive way. You can click on these averages to display the histogram of values that were used to compute them.

The keywords *:time-limit* and *:count-limit* can be used to control how many times the form is evaluated. The *:verbose* and *:values* keywords control the output of this macro. See the section "Keyword Options for Metering Macros".

This form is most useful in compiled code. When they are used in interpreted code, the results are primarily a measurement of the interpreter, and not the form. See the section "Output of the Metering Macros".

metering:define-metering-function *name args (*&key *(:no-ints* **t***) :verbose :values :count-limit :time-limit)* &body *form* *Macro*

Returns a compiled function which can be used to meter the *form* more than once. This is useful when you know in advance that you will be metering a form repeatedly.

This is an abbreviation for the following form (where where *keywords1* and *keywords2* are constructed according to the rules explained below):

```
(compile (defun function-name (arglist . keywords1)
             (metering:with-form-measured (keywords2) form)))
```

Any keywords specified in **metering:define-metering-function** will not be accessible in the function *function-name*. Any keywords omitted from the keyword list in **metering:define-metering-function** will become part of the arglist of *function-name*.

The compiled function *function-name* can be used to execute the *form* many times, and meter it. It tells you how many microseconds (on average) were needed to evaluate the form or subform. It also measures the overhead of the metering code.

By default, the average time and the average overhead are printed out in a mouse-sensitive way. You can click on these averages to display the histogram of values that were used to compute them.

The keywords *:time-limit* and *:count-limit* can be used to control how many times the form is evaluated. The *:verbose* and *:values* keywords control the output of this macro. See the section "Keyword Options for Metering Macros". See the section "Output of the Metering Macros".

metering:measure-time-of-form *(*&key *(:no-ints* **'t***) :verbose :values :time-limit :count-limit)* &body *form* *Macro*

Has the same effect as **metering:define-metering-function** in that it uses a compiled function to meter the *form*, but instead of returning the metering function, it runs it once to meter the form. The metering function is not saved for further use.

See the macro **metering:define-metering-function**. See the section "Keyword Options for Metering Macros". See the section "Output of the Metering Macros".

Keyword Options for Metering Macros

The metering macros accept the following keyword arguments:

:no-ints The default is **t**. A non-**nil** value causes the metering to be done inside a **without-interrupts**. If the form is exceptionally long, or if it relies on other processes to work correctly, then **:no-ints** should be **nil**. If you specify **:no-ints t**, and the length of the form times **:count-limit** (if specified) is greater than five minutes, you will be prompted to check if you really want to disable the scheduler for that long.

:time-limit
Value is an integer, expressing a number of seconds. The default is 1 second. This specifies that the form should be repeated until this many seconds of real-time have elapsed. This includes the amount of time spent recording the metering results. **:time-limit** and **:count-limit** are mutually exclusive keywords.

:count-limit
Value is an integer. This specifies that the form should be repeated this many times. **:time-limit** and **:count-limit** are mutually exclusive keywords.

:verbose The default is **nil**. A non-**nil** value causes the full histograms to be printed out instead of just the averages. This keyword is overridden by the **:values** keyword.

:values The default is **nil**. A non-**nil** value causes nothing to be printed out; the metering results are represented by three returned values. The first is the average time, the second is the histogram of the times for evaluation of the form, and the third is the histogram for the overhead loop. You can get other information by using the histograms as described below.

Using the Histograms

The following functions can be done to the histograms returned when you give the **:values** option:

```
;; To display the results of a histogram
(metering:display-collector histogram stream)
```

```
;; Returns the average value of the histogram
(metering:average histogram)

;; Returns the total of the data in the histogram
(metering:total histogram)

;; Returns standard-deviation of the data
(metering:standard-deviation histogram)

;; Maps over the buckets in the histogram
(metering:map-over-histogram-buckets
  histogram #'(lambda (low high count)))
```

To display a number in such a way so that clicking Middle will expand the data into a full histogram, you must present the data with the **'metering:metering-results** presentation type. For example:

```
(dw:with-output-as-presentation (:object rainfall
                                 :type 'metering:metering-results)
  (format t "~&The average rainfall was ~,5F inches."
          (metering:average rainfall)))
```

Output of the Metering Macros

By default, the output displays two or three quantities.

Average time

The first quantity, "Average time", is the time a single execution of the body took to execute, averaged over some number of repetitions. The number of repetitions can be controlled by using either the **:count-limit** or **:time-limit** keyword options.

Average clock overhead

The second quantity, "Avg clock overhead", is the amount of time spent by identical metering code metering the empty loop. This is provided for calibration (you can subtract this time from the "Average time") and to provide some measure of the significance of the result (if the "Average time" is close to the value of "Avg clock overhead", the results are suspect.

Clock variation

The third quantity may or may not be present. It begins with the phrase "A second sampling of the clock ....". This is printed out when a second measurement of the empty loop does not agree with the first. This indicates that something is making it hard to get reproducible metering results. This can be caused by many things. Although it does not always mean you should repeat the metering, it does mean that you should look at the numbers produced on such a run a little more carefully than normal.

The decision whether to print out the third value is controlled by the variable **metering:*tolerable-clock-variation***. Its value is a number between 0 and 1, which represents a percentage. When the two numbers differ by more than this percentage then the third value is printed.

Histograms are available

By default, the average time and the average overhead are printed out in a mouse-sensitive way. You can click on these averages to display the histogram of values that were used to compute them.

Usually you picture a histogram as having the majority of the data points gathered around one main peak. However, sometimes the data points are gathered around more than one recognizable peak; there might be an underflow peak (below the main peak) and/or an overflow peak (above the main peak). When the data points are gathered around more than one peak, the histogram is *multi-modal*. For multi-modal histograms, the display shows more than one histogram, in order to focus on each of the peaks. Thus there is always one histogram showing the main peak, and there might be one or two more histograms, showing the underflow and overflow peaks, if any.

PC Metering

PC Metering was available prior to the Metering Interface, which was introduced in Genera 7.2. Probably for most purposes the Metering Interface is a more convenient way to meter programs. See the section "Metering Interface".

Program counter (PC) metering is a tool to allow the user to determine where time is being spent in a given program. PC metering produces a histogram that you can interpret to improve the performance of your program.

The mechanism of PC metering is as follows. At regular intervals, the front-end processor (FEP) causes the main processor to task switch to special microcode. This microcode looks up the macro PC that contains the virtual address of the macroinstruction that the processor is currently executing. If this virtual address falls outside the *monitored range*, the microcode increments a count of the number of PCs that missed the monitored range. If the address is within the monitored range, the microcode subtracts the bottom of the monitored range from the PC, leaving a word offset. It then divides the word offset by the number of words per *bucket* and uses that as an index into the *monitor array*. Next, it increments that indexed element of the monitor array. This can only measure statistically where the macro PC is pointing; for the results to be valid, a relatively large number of samples per bucket must be available.

For Symbolics 367X, 365X, 364X, and 363X machines with Rev. 4 of the input/output board (this denotes machines with digital audio), PC metering is performed in the audio microtask and samples at a rate of 50,000 samples per second. This is useful for metering almost all code.

For Symbolics 3600 computers with Rev. 2 of the input/output board, the FEP samples at about 170 samples per second, so PC monitoring is probably valid only for sessions that take longer than five to ten seconds.

You specify a range of the program to be monitored. The range is specified by lower and upper bounding addresses, and compiled functions that lie between those addresses are monitored. The range is divided into some number of buckets. The relative amount of time that the program spends executing in each bucket is measured.

The parameters you specify are the range of addresses to be monitored, the number of buckets, and an array with one word for each bucket.

Some of the metering functions deal with *compiled functions*. In this context a compiled function is either a compiled code object or an **sys:art-16b** array, into which escape functions (small, internal operations used by the microcode) compile.

meter:make-pc-array *size* *Function*

Makes a PC array with *size* number of buckets. This storage is wired, so you probably do not want this to be more than about 64. pages, or **(* 64 sys:page-size)** words.

meter:monitor-all-functions *Function*

Changes the microcode parameters so that the monitor array refers to every possible function in the Genera world at the time of the execution of **meter:monitor-all-functions**. This usually causes many functions to map into a single bucket, and is therefore useful in obtaining a first estimate of which functions are using a significant portion of the execution time.

meter:setup-monitor &optional (*range-start* **0**) (*range-end* **268435456**) *Function*

Monitors the region between *range-start* and *range-end*.

meter:monitor-between-functions *lower-function upper-function* *Function*

Monitors all functions between *lower-function* and *upper-function*. This does not work in some situations, such as:

- You compile a function from a buffer, which puts its definition outside the range

- A previous region is extended, and new functions go there instead of in monotonically increasing virtual addresses.

Example:

```
    (defun start-of-library ()())
                ...code...
                (defun end-of-library ()())
       (meter:monitor-between-functions #'start-of-library #'end-of-library)
```

meter:expand-range *start-bucket* &optional (*end-bucket start-bucket*) *Function*

Changes the microcode parameters so that the entire monitor array refers only to the functions previously contained within the range specified by *start-bucket* and *end-bucket*. *start-bucket* and *end-bucket* are inclusive bounds.

meter:report *&optional function-list* *Function*

Prints a summary of the data collected into the monitor array. You should not have to supply the *function-list* argument.

meter:start-monitor &optional (*clear* **t**) *Function*

Enables collection of PC data. If *clear* is not **nil**, the contents of the monitor array are cleared. If *clear* is **nil**, the array is not modified, so that the new samples are simply added to the old.

meter:stop-monitor *Function*

Disables further collection of PC data.

meter:print-functions-in-bucket *bucket* *Function*

Prints all the compiled functions that map into the specified *bucket*.

meter:list-functions-in-bucket *bucket* *Function*

Returns a list of all the compiled functions that map into the specified *bucket*.

meter:range-of-bucket *bucket* *Function*

Returns the virtual address range that maps into the specified *bucket*.

meter:with-monitoring *clear body...* *Function*

Enables monitoring around the execution of *body*. If *clear* is not **nil**, clears the monitor array first. See the function **meter:start-monitor**.

meter:map-over-functions-in-bucket *bucket function* &rest *args* *Function*

Calls *function* for every compiled function in the specified *bucket*. The first argument to *function* should be the compiled function, and any remaining arguments are *args*.

meter:function-range *function* *Function*

Returns two values, the buckets that contain the first and last instructions of *function*.

meter:function-name-with-escapes *object* *Function*

If *object* is a compiled function, returns the function spec of the compiled function. Otherwise, returns **nil**.

Debugger

Overview of the Debugger

Genera, the Symbolics software environment, offers you a host of powerful debugging tools. The most comprehensive of these tools is the Symbolics interactive Debugger and its window-oriented counterpart, the Display Debugger.

Other debugging tools are:

- The *Trace* facility, which performs certain debugging actions when a function is called or when a function returns. See the section "Tracing Function Execution".

- The *Advise* facility, which modifies the behavior of a function. See the section "Advising a Function".

- The *Step* facility, which allows you to execute interpreted forms in your program, one at a time, so that you can examine what is happening when execution suspends at every "step." See the section "Stepping Through an Evaluation". The Debugger's :Single Step command also performs stepping through compiled functions. See the section "Single Step Command".

- The *evalhook* facility, which allows you to get a particular Lisp form whenever the evaluator is called. The Step facility also uses **evalhook**. See the section "A Hook Into the Evaluator".

- The *Inspector* is a window-oriented program that lets you inspect data objects and their components. See the section "The Inspector".

- *Peek* is a program that gives a dynamic display of various kinds of system status. See the section "Using Peek".

- The *Metering Interface* allows you to meter the performance of a form, function, or process. See the section "Metering Interface".

For information on the Display Debugger, see the section "Using the Display Debugger".

In the Genera software environment, unlike more traditional programming environments, you do not have to include the Debugger explicitly when you compile your programs. Generally, you can debug your code as you write it without having to perform a series of complicated compiling, loading, and executing procedures between source code development and debugging.

Because Symbolics user-interface features allow you to perform many Symbolics activities simultaneously — Zmacs, Zmail, the file system, a Dynamic Lisp Listener, and so on — debugging becomes an easy task, regardless of how many system activities you are using. You can move in and out of the Debugger as easily as you can move in and out of any other activity in Genera.

For example, the Debugger command c-E (:Edit Function) brings up a specified function for you to edit in a Zmacs editor window. This is useful when you have found the function that caused the error and want to edit that function immediately. Another command, c-M (:Mail Bug Report), creates a bug report message in a mail window and puts a backtrace into it. While composing the bug report, you can switch back and forth between the Debugger and the mail window.

The Debugger is there whenever you need it. It is invoked whenever an error occurs in your program's execution or the execution of a system function. That is, your machine brings you into the Debugger whenever it encounters an error that is not handled by a condition handler, for example, when you reference an unbound variable. See the section "Entering and Exiting the Debugger". Once in the Debugger, you are given a choice of actions that can correct the error. These actions are called *proceed* and *restart options*. See the section "Proceeding and Restarting in the Debugger".

You can also enter the Debugger explicitly, at any time, by pressing m-SUSPEND or c-m-SUSPEND. Or you can make your program enter the Debugger by inserting the **break** or **zl:dbg** function into your program code. See the section "Entering and Exiting the Debugger".

Upon Debugger entry, besides selecting one of the proceed and restart options, you can enter any of the Debugger's commands. These commands are full-form English commands, built on the normal Command Processor (CP) substrate. In fact, several Debugger commands are in the global command table. For more information on Debugger commands, see the section "Entering a Debugger Command" and see the section "Debugger Command Descriptions".

In the Debugger you can also evaluate a form in the lexical (user-program) context of the current frame. This context is referred to as the Debugger's *evaluation environment*. You can think of the Debugger's evaluation environment as a special read-eval-print loop that not only evaluates forms but also evaluates them in the context of the suspended function, where the lexically apparent values of all the local variables are accessible. For more information on the evaluation environment, see the section "Evaluating a Form in the Debugger".

Like other output in the Genera software environment, Debugger output is mouse sensitive, so you can perform many useful Debugger operations using the mouse. For more information on mouse capabilities, see the section "Using the Mouse in the Debugger".

The Debugger also provides some online help facilities. For more information on help facilities, see the section "Getting Help for Debugger Commands".

For complete information on the uses of these features and other Debugger features, see the section "Using the Debugger". For descriptions of all Debugger commands, see the section "Debugger Command Descriptions".

In general, you use the Debugger when:

- Your program triggers the Debugger because garbage — an unbound variable or too many arguments perhaps — was passed to a function, and you want to find out where the garbage came from. See the section "Analyze Frame Command".

- You want to see what's happening in the sequence of function calls just executed, including a history of these function calls, the argument values passed, the local-variable values, the source code, and the compiled code. See the section "Show Backtrace Command". See the section "Debugger Commands for Viewing a Stack Frame".

- You want to find out who or what is referencing a special variable or any other location in memory. See the section "Monitor Variable Command".

- You want to perform debugging operations using the mouse. See the section "Using the Mouse in the Debugger".

- You want to continue program execution, proceed from an error, restart a function, return from a function, or throw through a function. See the section "Debugger Commands to Continue Execution".

- Your condition handler does not work properly, and you want to debug this handler when it is encountered. See the section "Enable Condition Tracing Command".

- You want to edit your function's source code in Zmacs immediately after you have found the error. See the section "Edit Function Command".

- You want to put a Debugger backtrace into a mail message and send this message as a bug report. See the section "Mail Bug Report Command".

- You want to use Debugger breakpoint commands, instead of using the Trace facility or inserting a function in your code, to set Debugger breakpoints. See the section "Debugger Commands for Breakpoints and Single Stepping".

Overview of Debugger Commands

The Debugger offers more than 50 full-form English commands, which are implemented as CP commands. Debugger commands are entered inside the Debugger at the Debugger's command prompt, a right arrow (→). Commands fall into eight general categories:

- Commands for viewing a stack frame
- Commands for stack motion
- Commands for general information display
- Commands to continue execution
- Trap commands
- Commands for breakpoints and single stepping
- Commands for system transfer
- Miscellaneous commands

Most Debugger commands have corresponding key-binding accelerators, which means you can press a combination of one or more keys in place of the command. For example, you can press the accelerator `c-E` instead of entering the command :Edit Function.

Most Debugger commands also have keywords you can use to modify the command's behavior.

Many Debugger commands share the global command table. Therefore, you can enter these commands while you are in a CP command loop. You do not have to be in the Debugger. Note, however, that when you enter these commands while in the Debugger, you must type a preceding colon with every command; for example, you must type :Set Breakpoint in the Debugger.

These commands are:

- :Clear All Breakpoints
- :Clear Breakpoint
- :Disable Condition Tracing
- :Edit Function
- :Enable Condition Tracing
- :Monitor Variable
- :Set Breakpoint
- :Set Stack Size
- :Show Breakpoints
- :Show Compiled Code
- :Show Function Arguments
- :Show Monitored Locations
- :Show Source Code
- :Show Standard Value Warnings
- :Unmonitor Variable

For general information on using the Debugger, see the section "Using the Debugger". For documentation of each Debugger command, see the section "Debugger Command Descriptions".

Overview of Debugger Evaluation Environment

In the Debugger, you can evaluate a form as easily as you can in a Dynamic Lisp Listener read-eval-print loop. Evaluating a form in the Debugger, however, is particularly useful because you are evaluating the form in the context of a user program and the current stack frame. This means you can see the value of Lisp objects at the point in program execution where an error occurred or at the precise place in your program where you explicitly suspend execution and invoke the Debugger. You can even reference lexical (local) variables at the point where execution suspends.

Evaluating a form in the Debugger is a simple task. If you type a character other than the first character in a Debugger command — a colon or accelerator key — the Debugger immediately brings you into its evaluation environment. In other words, just type the form. Evaluation in the proper environment happens automatically.

For complete information on how to evaluate a form in the Debugger: See the section "Evaluating a Form in the Debugger".

Overview of Debugger Mouse Capabilities

When the output generated by Debugger commands is displayed in a Dynamic Window, it is mouse sensitive. You can perform several useful debugging operations simply by using the mouse to click on something. Some of these operations include: setting a breakpoint, monitoring a variable or another location in memory, evaluating a form, editing a function, setting the current frame, and choosing a proceed or restart option. The mouse documentation line at the bottom of the screen tells you what actions are available for the currently highlighted output item.

Besides performing certain mouse operations by clicking directly on displayed Debugger output, you can use menus to perform the usual large variety of other types of operations on Debugger output, just as you can with other kinds of output generated in the Genera software environment.

For more information on using the mouse in the Debugger: See the section "Using the Mouse in the Debugger".

Overview of Debugger Help Facilities

The Debugger provides online help for Debugger commands and their components, such as keywords. You can get help for all Debugger commands by typing c-HELP, which displays brief command descriptions and available key-binding accelerators. For more information about Debugger help: See the section "Getting Help for Debugger Commands".

Entering and Exiting the Debugger

Virtually anywhere in Genera, the Debugger is invoked during the signalling of an error to which no condition handlers are bound. The Debugger is invoked not only when errors occur during program execution, but also when errors occur in relation to functions that control various system operations, such as loading patches and executing commands in the Dynamic Lisp Listener.

The Debugger is invoked within the process that signalled the error. Since the Debugger is not a separate process, several distinct processes can all be in the Debugger at the same time, independently.

Usually, entry to the Debugger is triggered by an error. However, you can also enter the Debugger explicitly at any time. You exit the Debugger via the ABORT key, the :Abort command, or by invoking a proceed or restart handler.

This chapter describes various ways to enter and exit the Debugger.

Entering the Debugger

Enter the Debugger in one of three ways:

- Automatically, by causing an error.
- Explicitly, by pressing m-SUSPEND or c-m-SUSPEND.
- Through your program execution, by inserting and calling the **break** function or the **zl:dbg** function.

Entering the Debugger by Causing an Error

The Debugger is invoked automatically when errors occur during your program execution, or during the execution of system functions, or when you explicitly cause an error.

Error Display

Upon entering the Debugger via an error, you receive an error message and a choice of actions to take. Errors are signalled by the microcode and by Lisp programs by **error** or related functions.

For example, suppose you trigger an error by using an unbound variable, **FOO**. The Debugger error display might look like this:

```
Trap: The variable FOO is unbound.

SI:*EVAL:
   Arg 0 (SYS:FORM): FOO
   Arg 1 (SYS:ENV): NIL
   --Defaulted args:--
   Arg 2 (SI:HOOK): NIL
s-A, RESUME:   Supply a value to use this time as the vaue of FOO
s-B, s-sh-C:   Supply a value to store permanently as the value of FOO
s-C:           Retry the SYMEVAL instruction
s-D, ABORT:    Return to Lisp Top Level in Dynamic Lisp Listener 1
→
```

The word Trap, Error, or Break followed by a boldface message, such as the line at the top of this display, indicates you have entered the Debugger. Trap, Error, and Break are the most common causes, although there are others. Trap, Error, and Break have the following meanings:

- Trap indicates an error signalled by the microcode.
- Error indicates an error signalled by a program.
- Break indicates entry to the Debugger by keystroke (m-SUSPEND or c-m-SUSPEND), the **break** function, or the **zl:dbg** function.

The message that follows describes the error in English — in this example, an unbound variable. The next five lines in the example show the stack frame in which the error occurred, the function that was being called, and the current values of arguments. The next six lines are available proceed and restart options, which are discussed in the next section.

The right-facing arrow at the end of the display (→) is the Debugger's command prompt, which waits for you to enter a command. Multiple arrow prompts indicate recursive invocations of the Debugger. For more information on recursive Debugger invocations: See the section "Using Recursive Debugger Invocations".

Debugger Proceed and Restart Options

Whenever you enter the Debugger, either for the first time or recursively, it displays a list of possible actions for you to take. These actions, called *proceed* and *restart options*, allow you to proceed (continue program execution) from the error, leave the Debugger, restart (return to) a previous activity, or take some other action.

A list of proceed and restart options might look like this:

```
s-A, RESUME:   Supply a value to use this time as the vaue of FOO
s-B, s-sh-C:   Supply a value to store permanently as the value of FOO
s-C:           Retry the SYMEVAL instruction
s-D, ABORT:    Debugger command level 1
s-E:           Return to Lisp Top Level in Dynamic Lisp Listener 1
```

You can select one of these options by pressing the keys that appear in the left-hand column or by clicking on an option with the mouse. All of these options are bound to the SUPER key.

For more information on proceed and restart options: See the section "Proceeding and Restarting in the Debugger".

Entering the Debugger with m-SUSPEND, c-m-SUSPEND

When you want to enter the Debugger explicitly, without waiting for an error to occur, you can do so in one of two ways:

Press m-SUSPEND

Press c-m-SUSPEND

If the program you are running is waiting for keyboard input, use m-SUSPEND.

If you want to enter the Debugger while your program is actually running, use c-m-SUSPEND, which calls the Debugger immediately, at any time, regardless of your program's state.

Entering a Break Loop with SUSPEND, c-SUSPEND

Using SUSPEND or c-SUSPEND, without the META key modifier, causes entry to a *break loop*. A break loop, also called a *breakpoint loop*, is a Dynamic Lisp Listener read-eval-print loop that comes up on your screen in a special small "breakpoint" window whenever you temporarily suspend an activity, such as Zmacs or Zmail. This allows you to suspend into a Dynamic Lisp Listener instead of pressing SELECT L to actually change activities.

Do not confuse this break loop with a Debugger breakpoint. A break loop is a Dynamic Lisp Listener read-eval-print loop, which is activated when you suspend your current activity. A Debugger breakpoint, which you set via the Set Breakpoint command, the **break** function, the **zl:dbg** function, m-SUSPEND, or c-m-SUSPEND, suspends into the Debugger, usually for the purpose of debugging a program. Once in the Debugger, you can evaluate forms using the Debugger's read-eval-print loop (evaluation environment).

When you want to enter a break loop, you can do so in one of two ways:

Press SUSPEND

Press c-SUSPEND

If the program you are running is waiting for keyboard input, use SUSPEND.

If you want to enter a break loop while your program is actually running, use c-SUSPEND, which brings up the break loop immediately, at any time, regardless of your program's state.

To leave the break loop and return to your previous activity, press the RESUME key.

Entering the Debugger with break and zl:dbg Functions

A third way of entering the Debugger is by inserting the **break** or the **zl:dbg** function into your program's source code. These functions can help you detect errors when you place one of them at strategic points in your program — places where you can examine the stack and pinpoint probable causes of errors.

The following paragraphs provide more information on the **break** and the **zl:dbg** functions.

break &optional *format-string* &rest *format-args* *Function*

Like **zl:dbg**, when evaluated, causes entry to the Debugger (a Debugger `Break`). However, **break** takes a *format-string* and *format-args* instead of a process.

The *format-string* is a user-written error message that is printed in the Debugger's `Break` message whenever **break** is encountered and you enter the Debugger. *format-args* are the **zl:format**-style arguments to **zl:format** directives in *format-string*.

break is a temporary way to insert Debugger breakpoints into your program while you are debugging it. It is not designed for permanent use in your program as a way of signalling errors. Therefore, you would use this function only for the duration of your debugging session. Continuing from **break** will not trigger any unusual recovery action.

zl:dbg &optional *process* *Function*

Forces *process* into the Debugger so that you can look at its current state. **zl:dbg** sets up a restart handler for `ABORT` and `RESUME` that exits from the **zl:dbg** function back to the original process. The message for this restart handler is "Allow process to continue". You can use :Throw, :Return, :Reinvoke, and other similar Debugger commands when you enter the Debugger via **zl:dbg**.

- With no argument, it enters the Debugger as if an error had occurred for the current process. It is not an error; in particular, **catch-error** does not handle it. You can include this form in program source code as a means of entering the Debugger. This is useful for breakpoints and causes a special compiler warning.
- With an argument of **t**, rather than a process, window, or stack group, it finds a process that has sent an error notification.

Suppose you are running in process **X** and you use **zl:dbg** on some process **Y**. Process **Y** is forced into the Debugger, no matter what it is doing. Technically, it is "interrupted", similar to how `c-SUSPEND` and `c-m-SUSPEND` work. Process **Y** starts running the Debugger, using the stream ***debug-io***, which gets the same stream as was bound to ***terminal-io*** in process **X**. At this time, process **X** waits in a state called DBG until process **Y** leaves the Debugger, and so process **X** does not contend for the stream.

Exiting The Debugger

To exit the Debugger, use the ABORT key, the :Abort command, or invoke a restart option. ABORT, which is a very powerful command, takes you out of the process that received the error.

If an error brings you into the Debugger, and you don't want to use the Debugger, you can get back to the top command level in which your program is running by simply pressing ABORT. In this case, the top command level is the level in which you were working prior to the Debugger call — the first and only invocation of the Debugger.

If you have made a number of errors, or if you have called the Debugger explicitly several times, then you probably are in the middle of a series of recursive Debugger invocations. In this case, ABORT returns you to the previous invocation. If you keep pressing ABORT, the invocations unwind until you actually leave the Debugger and return to top level.

If you find yourself in the middle of many recursive Debugger invocations, or if you are in the Debugger's evaluation environment, and you want to leave the Debugger immediately: Press m-ABORT, which brings you back to top level immediately.

Using the Debugger

This chapter offers some general instructions for using the Debugger. Specifically, it covers the following topics:

- Entering a Debugger command
- Getting help for Debugger commands
- Proceeding and restarting in the Debugger
- Evaluating a form in the Debugger
- Using recursive Debugger invocations
- Using the mouse in the Debugger
- Creating Debugger proceed menus

Entering a Debugger Command

Entering a Debugger command is almost identical to entering a command in the Command Processor (CP) to a Dynamic Lisp Listener. In fact, you can enter many Debugger commands in both the Debugger and the CP because these commands share the same command table. If you have not done so already, read the section "Entering a Command" about entering commands in the CP. For more general information about the Command Processor: See the section "Communicating with Genera".

When an error brings you into the Debugger, or when you enter the Debugger through m-SUSPEND, c-m-SUSPEND, **break**, or **zl:dbg**, the Debugger prompts you for commands. The Debugger's command prompt is a right arrow (→). Recursive Debugger invocations prompt you with two or more arrows. For example, the third Debugger invocation prompts you with →→→. See the section "Using Recursive Debugger Invocations".

At its command prompt, the Debugger expects a full-form command, such as :Show Backtrace, or a command accelerator, such as `c-B`. When giving a full-form command in the Debugger, you must precede the command with a colon. For example:

```
:Show Backtrace
```

If you enter anything other than a colon or an accelerator — anything that is not a Debugger command — the Debugger brings you into its evaluation environment, where you can evaluate Lisp expressions. See the section "Evaluating a Form in the Debugger".

Because they are implemented as CP commands, Debugger commands have positional arguments, keywords, and command *completion*, which allows you to enter a command without typing the whole command name. You can also edit a Debugger command with the input editor. For more information on positional arguments, keywords, and command completion: See the section "Communicating with Genera".

Debugger Command Accelerators

Most Debugger commands have key-binding accelerators. You can enter a command's accelerator instead of its full-form command name. Some commands have only one corresponding accelerator. For example, the accelerator `c-m-F` stands for:

:Show Function

Other commands, however, have two or more accelerators that correspond to different variations of the command. For example, the :Previous Frame command has five accelerators:

```
RETURN, c-P, m-P, c-m-P, c-m-U
```

In this case, each accelerator corresponds to a command/keyword combination. For example, `m-P` stands for:

:Previous Frame :Detailed Yes

and `c-m-P` stands for:

:Previous Frame :Internal Yes

Where applicable, accelerators take a numeric argument to complete the command successfully. For example, if you type the :Show Backtrace command, you can specify how many stack frames to display with the :Nframes keyword — :Nframes 2, 3, 4, and so on. However, if you enter the command accelerator, `c-B`, you can specify how many frames to display by giving a numeric argument. For example, `c-9 c-B` would display nine frames. Likewise, `c-1 c-5 c-B` would display 15 frames.

When you press an accelerator, the Debugger displays an italic message that defines what the accelerator stands for. It then executes the command. For example, when you press the `c-B` accelerator, you get this message:

→ `Control-B` *Show Backtrace :Nframes 10000 :Internal No :Detailed No*

Editing a Debugger Command

When you make a mistake while typing a Debugger command or change your mind about entering the command, you have two choices:

Press ABORT and begin again.

Edit your input.

The *input editor* allows you to type, display, and edit a Debugger command. With the input editor, you can edit all Debugger command components — command name, positional arguments, and keywords — before entering the command.

For more information on the input editor: See the section "The Input Editor Program Interface".

The input editor is also used to edit a form in the Debugger's evaluation environment. For more information on the Debugger's evaluation environment: See the section "Evaluating a Form in the Debugger".

Entering a Debugger Command with the Mouse

You can use the mouse to enter a Debugger command. This is accomplished by simply pointing the mouse at a Debugger command previously displayed in the screen output and clicking on that command. See the section "Using the Mouse in the Debugger".

Getting Help for Debugger Commands

The Debugger offers you online help. Pressing the HELP key inside the Debugger displays several help options for you to choose:

- c-HELP displays documentation about all Debugger commands. This documentation consists of brief command descriptions and available key-binding accelerators.

- The ABORT key takes you out of the Debugger. (You can enter the :Abort command or press c-Z instead of pressing ABORT.)

- c-m-W brings you into the Window Debugger. (You can enter the :Window Debugger command instead of pressing c-m-W.)

The REFRESH key, the :Show Frame command, or the :Show Frame command accelerator c-L clears the screen, then redisplays the error message for the current stack frame.

You can also ask for help with keywords. If you do not remember what keywords are available for the command you are entering, press the HELP key after you receive the keywords prompt. The Debugger displays a list of keywords for that command. For example:

```
→ :Previous Frame (keywords) HELP
You are being asked to enter a keyword argument

These are the possible keyword arguments:
:Detailed           Show locals and disassembled code
:Internal           Show internal interpreter frames
:Nframes            Move this many frames
:To Interesting     Move out to an interesting frame
```

Proceeding and Restarting in the Debugger

Upon entering the Debugger, you might not want to use Debugger commands. Instead, you might want, for example, to continue program execution, leave the Debugger, or return to a previous activity. These alternatives are called *proceeding* and *restarting* in the Debugger. Proceeding means to continue execution from the point where the error occurred. Restarting means to return to a prior activity, such as the Lisp Listener or Zmail.

Proceeding and restarting are implemented through a displayed list of possible actions for you to take. These actions are called *proceed* and *restart options*.

Using Debugger Proceed and Restart Options

Whenever you enter the Debugger, either for the first time or recursively, the Debugger displays a list of possible actions for you to take. These actions, called *proceed* and *restart options*, allow you to proceed from the error (continue program execution), leave the Debugger, restart (return to) a previous activity, or take some other action.

A list of proceed and restart options might look like this:

```
s-A, RESUME:   Supply a value to use this time as the value of FOO
s-B, s-sh-C:   Supply a value to store permanently as the value of FOO
s-C:           Retry the SYMEVAL instruction
s-D, ABORT:    Debugger command level 1
s-E:           Return to Lisp Top Level in Dynamic Lisp Listener 1
```

You can select one of these actions by pressing the keys that appear in the left-hand margin or by selecting an option with the mouse. All of these options are bound to the SUPER key.

Proceed and restart options are assigned to internal proceed handlers or restart handlers respectively. A proceed handler allows you to proceed from the error — continuing execution from the point where the error occurred. For example, you can assign a correct value to an unbound variable then continue execution. A restart handler allows you to unwind the stack — the series of calls that led to the error — and return to a previous system level prior to the error. For example, you can return to a previous Debugger invocation, Zmail, or Zmacs, or you can leave the Debugger and return to the *top level* activity, such as the Dynamic Lisp Listener, as shown above.

Using ABORT and RESUME in the Debugger

Debugger proceed and restart options are listed in order from the most recent handler that was called to the least recent, oldest handler that was called. The RESUME key is always assigned to the innermost proceed handler or the innermost restart handler if there are no proceed handlers. The ABORT key is always assigned to the innermost restart handler. Pressing the ABORT key usually brings you back to the next previous top-level process in which you were working before the error occurred.

In general, therefore, whenever you want to proceed from the error, press RESUME. Whenever you want to restart the previous activity, press ABORT.

The exact way RESUME works depends on the kind of error that happened. For some errors, there is no standard way to proceed, and the RESUME option just tells you so and returns to the Debugger's command level. For the very common "unbound variable" error, it requests that you supply the Lisp object that should be used in place of the (nonexistent) value of the symbol. For unbound-variable or undefined-function errors, you can also just type Lisp forms to set the variable or define the function, and then press RESUME; execution proceeds after the Debugger asks you to confirm that the new value is acceptable.

The ABORT key, of course, is used in general to exit from the Debugger. See the section "Exiting The Debugger".

Supplying a Value to Store Permanently

The value you supply with the RESUME proceed option provides a replacement value but does not change the value of the Lisp object permanently. If you want to change the value permanently, use the proceed option s-sh-C, which instructs you to supply a value to store permanently. This option is similar to RESUME, except s-sh-C actually sets a variable or defines a function and stores the new value so that the error does not happen again.

Supplying a Missing Package Prefix

The proceed option c-sh-P is only available for such errors as an unbound variable or undefined function when there is a variable or function in another package that has the same name. It permits easy recovery when you forget to supply a package prefix.

Evaluating a Form in the Debugger

You can evaluate a form in the Debugger as easily as you can evaluate a form in a Dynamic Lisp Listener. Evaluation in the Debugger is useful because the Debugger evaluates a form in the context of the function that got the error. All bindings that were in effect at the time the error occurred are in effect when your form is evaluated. You can also evaluate a form using the lexical context of the current frame. For example, you can see the values of lexical variables within LET and LOOP operations. Lexical variables are local variables created temporarily; they exist only for the duration of the lexical operation.

To evaluate a form in the Debugger, simply press a key that is not a command — a character other than a colon or command accelerator key. (As you recall, a full-form Debugger command must begin with a colon.) To evaluate a form, you can type, for example, an open parenthesis. The Debugger gives you the following evaluation prompt:

Eval (program):

This *Eval (program):* prompt indicates you are evaluating a form using the *lexical*, user-*program* context of the current frame. This means you can see the values of Lisp objects, including local variables, at the place where your program execution suspends.

The evaluation prompt comes up the moment you type a non-command character. Your character is immediately placed to the right of the prompt. For example, suppose you type an open parenthesis at the Debugger's right-arrow prompt. This is what happens the moment you type the character:

```
→ Eval (program): (
```

After it evaluates a form, the Debugger prompts again with the right arrow. If, while typing the form, you change your mind and want to get back to the Debugger's right-arrow prompt, press ABORT. Deleting all the characters in the form also brings you back to the Debugger prompt.

The Debugger's evaluation environment is actually a read-eval-print loop that uses the context of the function that received the error. Like a Dynamic Lisp Listener read-eval-print loop, the Debugger's evaluation environment maintains the values of +, *, and related variables.

If a complex error occurs in the evaluation of the Lisp expression, you are brought into a second Debugger looking at the new error, unless you have specified that your program handle that error. The Debugger prompts with two arrows (→→) to show that you are inside two Debuggers. You can get back to the first Debugger by pressing the ABORT key. However, if the error is not complex, the abort is done automatically and the original error message is reprinted. See the section "Using Recursive Debugger Invocations".

Various Debugger commands ask for Lisp objects, such as an object to return or the name of a catch-tag. Whenever it requests a Lisp object, it expects you to type in a form; it will evaluate what you type in. This provides greater generality, since there are objects to which you might want to refer that cannot be typed, such as arrays. If the form you type is not complex (not just a constant form), the Debugger shows you the result of the evaluation and asks you if it is what you intended. It expects a Y or N answer. (See the function **zl:y-or-n-p**.) If you answer negatively it asks you for another form. To exit the command, just press ABORT.

Besides the Debugger's lexical, user-program evaluation environment, the Debugger also has a *dynamic* evaluation environment, created specifically for the task of debugging the debugger. Unless you have to redesign or debug the Symbolics Debugger — an extremely unlikely prospect — *do not* use this evaluation environment. It is used exclusively by Symbolics software development personnel. The prompt for the dynamic evaluation environment is:

Eval (debugger):

If you accidentally bring up this prompt, you can change the environment and bring up the *Eval (program):* prompt by entering the :Use Lexical Environment command or by pressing `c-X I`, which toggles between the two environments.

The current evaluation environment is established by the previous environment you chose. Therefore, once you're in the lexical program environment, you will stay there until you explicitly enter the :User Dynamic Environment command or press `c-X I`.

For more information: See the section "Use Lexical Environment Command". Also: See the section "Use Dynamic Environment Command".

Editing a Form in the Debugger

When you make a mistake while typing a form in the Debugger or change your mind about entering the form, you can do one of two things:

Press `ABORT` and begin again.

Edit your input.

The *input editor* allows you to type, display, and edit a form in the Debugger's evaluation environment. The input editor is also used for input in Debugger commands and a Dynamic Lisp Listener command processor and read-eval-print loop. For information about editing a Debugger command: See the section "Editing a Debugger Command". For more information on the input editor: See the section "The Input Editor Program Interface".

Rebound Variable Bindings During Evaluation

When the Debugger evaluates a form, the variable bindings at the point of error are in effect with the following exceptions:

- ***terminal-io*** is rebound to the stream the Debugger is using. **dbg:old-terminal-io** is bound to the value that ***terminal-io*** had at the point of error.

- ***standard-input*** and ***standard-output*** are rebound to be synonymous with ***terminal-io***; their old bindings are saved in **dbg:old-standard-input** and **dbg:old-standard-output**.

- ***query-io***, ***debug-io***, and ***error-output*** are rebound to be synonymous with ***terminal-io***; their old bindings are not directly accessible.

- **+** and ***** are rebound to the Debugger's previous form and previous value. When the Debugger is first entered, **+** is the last form typed, which is typically the one that caused error, and ***** is the value of the *previous* form. **++**, **+++**, ******, *******, **-**, and **zl:/** are treated in an analogous fashion. See the section "The Lisp Top Level". When the Debugger is exited, all of these variables are restored to their original values; the interactions with the Debugger's read-eval-print loop do not affect the interactions with the top-level Lisp read-eval-print loop.

- **sys:rubout-handler** and **zl:read-preserve-delimiters** are rebound to **nil**, in case the error occurred while in the input editor or the reader.

- **evalhook** is rebound to **nil**, turning off the **zl:step** facility if it was in use when the error occurred. See the section "A Hook Into the Evaluator".

- **dbg:*bound-handlers*** and **dbg:*default-handlers*** are rebound to **nil**, preventing conditions signalled by the form the Debugger is evaluating from reaching condition handlers in the program being debugged. This prevents you from accidentally being thrown out of the Debugger.

- ***print-base***, **zl-user:*read-base***, ***package***, and **zl-user:*read-default-float-format*** are checked to insure that they contain legal values. If not, they are set to their standard values.

Note that the variable bindings are those in effect in the current frame being examined, unless you are not inheriting the lexical environment, in which case the bindings are those in effect at the point of error.

Using Recursive Debugger Invocations

Whenever you cause an error from within the Debugger, or call the Debugger explicitly from within the Debugger, you are brought into another Debugger.

For example, suppose you used an unbound variable in the Dynamic Lisp Listener. The Debugger is invoked. Then suppose, inside this first Debugger, you reference an undefined function. You are brought into a second Debugger. Then suppose you reference a function that contains a **zl:dbg** function. You are brought into a third Debugger.

In the scenario described above, the three Debugger calls are *recursive Debugger invocations*, where the Debugger causes itself to be called. Each Debugger call is known as a Debugger *command level*. The first call is the first level, the second call is the second level, and the third call is the third level. You can simply refer to the first Debugger, second Debugger, and third Debugger.

If you were to get a backtrace at the third Debugger, you would see that each call to the Debugger appears as a separate stack frame. Like other stack frames, you can unwind the stack — usually with the ABORT key — and thereby have each Debugger return to the previous Debugger. The term *unwind* means to return the function in the current frame to the function in the previous frame. Remember: In the third Debugger, you have three *active* Debuggers. They have been called but have not yet returned.

The Debugger command prompt lets you know which Debugger you are in at any given time. For example, three right arrows (→→→) indicate you are in the third Debugger. Two right arrows (→→) indicate you are in the second. One arrow, of course, indicates you are at the first.

Using the same example, suppose, in a Dynamic Lisp Listener, you reference an unbound variable, **foo**:

```
Trap: The variable FOO is unbound.

SI:*EVAL:
      Arg 0 (SYS:FORM): FOO
      Arg 1 (SI:ENV): NIL
      --Defaulted args:--
      Arg 2 (SI:HOOK): NIL
s-A, RESUME:     Supply a value to use this time as the value of FOO
s-B, s-sh-C:     Supply a value to store permanently as the value of FOO
s-C:             Retry the SYMEVAL instruction
s-D, ABORT:      Return to Lisp Top Level in Dynamic Lisp Listener 1
s-E:             Restart process Dynamic Lisp Listener 1
→
```

Then suppose, within the Debugger, you reference an undefined function, **glitch**:

```
Trap: The function GLITCH is undefined.

SI:*EVAL:
      Arg 0 (SYS:FORM): (GLITCH)
      Arg 1 (SI:ENV): NIL
      --Defaulted args:--
      Arg 2 (SI:HOOK): NIL
 Debugger was entered because an error occurred while evaluating a form in the debugger
s-A, RESUME:     Supply a value to use this time as the definition of GLITCH
s-B, s-sh-C:     Supply a value to store permanently as the definition of GLITCH
s-C:             Retry the FSYMEVAL instruction
s-D, ABORT:      Debugger command level 1
s-E:             Return to Lisp Top Level in Dynamic Lisp Listener 1
s-F:             Restart process Dynamic Lisp Listener 1
→→
```

In the example shown above, notice the two right arrows, which indicate entry to the second Debugger. Notice also the restart option, ABORT, which allows you to return to the first Debugger. Suppose now, within the second Debugger, you reference **zl:dbg**:

```
Break:

SI:*EVAL:
      Arg 0 (SYS:FORM): (ZL:DBG)
      Arg 1 (SI:ENV): NIL
      --Defaulted args:--
      Arg 2 (SI:HOOK): NIL
s-A, RESUME:      Proceed without any special action
s-B, ABORT:       Debugger level 2
s-C:              Debugger command level 1
s-D,              Return to Lisp Top level in Dynamic Lisp Listener 1
s-E:              Restart process Dynamic Lisp Listener 1
→→→
```

Now notice the three right arrows, which indicate entry to the third Debugger. Notice also the two restart options, ABORT and s-C, which allow you to return to the second Debugger and the first Debugger respectively.

Pressing the ABORT key is the fundamental way of leaving the current Debugger and returning to the previous Debugger level. If you have amassed many Debugger invocations and want to leave the Debugger entirely and return to top level immediately — in this case Dynamic Lisp Listener 1 — press m-ABORT, which keeps unwinding the stack until you reach top level. m-ABORT always gets you back to top level.

The following example shows what happens when you keep pressing ABORT, beginning at the third Debugger:

```
→→→ Abort Abort
Debugger command level 2
Back to Trap: The function GLITCH is undefined.
→→ Abort Abort
Debugger command level 1
Back to Trap: The variable FOO is unbound.
→ Abort Abort
Return to Lisp Top Level in Dynamic Lisp Listener 1
Back to Lisp Top Level in Dynamic Lisp Listener 1.

Command:
```

Using the Mouse in the Debugger

Like most other screen output generated in the Genera software environment, Debugger output is mouse sensitive. You can perform some useful debugging operations simply by clicking on output produced by Debugger commands. For example, you can perform the following operations:

- Execute a Debugger command by clicking on any command name that is already displayed on the screen as a result of the command's prior use.

- Set the current stack frame by clicking on a frame's function name displayed in backtrace output.
- Evaluate a form by entering the :Show Source Code command, pointing the mouse at a code fragment in the source code output, and pressing m-Middle.
- Set a Debugger breakpoint on a compiled function by entering the :Show Compiled Code command, pointing the mouse at a PC (program counter) line in the disassembled code output, and pressing c-m-Left.
- Set a Debugger breakpoint on a form in the source code by entering the :Show Source Code command, pointing the mouse at a code fragment in the source code output, and pressing c-m-Left.
- Clear a Debugger breakpoint on a compiled function by entering the :Show Compiled Code command, pointing the mouse at a PC line in the disassembled code output, and pressing c-m-Middle.
- Clear a Debugger breakpoint on a form in the source code by entering the :Show Source Code command, pointing the mouse at a code fragment in the source code output, and pressing c-m-Middle.
- Monitor the access of a variable or other location by pointing the mouse at a locative, structure slot, or instance variable and pressing c-m-sh-Left. (When a program or process accesses the monitored location, a Debugger trap is signalled.)
- Unmonitor a variable or other location by pointing the mouse at a locative, structure slot, or instance variable and pressing c-m-sh-Middle. (When you stop monitoring the access of a location, the Debugger trap is no longer signalled.)
- Edit a function in a Zmacs editor window by pointing the mouse at a function's stack frame and pressing m-Left.
- Activate a proceed or restart option by clicking on one.
- Perform a **describe** function on a Lisp object by pointing the mouse at any object and pressing Middle.

Suggested mouse operations are listed in the individual descriptions of some Debugger commands later in this chapter. See the section "Debugger Command Descriptions". Since so much of the Debugger output is mouse sensitive, the documentation lists only the most useful mouse operations. However, you are encouraged to experiment with the mouse while using the Debugger. You most likely will discover some other mouse or mouse/keyboard capabilities that are particularly suited to your personal debugging style.

Of course, you can perform virtually all of the suggested mouse operations listed in the documentation via momentary menus. As with all other screen output in Genera software environment, you can also use menus and submenus to perform a huge variety of system operations on Debugger output. To perform system or Debugger operations via menus, just point the mouse at the desired piece of Debugger output — a form, function, argument, flavor, instance, locative, or whatever — and click Right.

Creating Debugger Proceed Menus

Debugger pop up proceed menus are an alternative interface to the regular Debugger. When an error occurs, a menu pops up, enabling the user to select a retry option. Debugger proceed menus are perfect for situations where there is an abnormal but expected error with a possible clean recovery. An example is programs that perform long file operations where there is a good possibility of a network break.

The following is a list of standard errors in the Genera system that cause a Debugger proceed menu to appear in case of an error:

fs:file-operation-failure
fs:unknown-pathname-host
fs:host-not-accessible-for-file
fs:host-not-available
sys:host-not-responding
sys:unknown-host-name
tape:mount-error

If you want to disable Debugger menus from appearing, set or bind the variable **dbg:*disable-menu-proceeding*** to **t**.

dbg:*disable-menu-proceeding* *Variable*

Disables Debugger menu proceeding. When set or bound to **t**, it forces all error conditions to enter the standard Debugger.

If you have a condition that you want to add to the list of standard errors that causes a pop up Debugger menu to appear, use the macro **dbg:with-extra-debugger-menu-conditions**.

dbg:with-extra-debugger-menu-conditions *(conditions)* &body *body* *Macro*

Given a set of *conditions*, executes the *body* with the *conditions* added to the standard list of conditions which cause a Debugger menu to appear, rather than entering the Debugger.

This macro is useful for error conditions which are abnormal but expected, and for which there is a possible clean recovery. For example, if you have a program that is performing long file operations, and you expect that it may run into a network break, you can use **dbg:with-extra-debugger-menu-conditions** to provide a pop-up menu of options when it runs into the network break, rather than putting you into the standard Debugger.

conditions is a list of flavors.

body is the code that may encounter the conditions.

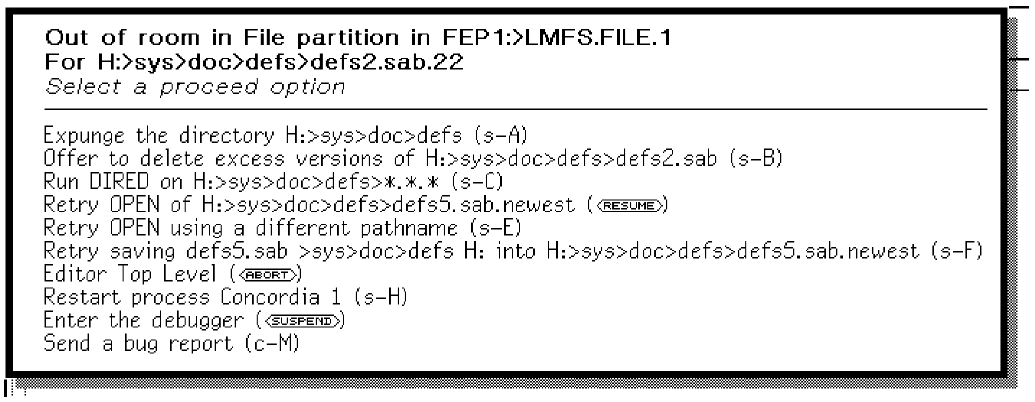
Here is an example of **dbg:with-extra-debugger-menu-conditions** used with code designed to save the results of a computation on a file. **dbg:with-extra-debugger-menu-conditions** provides for a Debugger menu interface in case any kind of file or network error occurs.

```
(dbg:with-extra-debugger-menu-conditions (fs:file-error sys:network-error)
  (catch-error-restart-with-form ((fs:file-error sys:network-error)
                                  "Skip saving file ~A." name)
      (abort-current-command)
    (error-restart ((fs:file-error sys:network-error)
                    "Retry saving file ~A." name)
      (with-open-file (stream name :direction :output)
        ... <code to output data to the file> ... )
      )))
```

To disable the Debugger menu from appearing, you can set or bind the variable **dbg:*disable-menu-proceeding*** to **t**.

Figure

! shows an example of a Debugger menu which occurred when a user tried to save a file on a disk when there was not enough room. This menu is unrelated to the code example above.



Out of room in File partition in FEP1:>LMFS.FILE.1
For H:>sys>doc>defs>defs2.sab.22
Select a proceed option

```
Expunge the directory H:>sys>doc>defs (s-A)
Offer to delete excess versions of H:>sys>doc>defs>defs2.sab (s-B)
Run DIRED on H:>sys>doc>defs>*.*.* (s-C)
Retry OPEN of H:>sys>doc>defs>defs5.sab.newest (RESUME)
Retry OPEN using a different pathname (s-E)
Retry saving defs5.sab >sys>doc>defs H: into H:>sys>doc>defs>defs5.sab.newest (s-F)
Editor Top Level (ABORT)
Restart process Concordia 1 (s-H)
Enter the debugger (SUSPEND)
Send a bug report (c-M)
```

Figure 6. The Debugger Menu

Debugger Command Descriptions

This section provides descriptions for all Debugger commands. These commands fall into eight categories according to their functions:

- Commands for viewing a stack frame
- Commands for stack motion
- Commands for general information display
- Commands to continue execution
- Trap commands
- Commands for breakpoints and single stepping
- Commands for system transfer
- Miscellaneous commands

Debugger commands are implemented as Command Processor (CP) commands. There are many Debugger commands that share the global command table with CP commands. Therefore, you can enter these commands in the CP as well as the Debugger. They are:

- :Clear All Breakpoints
- :Clear Breakpoint
- :Disable Condition Tracing
- :Edit Function
- :Enable Condition Tracing
- :Monitor Variable
- :Set Breakpoint
- :Set Stack Size
- :Show Breakpoints
- :Show Compiled Code
- :Show Monitored Locations
- :Show Source Code
- :Unmonitor Variable

Note, however, that you must precede every command entered in the Debugger with a colon; for example, you must type :Set Breakpoint in the Debugger.

In the sections that follow, Debugger commands are presented in alphabetical order within their logical groups. Each command presentation contains a command format line, a brief command description, and lists of positional arguments, keywords, and useful mouse operations, if any. Key-binding accelerators, if any, appear against the right margin on the command format line. If a command has two or more accelerators, then its accelerators are listed separately with corresponding command/keyword definitions.

Command descriptions use the terms *default* and *mentioned default*. A *default* is the result of entering a Debugger command without a keyword and/or positional argument. A default also means the result of entering a positional argument without a modifier. A *mentioned default* is the result of entering a keyword without a keyword modifier, such as Yes or No. In other words, once you type in the keyword, the Debugger *mentions* the consequences of pressing RETURN without a keyword modifier.

Debugger Commands for Viewing a Stack Frame

The Debugger provides commands for displaying information about the current stack frame. Information that you can display includes, for example, argument values, local variable values, disassembled code, source code, and **&rest** arguments. These commands, in alphabetical order, are:

- :Show Arglist (c-X c-A)
- :Show Argument (c-m-A)
- :Show Compiled Code (c-X D)
- :Show Frame (REFRESH, c-L, m-L)
- :Show Function (c-m-F)
- :Show Local (c-m-L)
- :Show Rest Argument
- :Show Source Code (c-X c-D)
- :Show Stack
- :Show Value (c-m-V)

All of these commands operate in the context of the *current stack frame*. The Debugger knows about the current frame at any given time, and it uses the current frame environment to perform operations according to the suspended state of your program. For example, it evaluates forms in the lexical context of the function suspended in the current frame.

Initially, the current stack frame is the one that signalled the error — either the one that got the microcode-detected error or the one that called **ferror**, **error**, or a related function. The current frame can change, depending on which Debugger operations you perform.

When the Debugger is invoked, it shows you the current frame in the following format:

foo:
```
Arg 0 (X): 13
Arg 1 (Y): 1
```

The Debugger displays the name of the function in the current frame, then lists the numbers, names, and values of all arguments in the current frame. In the case shown above, **foo** was called with two arguments, whose numbers are 0 and 1 and whose names in the Lisp source code are **x** and **y**. The current values of **x** and **y** are 13 and 1 respectively. Numbering of arguments begins with 0. Therefore, argument 0 refers to the first argument, argument 1 refers to the second argument, and so on.

Show Arglist Command

:Show Arglist c-X c-A

Displays the argument list for the function in the current frame. When you enter this command, the Debugger replies:

```
     The argument list for (function-name) is (argument-names)
```

The *function-name* is the name of the function in the current frame — the name of the function that appears when the Debugger is invoked. It is also the name of the function that would appear at the top of the stack if you were to perform a backtrace.

The *function-name* is the name of the function in the current frame, that is the name of the function that appears when the Debugger is invoked. This is also the name of the function that appears at the top of the stack if you perform a backtrace.

```
     Debug 1> :set-current-frame 5
     => 5: (#<function:91e933> # # # 4 ...)
     Debug 1> :show-arglist
     Arguments for frame 5
     Argument (0) = (1)
     Argument (1) = (1 2)
     Argument (2) = (1 (2 3))
     Argument (3) = 4
     Argument (4) = 5
```

Show Argument Command

:Show Argument *argument* `c-m-A`

Displays the value of one or all arguments in the current frame. You can also use the Lisp function **(dbg:arg** *number***)** where *number* specifies the number of the argument you want to display. Numbering begins with 0. For example, **(dbg:arg 3)** displays the fourth argument. A numeric argument given with this command's accelerator also specifies the number of the argument you want to display; for example, `c-m-3 c-m-A` displays the fourth argument. To change the value of an argument, **setf** on **(dbg:arg** *number***)**.

When you ask to see all arguments — the default for this command — the Debugger displays the arguments in the same way it would display them upon entry to the Debugger. It displays the name of the function in the current frame, then lists the numbers, names, and values of all arguments in that function. When you specify an argument number, the Debugger displays only the value of that argument.

When you are using the lexical context of the current frame, you can evaluate an argument by typing in its name (or clicking on its name using the mouse) in the Debugger's evaluation environment.

The :Show Argument command leaves * set to the value of the argument so you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack so you can change that argument by calling **setf** on the locative.

argument {*number*, All} The *number* is an integer that specifies which argument you want to display in the current frame. All displays all arguments in the current frame. (Default is All.)

Displays the value of one of the arguments in the current frame. Numbering begins with 0. A numeric argument given with this command's accelerator specifies the number of the argument you want to display.

```
Debug 1> :set-current-frame 5
=> 5: (#<function:91e933> # # # 4 ...)
Debug 1> :show-argument 2
=> 5: Argument (2):
(1 (2 3))
```

Show Compiled Code Command

Show Compiled Code *compiled-function-spec from-pc to-pc keywords* `c-X D`

Displays the disassembled code for a function. When you enter this command and specify a compiled-function-spec, the Debugger displays this message:

Disassembled code for (*function*):

where *function* is the name of the compiled function for which you want to see disassembled code. Immediately under this message, the Debugger lists the disassembled code instructions for this function. Each instruction — `PUSH`, `CALL`, `BRANCH`, and so on — is listed on its own line, numbered by the PC (program counter). PCs are numbered in octal (base 8), and numbering begins with 0.

compiled-function-spec
The name of a compiled function for which you want to see disassembled code. (Default is the function in the current frame.)

from-pc The number of the PC at which you want to begin seeing disassembled code. (Default displays all disassembled code.)

to-pc The number of the PC at which you want to stop seeing disassembled code. (Default displays disassembled code from PC 0, or from the number specified in *from-pc*, to the last PC in the disassembled code.)

keywords :More Processing, :Output Destination

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** pro-

	cessing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Suggested mouse operations

- To use this command with the mouse: Type in the Show Compiled Code command. When the Debugger asks you for a compiled-function-spec, point the mouse at the name of a compiled function previously displayed in the output of another command, such as Show Backtrace or Next Frame, and click Left. (You can do this only when your previous command output includes the name of a compiled function.)

- To set a breakpoint: Point the mouse at a PC in the disassembled code and press c-m-Left.

- To clear a breakpoint: Point the mouse at a PC in the disassembled code and press c-m-Middle.

Displays the disassembled code for the function associated with the current frame.

Show Frame Command

:Show Frame *keywords* REFRESH, c-L, m-L

Displays information about the current frame. (Default redisplays the error message for the current frame then lists the name of the function and its arguments in the current frame.)

keywords	:Clear Window, :Detailed
:Clear Window	{Yes, No} Clears the screen and redisplays at the top of the screen the error message for the current frame. The name of the function and its arguments in the current frame are also displayed. (Default is No. Mentioned default is Yes.)
:Detailed	{Yes, No} Redisplays the error message for the current frame then displays detailed information, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

Key-binding accelerators

`REFRESH, c-L`	:Show Frame :Clear Window Yes
`m-L`	:Show Frame :Clear Window Yes :Detailed Yes

Displays information about the current frame.

```
Debug 1> :set-current-frame 5
=> 5: (#<function:91e933> # # # 4 ...)
Debug 1> :show-frame
=> 5: (#<function:91e933> (1) (1 2) (1 (2 3)) 4 5)
```

Show Function Command

:Show Function `c-m-F`

Displays the name of the function in the current frame. You can also use the Lisp function **(dbg:fun)**. The Show Function command leaves * set to the value of the function so that you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the function so that you can change it by calling **setf** on the locative.

Displays the name of the function in the current frame.

```
Debug 1> :set-current-frame 5
=> 5: (#<function:91e933> # # # 4 ...)
Debug 1> :show-function
=> 5: Function = #<function:91e933>
```

Show Local Command

:Show Local *local-variable* `c-m-L`

Displays the value of one or all local variables for the function in the current frame. When you enter this command, the names of local variables and their values are listed in a sequence: Local 0, Local 1, Local 2, and so on. In this list, locals are numbered in decimal (base 10), and numbering begins with 0.

You can also use the Lisp function **(dbg:loc** *number***)** where *number* specifies which local variable you want to display. For example, **(dbg:loc 3)** displays the fourth local variable. A numeric argument given with this command's accelerator also specifies which local variable you want to display; for example, `c-m-3 c-m-L` displays the fourth local variable. To change the value of a local variable, use the **setf** function with **(dbg:loc** *number***)**.

When you are using the lexical context of the current frame, you can evaluate a local variable by typing its name (or clicking on its name using the mouse) in the Debugger's evaluation environment.

The :Show Local command leaves * set to the value of the local variable so you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the local variable on the stack so you can change that argument by calling **setf** on the locative.

local-variable	{*number*, All} The *number* is an integer that specifies which local variable you want to see in the current frame. All displays all local variables in the current frame. (Default is All.)

Displays the value of one of the local variables for the function in the current frame. When you enter this command, the local variables and their values are listed in a sequence: Local 0, Local 1, Local 2, and so on. In this list, locals are numbered in decimal (base 10), and numbering begins with 0.

Show Rest Argument Command

:Show Rest Argument

Displays the **&rest** argument, if there is one, and formats it neatly. :Show Rest Argument sets the value of *.

Show Source Code Command

:Show Source Code *compiled-function-spec keywords* `c-X c-D`

Displays the source code for a function. This command works only when your code resides in an editor buffer. The output is mouse sensitive only when the function is compiled with source locators. When you specify a compiled function for which you want to see source code — for example, **myfunction** — the Debugger displays the source code for **myfunction** beneath the following message:

Source code for `MYFUNCTION:`

If **myfunction** were not compiled with source locators, the Debugger would still display the source code, but the output would not be mouse sensitive. The Debugger would display the source code only after giving you this message:

```
Function MYFUNCTION has no source locators; the code will not be sensitive.
```

compiled-function-spec	The name of a compiled function for which you want to see source code. (Default is the function in the current frame.)
keywords	:More Processing, :Output Destination

:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not

subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Suggested mouse operations

When a function has been compiled using source locators — mapping source code to PCs via the editor's `c-m-sh-C` command — you can perform the following mouse operations:

- To use this command with the mouse: Type in the :Show Source Code command. When the Debugger asks you for a compiled-function-spec, point the mouse at the name of a compiled function previously displayed in the output of another command, such as :Show Backtrace, and click Left.

- To set a breakpoint: Point the mouse at a form (a code fragment) in the displayed source code and press `c-m`-Left.

- To clear a breakpoint: Point the mouse at a form (a code fragment) in the displayed source code and press `c-m`-Middle.

- To evaluate a code fragment: Point the mouse at a form in the displayed source code and press `m`-Middle.

Show Stack Command

:Show Stack

Displays all of the local-variable and temporary stack slots in the current frame. This command is very similar to :Show Local, except that in addition to local-variable slots, :Show Stack displays stack slots that do not necessarily correspond to named local variables. Therefore, :Show Stack gives you more information than does :Show Local. The output for this command is displayed the way :Show Local output is displayed; that is, locals and their values are listed in sequence: Local 0, Local 1, Local 2, and so on. In this list, stack slots are numbered in decimal (base 10), and numbering begins with 0.

Show Value Command

:Show Value *value* `c-m-V`

Displays one or all values being returned from the function that is being returned. If the frame is not in the process of returning values, the Debugger tells you:

```
No values are being returned now
```

:Show Value is useful only when you are using a trap on exit or looking at a frame that is about to return. See the section "Set Trap On Exit Command".

You can also use the Lisp function **(dbg:val** *number***)** where *number* specifies which value to display. Numbering begins with 0. For example, **(dbg:val 3)** displays the fourth value. A numeric value used with this command's accelerator also specifies which value to display; for example, `c-m-3 c-m-V` displays the fourth value. To change a particular value being returned from a frame, use **setf** on **(dbg:val** *number***)**.

The :Show Value command leaves * set to the value of the argument, so you can use the read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack so you can change that argument by calling **setf** on the locative.

value	{*number*} The *number* is an integer that specifies which value to display.

Debugger Commands for Stack Motion

The Debugger provides commands that allow you to move up and down the stack. The term *move* in the context of these commands means to make another frame the current frame. For example, moving to the top of the stack makes the most recent frame — the frame where the error occurred — the current frame.

Moving down the stack takes you back in time toward the oldest, least-recent frame. Moving up the stack takes you forward in time toward the newest, most-recent frame, which is usually the call to the Debugger itself.

Stack motion commands not only traverse the stack, but they also display information about the frame to which you move. Most of these commands can optionally display local variables, disassembled code, and internal interpreter frames.

The motion commands, in alphabetical order, are:

- :Bottom Of Stack (`m->`)
- :Find Frame (`c-S`)
- :Next Frame (`LINE, c-N, m-N, c-m-N`)
- :Previous Frame (`RETURN, c-P, m-P, c-m-P, c-m-U`)
- :Set Current Frame
- :Top Of Stack (`m-<`)

Bottom Of Stack Command

:Bottom Of Stack *keyword* `m->`

Moves to the bottom of the stack, displays the least recent frame, and makes that frame current. When you enter this command, the Debugger displays the name of the function at the bottom of the stack, followed by its arguments.

keyword	:Detailed
:Detailed	{Yes, No} Displays detailed information about the frame at the bottom of the stack, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

Moves to the bottom of the stack, displays the least recent frame, and makes that frame current. When you enter this command, the Debugger displays the name of the function at the bottom of the stack, followed by its arguments.

```
Debug 1> :bottom-of-stack
=> 14: (SYSTEM::APPLICATION-TOP-LEVEL #)
```

Find Frame Command

:Find Frame *string keywords* c-S

Searches the stack for a frame's function name that contains a specified string and makes that frame current. When you enter this command, the Debugger displays the name of the function in the specified frame, followed by its arguments.

string	A *string* that can be part or all of a function name.
keywords	:Detailed, :Invisible, :Reverse
:Detailed	{Yes, No} Displays detailed information about the specified frame, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)
:Invisible	{Yes, No} Yes searches all frames, visible and invisible. (The default is No. Mentioned default is Yes.)
:Reverse	{Yes, No} Searches backwards, toward the most recent frame, for the specified frame. (Default is No. Mentioned default is Yes.)

Searches the stack for a frame's function name that contains a specified string and makes that frame current. When you enter this command, the Debugger displays the name of the function in the specified frame, followed by its arguments. Keywork @i(string) is a string that can be part or all of a function name.

Next Frame Command

:Next Frame *keywords* `LINE, c-N, m-N, c-m-N, m-sh-N`

Moves down one frame, to the next less-recent frame — the calling frame — displays information about that frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the next frame, followed by its arguments. A numeric argument given with this command's accelerators, as well as the :Nframes keyword, specifies how many frames to move down; for example, `c-3 c-N` moves down three frames.

keywords	:Detailed, :Internal, :Invisible, :Nframes
:Detailed	{Yes, No} Displays detailed information about the next frame, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)
:Internal	{Yes, No} Displays internal interpreter frames in the next frame. Ordinarily, when running interpreted code, the Debugger tries to skip over frames that belong to functions of the interpreter, such as **si:*eval**, **prog**, and **cond**, and only show "interesting" functions. (Default is No. Mentioned default is Yes.)
:Invisible	{Yes, No} Yes allows selecting invisible frames. No skips over them. (Default is No. Mentioned default is Yes.)
:Nframes	{*number*} Specifies how many frames you want to move down. The *number* signifies that you want to move down to the *n*th frame from the current frame. (Default is 1.)

Key-binding accelerators

`LINE, c-N`	:Next Frame :Nframes 1
`m-N`	:Next Frame :Detailed Yes :Nframes 1
`m-sh-N`	:Next Frame :Detailed Yes :Nframes 1 :Invisible Yes
`c-N`	:Next Frame :Detailed Yes :Nframes 1 :Invisible No
`c-m-N`	:Next Frame :Internal Yes :Nframes 1

Moves down one frame to the next less recent frame, the calling frame. Displays information about that frame and makes is current.

```
    Debug 1> :next-frame
    => 6: (FIB1 1)
```

(:next-frame)
Moves down one frame, to the next less-recent frame or the calling frame, and displays information about that frame to makes it current. When you enter this command, the Debugger displays the name of the function in the next frame, followed by its arguments.

Previous Frame Command

:Previous Frame *keywords* `RETURN, c-P, m-P, c-m-P, c-m-U, m-sh-P`

Moves up one frame, to the next most-recent frame — the frame that the current frame called — displays information about that frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the previous frame, followed by its arguments. A numeric argument given with this command's accelerators, as well as the :Nframes keyword, specifies how many frames to move up; for example, `c-3 c-P` moves up three frames.

keywords	:Detailed, :Internal, :Invisible, :Nframes, :To Interesting
:Detailed	{Yes, No} Displays detailed information about the previous frame, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)
:Internal	{Yes, No} Displays internal interpreter frames in the previous frame. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as **si:*eval**, **prog**, and **cond**, and only show "interesting" functions. (Default is No. Mentioned default is Yes.)
:Invisible	{Yes, No} Yes allows selecting invisible frames. No skips over them. (Default is No. Mentioned default is Yes.)
:Nframes	{*number*} Specifies how many frames you want to move up. The *number* signifies that you want to move up to the *n*th frame from the current frame. (Default is 1.)
:To Interesting	{Yes, No} Moves to the next previous frame that is interesting (non-interpreter), skipping over interpreter frames. (Default is No. Mentioned default is Yes.)

Key-binding accelerators

RETURN, c-P	:Previous Frame :Nframes 1
m-P	:Previous Frame :Detailed Yes :Nframes 1
m-sh-P	:Previous Frame :Detailed Yes :Nframes 1 :Invisible Yes
c-P	:Previous Frame :Detailed Yes :Nframes 1 :Invisible No
c-m-P	:Previous Frame :Internal Yes :Nframes 1
c-m-U	:Previous Frame :To Interesting Yes

Moves up one frame to the next most recent frame, the frame that the current frame called. Displays that frame and makes it current.

```
Debug 1> :set-current-frame 6
:set-current-frame 6
=> 6: (FIB1 1)
Debug 1> :previous-frame
=> 5: (#<function:91e933> (1) (1 2) (1 (2 3)) 4 5)
```

:previous-frame
Moves up one frame, to the next most-recent frame or the frame that the current frame called, and displays information about that frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the previous frame, followed by its arguments.

Set Current Frame Command

:Set Current Frame *stack-frame*

Makes the stack frame that you specify with the mouse become the current frame.

stack-frame	A *stack frame* that you select with the mouse.

Suggested mouse operations

- To set the current frame: Display the stack with the :Show Backtrace command, point the mouse at the stack frame you want to make current, and click Left.

:set-current-frame *stack-frame*
This command set the current frame to be the frame numbered n. Frames are numbered from 0.

```
Debug 1> :set-current-frame 5
=> 5: (#<function:91e933> # # # 4 ...)
```

Top Of Stack Command

:Top Of Stack *keyword* m-<

Moves to the top of the stack — the frame where the error occurred — displays the most recent frame, and makes it current. When you enter this command, the Debugger displays the name of the function in the frame at the top stack, followed by its arguments.

keyword	:Detailed
:Detailed	{Yes, No} Displays detailed information about the frame at the top of the stack, including: Arguments and their values, local variables and their values, and disassembled code with an arrow pointing to the next instruction to be executed. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)

Moves to the top of the stack, displays the most recent frame, and makes that frame current. When you enter this command, the Debugger displays the name of the function at the top of the stack, followed by its arguments.

```
Debug 1> :top-of-stack
=> 0: (DEBUGGER::COMPUTE-FRAMES-IF-NECESSARY)
```

top-of-stack
Moves to the top of the stack, which is the frame where the error occurred, displays the most recent frame to make it current. When you enter this command, the Debugger displays the name of the function in the frame at the top stack, followed by its arguments.

Debugger Commands for General Information Display

The Debugger provides commands that allow you to examine the Lisp control stack and display general information about your program's execution as it relates to the error that triggered entry to the Debugger. Information that you display, for example, can be the value of *, special variable bindings, catch blocks, condition handlers, instructions, standard value warnings, proceed options, and so on.

The most powerful information-display command is :Show Backtrace, which displays the Lisp control stack. The stack keeps a record of all active functions. The term *active* refers to a function that has been called but has not yet returned. For example, if you call **foo** at Lisp's top level, and it calls **bar**, which in turn calls **baz**, and **baz** gets an error, then a *backtrace* displays this call history. Functions **foo**, **bar**, and **baz** appear on the stack because they have been called but have not yet returned. A backtrace, therefore, traces the execution of program functions and system functions back in time, and the Debugger displays the sequence of calls that led to the error.

The :Show Backtrace command can display a brief backtrace with only function names in a call history sequence, or it can display backtraces with more detailed information, such as arguments, local variables, disassembled code, and internal interpreter frames. Using the the **foo/bar/baz** example mentioned above, a brief backtrace of that call history might look like this:

```
BAZ ← BAR ← FOO ← EVAL ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

In the example shown above, the arrows indicate the direction of calling. See the section "Show Backtrace Command".

The general information display commands, in alphabetical order, are:

- :Analyze Frame (`c-m-Z`)
- :Describe Last (`c-m-D`)
- :Show Backtrace (`c-B`, `m-B`, `c-m-B`)
- :Show Bindings (`c-X B`)
- :Show Catch Blocks
- :Show Condition Handlers
- :Show Instruction (`c-m-I`)
- :Show Lexical Environment
- :Show Proceed Options
- :Show Special
- :Show Standard Value Warnings
- :Symeval In Last Instance (`c-X c-I`)
- :Use Dynamic Environment (`c-X I`)
- :Use Lexical Environment (`c-X I`)

Analyze Frame Command

:Analyze Frame `c-m-Z`

Analyzes the erroneous frame and locates the source code of the current error. Whenever your program blows up unexpectedly, for example, due to an incorrect argument value or undefined function, you can use the :Analyze Frame command to walk back up the stack and locate the origin of the error.

Specifically, the :Analyze Frame command can locate the source-code origin of these type of errors:

- Incorrect argument values
- Invalid or undefined functions
- Unclaimed messages
- Wrong number of arguments

If :Analyze Frame does not operate on a particular kind of error, the Debugger tells you:

There is nothing to analyze in this frame.

:Analyze Frame tells you the name of the function where the error occurred, moves to the previous frame, and examines the code in the previous frame. If it does not find the origin of the error in that frame, it keeps moving up the stack, examining code frame by frame. For each frame, the Debugger displays the name of the "bad argument" that received the error as well as the name of the function that passed the error — the calling function. It also highlights the bad argument and calling function in boldface type and displays the source code.

The last frame the Debugger displays is the frame that caused the error.

Suppose a bad argument, **foo**, was passed to a function, **myfunction** — the place where the error occurred — and **foo** originated from another function, **glitch**. The Debugger would display the source code of **myfunction** beneath the following message:

> *Error occurred in* **myfunction**:

Then the Debugger would tell you:

> *Probably bad argument* **foo**

followed by this message:

> *Called from* **glitch**:

The Debugger would then display the source code for **glitch**. If the bad argument, **foo**, had not originated from **glitch**, the Debugger would have kept crawling up the stack, and, for each frame, would have displayed the probable bad argument and the source code of the calling function.

Suppose you execute a function, **test**, without arguments, and **test** calls another function, **number-test**, which expects one argument, **n**. Via the :Analyze Frame command, the Debugger would display the following information:

> *Bad call ocurred in:*

```
(DEFUN TEST ()
  (NUMBER-TEST))
```

> *Correct arguments to* `NUMBER-TEST` *are* `(N)`

Describe Last Command

:Describe Last `c-m-D`

Executes the Lisp **describe** function on the most recently displayed value and leaves * set to that value.

Suggested mouse operations

- To perform a **describe** function on any Lisp object: Point the mouse at any object in the output and click Middle.

Show Backtrace Command

:Show Backtrace *keywords* c-B, m-B, c-m-B, m-sh-B, c-sh-B

Displays a backtrace of the stack. The default displays a brief backtrace of the stack.

A brief backtrace displays just the names of active function calls in the sequence in which they were called. In the display, each function points to the function it calls. For example:

```
BAZ ← BAR ← FOO ← EVAL ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

If you want a backtrace with more detailed information and/or with internal interpreter frames, use the :Detailed and :Internal keywords described below. See also the definitions of command accelerators below.

A numeric argument given with this command's accelerators, as well as the :Nframes keyword, specifies how many frames to display in the stack; for example, `c-9 c-B` displays nine frames.

keywords	:Continuations, :Detailed, :Internal, :Invisible, :Nframes
:Continuations	{Yes, No} Yes displays all continuation frames, which are frames that correspond to some internal function of a function. (Default is Yes. Mentioned default is No.)
:Detailed	{Yes, No} Displays a detailed backtrace of the stack, with arguments and their values. If a function sets one of the frame's arguments, then both the original argument supplied by the caller and the current value of the variable are displayed. (Default is No. Mentioned default is Yes.)
:Internal	{Yes, No} Displays internal interpreter frames in the backtrace. Ordinarily, when running interpreted code the Debugger tries to skip over frames that belong to functions of the interpreter, such as **si:*eval**, **prog**, and **cond**, and only show "interesting" functions. (Default is No. Mentioned default is Yes.)
:Invisible	{Yes, No} Yes displays all invisible frames. (Default is No. Mentioned default is Yes.)
:Nframes	{*number*} Designates how many frames to display in the backtrace. Enter a *number* to specify the number of frames to display. (Default is 10000.)

Key-binding accelerators

c-B	:Show Backtrace :Nframes 10000 (brief backtrace)
m-B	:Show Backtrace :Detailed Yes :Nframes 10000
c-m-B	:Show Backtrace :Internal Yes :Nframes 10000
m-sh-B	:Show Backtrace :Detailed No :Nframes 10000 :Invisible Yes :Continuations Yes

`c-sh-B`	:Show Backtrace :Detailed No :Nframes 10000 :Invisible No :Continuations No

Displays a backtrace of the stack. The default displays a brief backtrace of the stack.

```
1 Enter STOP (1) (1 2) (1 (2 3)) 4 5
Break: STOP.

 :1 (Continue) Return from BREAK.
 :2 (Abort) Return to toplevel.
Debug 1> :show-backtrace

;;; Stack Backtrace:

=> 0: (DEBUGGER::COMPUTE-FRAMES-IF-NECESSARY)
   1: (#<function:90faab>)
   2: (CONDITIONS::EXECUTE-DEBUGGER-COMMAND :SHOW-BACKTRACE NIL #)
   3: (INVOKE-DEBUGGER #)
   4: (BREAK "~a." STOP)
   5: (#<function:91e933> # # # 4 ...)
   6: (FIB1 1)
   7: (FIB1 2)
   8: (FIB1 3)
   9: (FIB1 4)
   10: ("Unknown")
   11: (EVAL #)
   12: (SYSTEM::LISTENER NIL)
   13: (SYSTEM::TOPLEVEL)
   14: (SYSTEM::APPLICATION-TOP-LEVEL #)
```

Show Bindings Command

:Show Bindings *keywords* `c-X B`

Displays the special variable bindings for one or more frames. When you enter this command, the Debugger displays special variable bindings beneath this message:

Names and values of specials bound in this frame:

keywords	:All, :Invisible, :Matching
:All	{Yes, No} Displays bindings for all frames in the stack. (Default is No. Mentioned default is Yes.)
:Invisible	{Yes, No} Yes shows bindings for all frames, visible and invisible. (The default is No. Mentioned default is Yes.)
:Matching	{*string*} Displays only the bindings for special variables whose symbol names contain a *string* that you specify. (Default is the current frame.)

Show Catch Blocks Command

:Show Catch Blocks *keyword*

Displays the active catch blocks for the current frame or for all frames. When you enter this command, the Debugger displays information in this format:

> *Open catch blocks and unwind-protects in this frame:*
> `Throwing to tag` *tag-name* `returns to` *frame* `at` *location*
> `with value(s)`

The *tag-name* is the name of the symbol that is catching the form. The *frame* is the name of the frame's function to which a **throw** operation returns. The *location* is a PC (program counter) line number in disassembled code.

keyword	:All
:All	{Yes, No} Displays active catch blocks for all frames in the stack. (Default is No. Mentioned default is Yes.)

Show Condition Handlers Command

:Show Condition Handlers *keyword*

Displays the condition handlers for the current frame or for all frames. Here is an example of what the Debugger displays when you enter this command for the current frame:

```
→ :Show Condition Handlers
Bound Handlers established in this frame:
  CONDITION-CASE handler for SYS:PARSE-ERROR
→
```

If the frame shown in the example above were not the current frame, and you used the :All keyword, the Debugger would display the name of the frame along with the condition handler information. For example:

```
→ :Show Condition Handlers (keywords) :All
For frame (DEFUN-IN-FLAVOR SI:INPUT-EDITOR-READ ... ):
  Bound Handlers established in this frame:
    CONDITION-CASE handler for SYS:PARSE-ERROR
→
```

keyword	:All
:All	{Yes, No} Displays condition handlers for all frames in the stack. (Default is No. Mentioned default is Yes.)

Show Instruction Command

:Show Instruction `c-m-I`

Displays the instruction that was just trapped in the Debugger or the instruction that would be executed next if you were to perform a single step operation. Here is an example of what the Debugger displays when you enter this command:

```
→ :Show Instruction
In (FLAVOR:METHOD :INPUT-EDITOR SI:INTERACTIVE-STREAM) at PC 160:
  PUSH-NIL
→
```

Show Lexical Environment Command

:Show Lexical Environment

Displays the lexical (local program) environment of the current frame, as established by the lexical ancestors of the frame. When you enter this command, the Debugger displays lexical (local) variables beneath this message:

Lexically inherited variables:

If the current frame has no lexical environment, the Debugger tells you:

```
This frame was not lexically called.
```

Show Proceed Options Command

:Show Proceed Options

Displays all of the currently available proceed and restart options. Here is an example of what the Debugger displays when you enter this command:

```
→ :Show Proceed Options
s-A, RESUME:
Supply a value to use this time as the value of FOO
s-B, s-sh-C:
Supply a value to store permanently as the value of FOO
s-C:
Retry the SYMEVAL instruction
s-D, ABORT:
Return to Lisp Top Level in Dynamic Lisp Listener 1
→
```

Suggested mouse operations

- To activate a proceed handler with the mouse: Display the proceed options with the :Show Proceed Options command, point the mouse at a proceed option, and click Left.

Show Special Command

:Show Special *symbol keyword*

Displays the special variable binding of a symbol in the context of the current frame's environment.

symbol	A symbol whose special variable binding you want to see.
keyword	:Environment
:Environment	{Program, Debugger, Streams} Evaluates and displays the symbol in the environment that you specify. Program specifies a program you are debugging. Debugger and Streams specify that you are debugging the Debugger. (Default is the environment of the current frame. Mentioned default is Program.)

Show Standard Value Warnings Command

Show Standard Value Warnings *keywords*

Displays more detailed information about standard variables that have been rebound. Here is an example of what the Debugger displays when you enter this command:

```
→  :Show Standard Value Warnings
The following standard values were bound:
  Rebinding CP:*COMMAND-TABLE* to #<COMMAND-TABLE User #o260252757>
  (old value was #<COMMAND-TABLE Debugger #o261747137>)
→
```

If no standard variables have been rebound, the Debugger tells you:

```
There were no standard values which required binding
```

keywords	:More Processing, :Output Destination
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

See the section "Standard Variables".

Symeval In Last Instance Command

:Symeval In Last Instance *symbol* `c-X c-I`

Evaluates a symbol as an instance variable in the context of the last instance to have been typed out.

symbol	A symbol to be evaluated.

Use Dynamic Environment Command

:Use Dynamic Environment `c-X I`

Changes the current evaluation mode from the lexical (local program) environment to the dynamic (global debugger) environment. Unless you debug your own Debugger, do not use this command. The :Use Dynamic Environment command is used by Symbolics development personnel who debug the Debugger. If you have entered the dynamic evaluation environment accidentally, you can get back to the lexical evaluation environment by entering the :Use Lexical Environment command or by pressing `c-X I`, which toggles between the two evaluation environments. The dynamic evaluation prompt is:

Eval (debugger):

Use Lexical Environment Command

:Use Lexical Environment `c-X I`

Changes the current evaluation mode from the dynamic (global debugger) environment to the lexical (local program) environment. When the Debugger is in this evaluation environment, you can examine local variables and arguments by simply typing their names, and you can use internal functions by name — functions defined with **flet** or **labels**. See the section "Evaluating a Form in the Debugger". The lexical evaluation prompt is:

Eval (program):

The `c-X I` accelerator toggles between the lexical evaluation environment and the dynamic evaluation environment.

Debugger Commands to Continue Execution

The Debugger provides commands that continue or restart execution. These commands, in alphabetical order, are:

- :Abort (`ABORT`, `c-Z`)
- :Disable Aborts

- :Enable Aborts
- :Proceed (RESUME)
- :Reinvoke (c-m-R)
- :Return (c-R)
- :Throw (c-T)

Abort Command

:Abort ABORT, c-Z

Depending on the context of its use: Returns to either top level or the previously invoked Debugger. Executes the abort instruction that appears in the list of proceed and restart options. :Abort is used to exit the Debugger. See the section "Exiting The Debugger". You can use the ABORT key in place of this command.

:abort
Depending on the context of its use, returns to either the top level or the previously invoked Debugger. Executes the abort instruction that appears in the list of proceed and restart options. Used to exit the Debugger.

```
1 Enter STOP (1) (1 2) (1 (2 3)) 4 5
Break: STOP.

 :1 (Continue) Return from BREAK.
 :2 (Abort) Return to toplevel.
Debug 1> :abort

=>
```

Disable Aborts Command

:Disable Aborts

Disables the use of the :Abort command. :Disable Aborts is useful for making sure you do not abort something accidentally.

Enable Aborts Command

:Enable Aborts

Enables the use of the :Abort command.

Proceed Command

:Proceed RESUME

Depending on the context of its use: Continues the execution of the program or process that has been suspended, executes the proceed-handler instruction that appears in the list of proceed and restart options, or returns to the previously invoked Debugger. You can use the RESUME key in place of this command.

Reinvoke Command

:Reinvoke *keyword* c-m-R, c-u c-m-R

Restarts execution of the function in the current frame. Any numeric argument given with this command's accelerator, as well as the :New Args keyword, prompts you for new argument values. If the function has been redefined — perhaps you edited the function to fix a bug — the new definition is used. The :Reinvoke command asks for confirmation before restarting the frame.

keyword	:New Args
:New Args	{Yes, No} Prompts you to supply new argument values for the reinvoked frame. (Default is No, which reinvokes the frame using current argument values. Mentioned default is Yes.)

Key-binding accelerators

c-m-R	:Reinvoke :New Args No
c-U c-m-R	:Reinvoke :New Args Yes

Return Command

:Return c-R

Returns from the current frame. This command prompts for as many values as the caller needs. You must enter values acceptable to the current frame's caller. For each value, the Debugger prompts you for a form, which it evaluates. It returns the resulting values, possibly after confirming them with you. If no values are expected, it requests confirmation before returning. The :Return command is useful when you want to simulate the return of a frame's execution, which was halted for some reason.

:return
Returns from the current frame. This command prompts for values You must enter values acceptable to the current frame's caller. Useful when you want to simulate the return of a frame's execution from a halt.

Throw Command

:Throw *symbol form* c-T

Executes a Lisp **throw** function and throws the result of evaluating *form* to the tag named by *symbol*. You can also use the Lisp function **throw**.

symbol	A catch tag.
form	A *form* to evaluate. The returned values from this evaluation are thrown to *symbol*.

throw *symbol*
Prompts for a form to be evaluated and thrown to *symbol*. You can also use the Lisp function `throw`.

Debugger Trap Commands

The Debugger provides commands associated with Debugger traps. These commands, in alphabetical order, are:

- :Clear Trap On Call (`c-X c-C`)
- :Clear Trap On Exit (`c-X c-E`)
- :Disable Condition Tracing (`c-X T`)
- :Enable Condition Tracing (`c-X T`)
- :Monitor Variable
- :Proceed Trap On Call (`c-X m-C`)
- :Restart Trap On Call (`c-X c-m-C`)
- :Set Trap On Call (`c-X C`)
- :Set Trap On Exit (`c-X E`)
- :Show Monitored Locations
- :Unmonitor Variable

A *trap* suspends a function's execution and, if there is no condition handler, causes entry to the Debugger. For example, a trap might be signalled when your program executes an illegal instruction, such as division by 0. Unless your program is prepared to handle the trap, the Debugger is entered.

The :Monitor Variable command also causes a trap and Debugger entry. This command triggers a *monitor trap* whenever a process accesses a special variable. If you have many different processes accessing a special variable, and you want to identify them all, you can simply specify the variable to be monitored. The trap occurs when that variable is referenced. You can also monitor instance variables and structure slots by clicking on them with the mouse. :Monitor Variable is useful if you want to keep track of and debug the interactions between the accessing processes. See the section "Monitor Variable Command".

The :Enable Condition Tracing command also signals a trap when you suspect a condition handler is broken and want to debug that handler. If you receive recursive error messages due to a defective handler, use :Enable Condition Tracing to cause a trap and enter the Debugger before the condition is signalled. See the section "Enable Condition Tracing Command".

Once in the Debugger, you can explicitly set traps by using the :Set Trap On Call and :Set Trap On Exit commands. A trap on exit suspends execution outside the called function, immediately after the function has returned. A trap on call suspends execution inside the called function, immediately before the first instruction.

The RESUME key can be used to continue returning or throwing whenever execution is suspended in a trap. When a trap on exit is set for a frame, throwing through that frame still signals the trap.

The ABORT key can be used to bypass the trap on exit.

The :Set Trap On Call, :Proceed Trap On Call, and :Restart Trap On Call commands have the following restriction: If you are metering all functions in a particular process, you cannot use trap on call in that process while metering is enabled.

Clear Trap On Call Command

:Clear Trap On Call c-X c-C

Clears trap on call for the current frame.

Clear Trap On Exit Command

:Clear Trap On Exit *keyword* c-X c-E

Clears trap on exit for the current frame or for all frames. Any numeric argument given with this command's accelerator clears trap on exit for all frames.

keyword	:All
:All	{Yes, No} Clears traps on exit for all frames in the stack. (Default is No. Mentioned default is Yes.)

Disable Condition Tracing Command

:Disable Condition Tracing c-X T

Disables condition tracing. The c-X T accelerator toggles between :Disable Condition Tracing and :Enable Condition Tracing. See the section "Enable Condition Tracing Command".

Enable Condition Tracing Command

:Enable Condition Tracing *condition keyword* c-X T

Enables condition tracing. That is, this command allows you to debug an error handler when it does not work properly. For example, when you receive continuous, recursive error messages due to a defective error handler, you can use :Enable Condition Tracing to cause a trap and enter the Debugger before the condition is signalled. Once in the Debugger, you can debug and fix the handler.

You should use this command only if you code your own error handlers. If you do not code your own handlers, and suspect there is a bug in a handler, send a bug report to your Symbolics customer representative.

Any numeric argument given with this command's accelerator sets **sys:trace-conditions** to **t**. The `c-X T` accelerator toggles between :Enable Condition Tracing and :Disable Condition Tracing.

condition	{**t**, **nil**, *conditions*} **t** enters the Debugger when any condition is signalled. **nil** turns off condition tracing previously specified by **t**. *condition* is a condition flavor, which causes entry to the Debugger when any flavor built on *conditions* are signalled.
keyword	:Conditional
:Conditional	{Always, Mode-Lock, Never, Once} Enables condition tracing according to certain conditions. Always: enables condition tracing in all cases. Mode-Lock: enables condition tracing only when the `MODE LOCK` key is held down. Never: has the effect of disabling condition tracing. Once: enables condition tracing only for the first time a condition is raised. (Default is Always.)

Monitor Variable Command

Monitor Variable *symbol keywords*

Monitors the access of a special variable. This command arranges for a trap to be signalled when any process accesses the monitored location. This command is used to answer the question: "What program or process is reading or writing this location in memory?". This is particularly useful when there are several processes sharing some data structures, and you want to debug the interactions between the processes.

symbol	The name of a symbol whose location in memory you want to monitor. Enter the name of a symbol and, optionally, its Value-Cell or Function-Cell. (See the :Cell keyword description below.)
keywords	:Boundp, :Cell, :Locf, :Makunbound, :Read, :Write
:Boundp	{Yes, No} Monitors the location for **boundp** operations. The default is No. The mentioned default is Yes.
:Cell	{Value-Cell, Function-Cell} Specifies the cell that you want to monitor within the location. The Debugger gives you two choices: Value-Cell or Function-Cell. The default is Value-Cell.

:Locf	{Yes, No} Monitors the location for **locf** operations. (Default is No.)
:Makunbound	{Yes, No} Monitors the location for **makunbound** operations. The default is No. The mentioned default is Yes.
:Read	{Yes, No} Monitors the location for reads. The default is No. The mentioned default is Yes.
:Write	{Yes, No, Change} Monitors the location for writes. The default is Yes.

Suggested mouse operations

- To monitor a location: Point the mouse at a locative, structure slot, or instance variable and press `c-m-sh`-Left.
- To unmonitor a location: Point the mouse at a locative, structure slot, or instance variable that was previously monitored and press `c-m-sh`-Middle.

Proceed Trap On Call Command

:Proceed Trap On Call `c-X m-C`

Resumes execution of the function in the current frame after setting trap on call. Use this command when you want to suspend execution at the entry to the next called function immediately. The :Restart Trap On Call command is similar, except that it restarts execution from the beginning of the current function before it suspends execution at the next called function. See the section "Restart Trap On Call Command". Using the :Proceed Trap On Call command is identical to using the :Set Trap On Call and :Proceed commands successively. The trap on call suspends execution inside the called function, immediately before the first instruction. See the section "Set Trap On Call Command".

Note: The Debugger might mistake this command for the :Proceed command if you attempt to type in the full command name. To avoid this problem, use the `c-X m-C` accelerator, surround the command name in quotes (excluding the colon), or type in:

```
:p t COMPLETE
```

to complete the command properly.

Restart Trap On Call Command

:Restart Trap On Call `c-X c-m-C`

Restarts execution of the function in the current frame, but first sets trap on call. Use this command when you want to restart execution of the current frame then immediately suspend execution at the entry to the next called function. The :Proceed Trap On Call command is similar, except that it resumes execution from wherever execution is suspended within the function instead of restarting execution from the beginning of the function. See the section "Proceed Trap On Call Command". Using the :Restart Trap On Call command is identical to using the :Set Trap On Call and :Reinvoke commands successively. The trap on call suspends execution inside the called function, immediately before the first instruction. See the section "Set Trap On Call Command".

Set Trap On Call Command

:Set Trap On Call `c-X C`

Sets trap on call for the next function called in the current frame. Use this command when you want to suspend execution at the entry of the next called function. (This command also sets trap on exit for the next called function.) The trap occurs only for the first time your program execution encounters the called function. A trap on call suspends execution inside the called function, immediately before the first instruction.

Set Trap On Exit Command

:Set Trap On Exit *keyword* `c-X E`

Sets trap on exit for the current frame or for all frames. The trap on exit occurs only for the first time your program execution returns the called function. Any numeric argument given with this command's accelerator sets traps on exit for all frames. When a trap on exit is set for a frame, throwing through that frame, via a Lisp **throw** function, still signals the trap.

keyword	:All
:All	{Yes, No} Sets traps on exit for all frames in the stack. (Default is No. Mentioned default is Yes.)

Show Monitored Locations Command

Show Monitored Locations

Displays all variables and other locations in memory that you are monitoring via the Monitor Variable command, the **dbg:monitor-location** function, and so on.

Unmonitor Variable Command

Unmonitor Variable *symbol keyword*

Stops monitoring one or all special variables in memory.

symbol	{*location*, RETURN} A *location* specifies one location that you want to stop monitoring. Enter the name of a symbol and, optionally, its Value-Cell or Function-Cell. (See the :Cell keyword description below.) Press the RETURN key if you want to stop monitoring all locations.
keyword	:Cell
:Cell	{Value-Cell, Function-Cell} Specifies which cell within the location you want to stop monitoring. The Debugger gives you two choices: Value-Cell or Function-Cell. (Default is Value-Cell.)

Suggested mouse operations

- To unmonitor a location: Point the mouse at a locative, structure slot, or instance variable that was previously monitored and press c-m-sh-Middle.

Debugger Commands for Breakpoints and Single Stepping

The Debugger provides breakpoint and single-step commands.

Like a trap, a Debugger *breakpoint* is also a suspension of a function's execution. Unlike a trap on call or trap on exit, any breakpoint that you set suspends execution every time your program encounters the breakpoint. You can set a breakpoint with the :Set Breakpoint command as well as other ways, listed below. Breakpoints are useful for examining data at strategic points in your program while your execution is frozen.

When you enter the Debugger via breakpoint, the Debugger displays the word `Break` in the top line of the error display. A Debugger breakpoint can be signalled by:

- Using the :Set Breakpoint command. See the section "Set Breakpoint Command".

- Performing mouse operations on the code fragments and disassembled code instructions output by the :Show Source Code and :Show Compiled Code commands respectively. See the section "Show Source Code Command". Also: See the section "Show Compiled Code Command".

- Pressing m-SUSPEND or c-m-SUSPEND. See the section "Entering the Debugger with m-SUSPEND, c-m-SUSPEND".

- Inserting the **break** or **zl:dbg** function into your program's source code. See the section "Entering the Debugger with **break** and **zl:dbg** Functions".

Do not confuse a Debugger breakpoint with a *break loop*. A break loop is a Dynamic Lisp Listener read-eval-print loop, which is activated when you suspend your current activity, via `SUSPEND` or `c-SUSPEND`. A Debugger breakpoint suspends into the Debugger, usually for the purpose of debugging a program.

You should set breakpoints only in your program's source code. Do not set a breakpoint in a system function — any code that the system depends on for its operations. Placing a breakpoint in a system function can produce dangerous results because your breakpoint may be encountered by other system functions. A breakpoint in the following types of functions can be particularly dangerous:

- Input/Output functions
- Input Editor functions
- Storage system functions
- Hardware I/O functions
- Garbage collecting functions

The term *single stepping* refers to the process of executing instructions, one instruction at a time. That is, the :Single Step command executes the next instruction, then suspends execution. The pattern becomes execute-suspend, execute-suspend, execute-suspend, and so on. The :Single Step command only operates on compiled code. To single step through interpreted code, use the Step facility or the **:step** option in the Trace facility. See the section "Stepping Through an Evaluation". Also: See the section "Tracing Function Execution". The :Single Step command steps over compiled functions. To step into a compiled function, use the :Set Trap On Call command on the function in which you want to step, then use the :Single Step command.

Commands for breakpoints and single stepping, in alphabetical order, are:

- :Clear All Breakpoints
- :Clear Breakpoint
- :Set Breakpoint
- :Show Breakpoints
- :Single Step (`c-sh-S`)

Clear All Breakpoints Command

:Clear All Breakpoints *compiled-function-spec*

Clears all breakpoints in the current frame's function or in any other compiled function.

compiled-function-spec
The name of a compiled function in which you want to clear breakpoints. (Default clears all breakpoints in the current frame's function.)

Clear Breakpoint Command

:Clear Breakpoint *compiled-function pc*

Clears a breakpoint.

compiled-function	The name of a *compiled function* in which you want to clear a breakpoint.
pc	The PC (program counter) at which you want to clear a breakpoint. The default is 1.

Suggested mouse operations

- To clear a breakpoint in a compiled function: Display disassembled code with the :Show Compiled Code command, point the mouse at a PC, and press c-m-Middle.
- To clear a breakpoint in a code fragment: Display the code with the :Show Source Code command, point the mouse at a code fragment, and press c-m-Middle.

Set Breakpoint Command

Set Breakpoint *compiled-function pc*

Sets a breakpoint.

compiled-function	The name of a *compiled-function* in which you want to set a breakpoint.
pc	The PC (program counter) at which you want to set a breakpoint.
keywords	:Action, :Conditional
:Action	{Show-All, Show-Args, Show-Locals} Specifies an action to take when the breakpoint is encountered. Show-All: Displays arguments and local variables. Show-Args: Displays arguments and no local variables. Show-Locals: Displays only local variables. Give an *expression* if you want it to be evaluated in the lexical context of the frame. (Default is no action. Mentioned default is Show-All.)
:Conditional	{Always, Mode-Lock, Never, Once} Executes the breakpoint trap according to certain conditions. Always: The breakpoint is always taken. Mode-Lock: The breakpoint is taken only when the MODE LOCK key is pressed. Never: The breakpoint is never taken. Once: The breakpoint is taken only for the first time it is encountered. Give an *expression* if you want it to be evaluated in the lexical context of the frame. (Default is Always.)

Suggested mouse operations

- To set a breakpoint in a compiled function: Display disassembled code with the Show Compiled Code command, point the mouse at a PC, and press c-m-Left.

- To set a breakpoint in a code fragment: Display the code with the Show Source Code command, point the mouse at a code fragment, and press c-m-Left.

Show Breakpoints Command

:Show Breakpoints

Displays all of the currently set breakpoints.

Single Step Command

:Single Step c-sh-S

Executes one instruction at a time and steps over function calls. This command works only on compiled code. For interpreted code, use the Step facility or the **:step** option in the Trace facility. For stepping into a compiled function, use the :Set Trap On Call command on the function in which you want to step, then use the :Single Step command.

Debugger Commands for System Transfer

The Debugger provides commands that allow you to enter other systems while debugging. These systems are:

- Zmacs, which allows you to edit your function
- A mail message window, which allows you to mail a bug report
- The Display Debugger

The Debugger commands that transfer you to these other systems are:

- :Edit Function (c-E)
- :Mail Bug Report (c-M)
- :Display Debugger (c-m-W)

Edit Function Command

:Edit Function *function* c-E

Enters the Zmacs editor to bring up the current function or any other function for editing. This command lets you look at the function's source code. This is useful when you have found the function that caused the error and want to fix the code right away. The editor command c-Z returns to the Debugger, if it is still there.

function	A stack frame that you select with the mouse or a function spec that specifies which function you want to edit. (Default edits the current function.)

Suggested mouse operations

To edit a function: Point the mouse at the function's stack frame and press m-Left.

Mail Bug Report Command

:Mail Bug Report *keyword* c-M

Brings up a mail message window and puts a backtrace into a mail message to be mailed as a bug report.

This command creates a new process and runs the **bug** function in that process. It starts up a mail-sending window that contains information from your herald identifying what version of the software you are using, a copy of the error message and a detailed backtrace of the stack. You are expected to report information explaining what you were doing when the problem occurred, preferably including a way for the person reading the bug report to make the problem happen again. The stack trace by itself is not adequate information for debugging. When you type the END key, the bug report is sent as mail, and you are brought back into the Debugger.

While composing the bug report, you can use normal window-switching commands such as FUNCTION S to switch back and forth between the Debugger and the mail-sending window.

A numeric argument given with this command's accelerator, c-M, as well as the :Nframes keyword, specifies the number of stack frames to put in your bug report; for example, c-5 c-M puts five frames into your report. The current stack frame begins the backtrace, so you might want to enter the :Top Of Stack command before you use :Mail Bug Report, if you have been examining frames other than the one that got the error. :Top Of Stack makes sure the error frame begins the backtrace.

You can control the character style of the herald information. See the variable **dbg:*character-style-for-bug-mail-prologue***.

keyword	:Nframes
:Nframes	{*stack-frame, number*} Specifies the number of stack frames to put into your bug report. Select a *stack-frame* with the mouse, or enter the *number* of most recent stack frames you want to send in your bug report. Frames that you specify show detailed

	information in the mail message. (Default places eight most recent frames into the mail message.)

Suggested mouse operations

- To put a backtrace in a mail message: Display the backtrace with the :Show Backtrace command, point the mouse at the *last* frame you want included in your backtrace, and click Left. All frames up to and including the frame you clicked on are put into the mail message.

Display Debugger Command

:Display Debugger `c-m-W`

Enters the Display Debugger.

Miscellaneous Debugger Commands

There are a few miscellaneous Debugger commands that do not fit into any logical category. These commands are:

- :Help (`c-HELP`)
- :Set Stack Size

Help Command

:Help `c-HELP`

Displays a list of all available Debugger commands with brief descriptions and keybinding accelerators.

Set Stack Size Command

:Set Stack Size *stack-type stack-size*

Sets the size of a stack.

stack-type	The type of the stack. Enter Control, Binding, or Data. (Default is Control.)
stack-size	The size of the stack. Enter a number of machine words that represents the stack size.

Summary of Debugger Commands

The following table summarizes all Debugger commands in alphabetical order. For each command, the table lists the command name, accelerators, positional arguments, keywords, and useful mouse operations.

This table appears only in the printed book. It does not appear online, in the Document Examiner. In the Document Examiner, you will see just an alphabetical list of all commands and their accelerators. To view a full command description for any command, simply point the mouse at the desired command in this list, and click `Mouse-Left`.

See the section "Debugger Command Descriptions". See also the online help file by pressing `c-HELP`.

Abort Command
: `ABORT`, `c-Z`: Returns to either top level or the previously invoked Debugger and executes the abort instruction that appears in the list of proceed and restart options.

Analyze Frame Command
: `c-m-Z`: Analyzes the erroneous frame and locates the source code of the current error.

Bottom Of Stack Command
: `m->`: Moves to the bottom of the stack, displays the least recent frame, and makes that frame current.

Clear All Breakpoints Command
: Clears all breakpoints in the current frame's function or in any other compiled function.

Clear Breakpoint Command
: Clears a breakpoint.

Clear Trap On Call Command
: `c-X c-C`: Clears trap on call for the current frame.

Clear Trap On Exit Command
: `c-X c-E`: Clears trap on exit for the current frame or for all frames.

Describe Last Command
: `c-m-D`: Executes the Lisp **describe** function on the most recently displayed value and leaves * set to that value.

Disable Aborts Command
: Disables the use of the :Abort command.

Disable Condition Tracing Command
: `c-X T`: Disables condition tracing.

Edit Function Command
: `c-E`: Enters the Zmacs editor to bring up the current function or any other function for editing.

Enable Aborts Command
: Enables the use of the :Abort command.

Enable Condition Tracing Command
: c-X T: The enable tracing command allows you to debug an error handler when it does not work properly.

Find Frame Command
: c-S: Searches the stack for a frame's function name that contains a specified string and makes that frame current.

Help Command
: c-HELP: Displays a list of all available Debugger commands with brief descriptions and key-binding accelerators.

Mail Bug Report Command
: c-M: Brings up a mail message window and puts a backtrace into a mail message to be mailed as a bug report.

Monitor Variable Command
: Monitors the access of a special variable.

Next Frame Command
: LINE, c-N, m-N, c-m-N: Moves down one frame, to the next less-recent frame (the calling frame), displays information about that frame, and makes it current.

Previous Frame Command
: RETURN, c-P, m-P, c-m-P, c-m-U: Moves up one frame, to the next most-recent frame (the frame that the current frame called), displays information about that frame, and makes it current.

Proceed Command
: RESUME: Continues the execution of the program or process that has been suspended, executes the proceed-handler instruction that appears in the list of proceed and restart options, or returns to the previously invoked Debugger.

Proceed Trap On Call Command
: c-X m-C: Resumes execution of the function in the current frame after setting trap on call.

Reinvoke Command
: c-m-R: Restarts execution of the function in the current frame.

Restart Trap On Call Command
: c-X c-m-C: Restarts execution of the function in the current frame, but first sets trap on call.

Return Command
: c-R: Returns from the current frame.

Set Breakpoint Command
: Sets a breakpoint.

Set Current Frame Command
: Makes the stack frame that you specify with the mouse become the current frame.

Set Stack Size Command
: Sets the size of a stack.

Set Trap On Call Command
: `c-X C`: Sets trap on call for the next function called in the current frame.

Set Trap On Exit Command
: `c-X E`: Sets trap on exit for the current frame or for all frames.

Show Arglist Command
: `c-X c-A`: Displays the argument list for the function in the current frame.

Show Argument Command
: `c-m-A`: Displays the value of one or all arguments in the current frame.

Show Backtrace Command
: `c-B`, `m-B`, `c-m-B`: Displays a backtrace of the stack.

Show Bindings Command
: `c-x B`: Displays the special variable bindings for one or more frames.

Show Breakpoints Command
: Displays all of the currently set breakpoints.

Show Catch Blocks Command
: Displays the active catch blocks for the current frame or for all frames.

Show Compiled Code Command
: Displays the disassembled code for a function.

Show Condition Handlers Command
: Displays the condition handlers for the current frame or for all frames.

Show Frame Command
: `REFRESH`, `c-L`, `m-L`: Displays information about the current frame.

Show Function Command
: `c-m-F`: Displays the name of the function in the current frame.

Show Instruction Command
: `c-m-I`: Displays the instruction that was just trapped in the Debugger or the instruction that would be executed next if you were to perform a single step operation.

Show Lexical Environment Command
: Displays the lexical (local program) environment of the current frame, as established by the lexical ancestors of the frame.

Show Local Command
: `c-m-L`: Displays the value of one or all local variables for the function in the current frame.

Show Monitored Locations Command
: Displays all variables and other locations in memory that you are moni-

toring via the Monitor Variable command, the **dbg:monitor-location** function, and so on.

Show Proceed Options Command
: Displays all of the currently available proceed and restart options.

Show Rest Argument Command
: Displays the **&rest** argument, if there is one, and formats it.

Show Source Code Command
: `c-X c-D`: Displays the source code for a function.

Show Special Command
: Displays the special variable binding of a symbol in the context of the current frame's environment.

Show Stack Command
: Displays all of the local-variable and temporary stack slots in the current frame.

Show Standard Value Warnings Command
: Displays more detailed information about standard variables that have been re-bound.

Show Value Command
: `c-m-V`: Displays one or all values being returned from the function that is being returned.

Single Step Command
: `c-sh-S`: Executes one instruction at a time and steps over function calls.

Symeval In Last Instance Command
: `c-X c-I`: Evaluates a symbol as an instance variable in the context of the last instance to have been typed out.

Throw Command
: `c-T`: Executes a Lisp **throw** function and throws the result of evaluating *form* to the tag named by *symbol*.

Top Of Stack Command
: `m-<`: Moves to the top of the stack (the frame where the error occurred), displays the most recent frame, and makes it current.

Unmonitor Variable Command
: Stops monitoring one or all special variables in memory.

Use Dynamic Environment Command
: `c-X I`: Changes the current evaluation mode from the lexical (local program) environment to the dynamic (global debugger) environment.

Use Lexical Environment Command
: `c-X I`: Changes the current evaluation mode from the dynamic (global debugger) environment to the lexical (local program) environment.

Display Debugger Command
: `c-m-W`: Enters the Display Debugger.

Debugger Functions

The Debugger's evaluation environment lets you type in Lisp forms, which it reads and evaluates in the lexical context of the current frame, and then prints. When you are typing these forms, you can use the following functions to examine or modify the arguments, local variables, function object, and values being returned in the current frame.

dbg:arg *name-or-number* *Function*

Returns the value of argument *name-or-number* in the current stack frame. **(setf (dbg:arg** *n***) x)** sets the value of the argument *n* in the current frame to the value of **x**. *name-or-number* can be the number of the argument (for example, 0 to specify the first argument) or the name of the argument. This function can be called only from the Debugger's evaluation environment.

dbg:loc *name-or-number* *Function*

Returns the value of the local variable *name-or-number* in the current stack frame. **(setf (dbg:loc** *n***) x)** sets the value of the local variable *n* in the current frame to the value of **x**. *name-or-number* can be the number of the local variable (for example, 0 to specify the first local variable) or the name of the local variable. This function can be called only from the Debugger's evaluation environment.

dbg:fun *Function*

Returns the function object of the current stack frame. **(setf (dbg:fun) x)** sets the function object of the current frame to the value of **x**. This function can be called only from the Debugger's evaluation environment.

dbg:val &optional *val-no 0* *Function*

Returns the value of the *val-no*th value to be returned from the current stack frame. **(setf (dbg:val** *val-no***) x)** sets the value of the *val-no*th value to be returned from the current frame to the value of **x**. *val-no* must be a fixnum (since values do not have names) and defaults to 0. **(dbg:val)** without a value number gives the first value. This function can be called only from the Debugger's evaluation environment.

dbg:monitor-location (*location* &key (*read* nil) (*write* t) (*makunbound* (eq write t)) (*boundp* (eq read t)) *locate name*) *Function*

Monitors a location; that is, causes entry to the Debugger whenever *location* is accessed by a process. *location* is a locative to the location to be monitored; for example, **(zl:value-cell-location 'foo)**. Descriptions of other arguments follow:

> *read* {**t**, **nil**} monitors the location for reads. (Default is **nil**.)
>
> *write* {**t**, **nil**} monitors the location for writes. (Default is **t**.)
>
> *makunbound* {**t**, **nil**} monitors the location for **makunbound** operations. (Default is the value of *write*)
>
> *boundp* {**t**, **nil**} monitors the location for **boundp** operations. (Default is the value of *read*.)
>
> *locate* {**t**, **nil**} monitors the location for **locf** operations. (Default is **nil**.)

dbg:monitor-instance-variable *instance instance-variable-name* &key (*read* nil) (*write* t) *makunbound boundp locate* *Function*

Monitors an instance variable; that is, causes entry to the Debugger whenever the instance variable is accessed by a process. *instance* is the name of an instance containing an *instance-variable* you want to monitor. Descriptions of other arguments follow:

> *read* {**t**, **nil**} monitors the instance variable for reads. (Default is **nil**.)
>
> *write* {**t**, **nil**} monitors the instance variable for writes. (Default is **t**.)
>
> *makunbound* {**t**, **nil**} monitors the instance variable for **makunbound** operations. (Default is **nil**.)
>
> *boundp* {**t**, **nil**} monitors the instance variable for **boundp** operations. (Default is **nil**.)
>
> *locate* {**t**, **nil**} monitors the instance variable for **locf** operations. (Default is **nil**.)

dbg:unmonitor-location *location* *Function*

Unmonitors a location. *location* is a locative to the location you want to stop monitoring.

Debugger Variables

The Debugger uses the following variables:

dbg:*frame* *Variable*

Inside the Debugger's evaluation environment, the value of **dbg:*frame*** is the location of the current frame.

dbg:*defer-package-dwim* *Variable*

When this is **nil** (the default), the Debugger searches over all packages to find any look-alike symbols when errors concerning unbound variables occur.

When the option is not **nil**, the search does not occur until you press `c-sh-P`. In this case, the Debugger offers `c-sh-P` in the list of commands even if the search would find no look-alike symbols.

dbg:*debug-io-override* *Variable*

Diverts the Debugger to a stream that is known to work; this can be useful when debugging. If the value of this variable is **nil** (the default), the Debugger uses the stream that is the value of ***debug-io***. But if the value of **dbg:*debug-io-override*** is not **nil**, the Debugger uses the stream that is the value of this variable instead. This variable should always be set (using **setq**), not bound, so all processes and stack groups can see it.

dbg:*show-backtrace* *Variable*

Backtrace information appears when you enter the Debugger. The default is **nil**.

Value	*Meaning*
nil	The Debugger startup message does not include any backtrace information.
t	The Debugger startup message includes a three-element backtrace.

dbg:*character-style-for-bug-mail-prologue* *Variable*

Creates the bug-report banner inserted into the text of bug messages, enabling you to choose the font. The default is NIL.NIL.TINY, specifying a small font for the bug-report banner.

To display a bug-report banner in a small font you can type the following:

```
(setq dbg:*character-style-for-bug-mail-prologue*
      (si:character-style-for-device-font 'fonts:quantum si:*b&w-screen*))
```

To display a bug-report banner in a large font you can type the following:

```
(setq dbg:*character-style-for-bug-mail-prologue*
      (si:parse-character-style '(nil nil :huge)))
```

You can also type the following to specify a particular font:

```
(setq dbg:*character-style-for-bug-mail-prologue* '(nil nil :huge))
```

Miscellaneous Debugging Techniques

Tracing Function Execution

The trace facility allows you to *trace* some functions. Tracing is useful when you need to find out why a program behaves in an unexpected manner, particularly when you suspect that arguments are being passed incorrectly or functions are being called in the wrong sequence.

Certain special actions are taken when a traced function is called and when it returns. The default tracing action prints a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and values.

You invoke the trace facility in several ways:

- Use the **trace** and **untrace** special forms.
- Click on [Trace] in the System menu. Enter or point to the function to be traced; a menu of options pops up.
- Invoke the Trace (m-X) command in the editor. Enter the function to be traced; a menu of options pops up.

The menu options are also available with **trace**; however, the syntax is complex.

trace &rest *specs* *Special Form*

A **trace** form looks like:

`(trace` *spec-1 spec-2* `...)`

Each *spec* can be one of the following:

A symbol

This is a function name, with no options. The function is traced in the default way, printing a message each time it is called and each time it returns.

A list (*function-name option-1 option-2* **...**)

function-name is a symbol and the *options* control how it is to be traced. For a list of the various options, see the section "Options to **trace**". Some options take arguments, which should be given immediately following the option name.

A list (**:function** *function-spec option-1 option-2 ...*)

This option is like the previous form except that *function-spec* need not be a symbol. Note that you cannot use this feature on a 386 based machine.

See the section "Function Specs". It exists because if *function-name* were a list in the previous form, it would instead be interpreted as the following form:

A list **(**(*function-1 function-2*...) *option-1 option-2* ...**)**
All the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

trace returns as its value a list of names of all functions it traced. If called with no arguments, as just **(trace)**, it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, **trace** calls **untrace** before setting up the new trace.

Tracing is implemented with encapsulation, so if the function is redefined (for example, with **defun** or by loading it from a compiled code file) the tracing is transferred from the old definition to the new definition.

It is recommended that you trace only user-defined functions and avoid tracing the system functions. Although some of the background processes use these functions, they never expect to have to type out anything. If they do have to type out something, the process will hang until you let it type out.

See the section "Encapsulations".

See the section "Options to **trace**".

Options to trace

The options to **trace** are:

:break *pred*
Enters a Dynamic Lisp Listener break loop after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-**nil**. During the break, the symbol **arglist** is bound to a list of the arguments of the function.

:exitbreak *pred*
This is just like **:break** except that the break loop is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the break, the symbol **arglist** is bound to a list of the arguments of the function, and the symbol **values** is bound to a list of the values that the function is returning.

:error
Calls the Debugger when the function is entered. Use `RESUME` to continue execution of the function. If this option is specified, no printed trace output appears other than the error message displayed by the Debugger. (Note: If you also want to call the Debugger when the function returns, use the Debugger's :Set Trap On Exit (`c-X E`) command.)

:step

Steps through interpreted code of a function whenever the function is called. For compiled code, use the Debugger's :Single Step command. See the section "Single Step Command".

See the section "Stepping Through an Evaluation".

:entrycond *pred*
Prints trace information on function entry only if *pred* evaluates to non-**nil**.

:exitcond *pred*
Prints trace information on function exit only if *pred* evaluates to non-**nil**.

:cond *pred*
Prints trace information on function entry and exit only if *pred* evaluates to non-**nil**.

:wherein *function*
Traces the function only when it is called, directly or indirectly, from the specified function *function*. You can give several trace specs to **trace**, all specifying the same function but with different **:wherein** options, so that the function is traced in different ways when called from different functions.

This is different from **advise-within**, which only affects the function being advised when it is called directly from the other function. The **trace :wherein** option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, and so on.

:per-process *process*
Traces the function in the specified process only. You must specify the processes as an argument.

:argpdl *pdl*
Specifies a symbol *pdl*, whose value is initially set to **nil** by **trace**. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments are pushed onto the *pdl* when the function is entered, and then popped when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl can serve several functions.

:entryprint *form*
form is evaluated and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values will appear, preceded by \\.

:exitprint *form*
form is evaluated and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:print *form*
form is evaluated and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:entry *list*
Specifies a list of arbitrary forms whose values are printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \\ to separate it from the other information.

:exit *list*
Similar to **:entry**, but specifies expressions whose values are printed with the exit-trace. The list of values printed is preceded by \\.

:arg :value :both nil
Specifies which of the usual trace printouts should be enabled.

If you specify	*Then*
:arg	On function entry prints the name of the function and the values of its arguments.
:value	On function exit prints the returned value(s) of the function.
:both	Same as if both **:value** and **:arg** were specified.
nil	Same as if neither **:value** nor **:arg** were specified.
None	The default is to **:both**.

If any further *options* appear after one of these, they are not treated as options. Rather, they are considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function, along with the normal trace information. The values printed are preceded by a //, and follow any values specified by **:entry** or **:exit**. Note that since these options "swallow" all following options, if one is given it should be the last option specified.

If the variable **arglist** is used in any of the expressions given for the **:cond**, **:break**, **:entry**, or **:exit** options, or after the **:arg**, **:value**, **:both**, or **nil** option, when those expressions are evaluated the value of **arglist** will be bound to a list of the arguments given to the traced function. Thus the following form would cause a break in **foo** if and only if the first argument to **foo** is **nil**.

```
    (trace (foo :break (null (car arglist))))
```

If the **:break** or **:error** option is used, the variable **arglist** will be valid inside the break-loop. If you **setq arglist**, the arguments seen by the function will change.

Similarly, the variable **values** will be a list of the resulting values of the traced function. For obvious reasons, this should only be used with the **:exit** option. If the **:exitbreak** option is used, the variables **values** and **arglist** are valid inside the break-loop. If you **setq values**, the values returned by the function will change.

You can "factor" the trace specifications, as explained earlier. For example,

```
    (trace ((foo bar) :break (bad-p arglist) :value))
```

is equivalent to

```
    (trace (foo :break (bad-p arglist) :value)
           (bar :break (bad-p arglist) :value))
```

Since a list as a function name is interpreted as a list of functions, nonatomic function names are specified as follows:

```
    (trace (:function (:method flavor :message) :break t))
```

See the section "Function Specs".

sys:trace-compile-flag *Variable*

If the value of **trace-compile-flag** is non-**nil**, the functions created by **trace** will get compiled, allowing you to trace special forms such as **cond** without interfering with the execution of the tracing functions. The default value of this flag is **nil**.

The following **trace** options are available:

:break *pred*
Enters a Dynamic Lisp Listener break loop after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-**nil**. During the break, the symbol **sys::arglist** is bound to a list of the arguments of the function.

:exitbreak *pred*
This is just like **:break** except that the break loop is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the break, the symbol **sys::arglist** is bound to a list of the arguments of the function, and the symbol **values** is bound to a list of the values that the function is returning.

:error
Calls the Debugger when the function is entered. If this option is specified, no printed trace output appears other than the error message displayed by the Debugger.

:entrycond *pred*
Prints trace information on function entry only if *pred* evaluates to non-**nil**.

:exitcond *pred*
Prints trace information on function exit only if *pred* evaluates to non-**nil**.

:cond *pred*
Prints trace information on function entry and exit only if *pred* evaluates to non-**nil**.

:wherein *function*
Traces the function only when it is called, directly or indirectly, from the specified function *function*. The **trace :wherein** option means that when the

traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, and so on.

:argpdl *pdl*
: Specifies a symbol *pdl*, whose value is initially set to **nil** by **trace**. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments are pushed onto the *pdl* when the function is entered, and then popped when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl can serve several functions.

:entryprint *form*
form is evaluated and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values will appear, preceded by \\.

:exitprint *form*
form is evaluated and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:print *form*
form is evaluated and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:entry *list*
Specifies a list of arbitrary forms whose values are printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \\ to separate it from the other information.

:exit *list*
Similar to **:entry**, but specifies expressions whose values are printed with the exit-trace. The list of values printed is preceded by \\.

:meter Enables metering when the function is entered and disables metering when the function is exited.

:arg On function entry prints the name of the function and the values of its arguments.

:value On function exit prints the returned value(s) of the function.

:both Same as if both **:value** and **:arg** were specified.

nil Same as if neither **:value** or **:arg** was specified.

None The default is to **:both**.

If any further *options* appear after one of these, they are not treated as options. Rather, they are considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function, along with the normal trace information. The values printed are preceded by a //, and follow any values specified by **:entry** or **:exit**. Note that since these options "swallow" all following options, if one is given it should be the last option specified.

If the variable **sys::arglist** is used in any of the expressions given for the **:cond**, **:break**, **:entry**, or **:exit** options, or after the **:arg**, **:value**, **:both**, or **nil** option, when those expressions are evaluated the value of **sys::arglist** will be bound to a list of the arguments given to the traced function. Thus the following form would cause a break in **foo** if and only if the first argument to **foo** is **nil**.

```
(trace (foo :break (null (car sys::arglist))))
```

If the **:break** or **:error** option is used, the variable **sys::arglist** will be valid inside the break-loop. If you **setq sys::arglist**, the arguments seen by the function will change.

Similarly, the variable **values** will be a list of the resulting values of the traced function. For obvious reasons, this should only be used with the **:exit** option. If the **:exitbreak** option is used, the variables **values** and **sys::arglist** are valid inside the break-loop. If you **setq values**, the values returned by the function will change.

You can "factor" the trace specifications, as explained earlier. For example,

```
(trace ((foo bar) :break (bad-p sys::arglist) :value))
```

is equivalent to

```
(trace (foo :break (bad-p sys::arglist) :value)
       (bar :break (bad-p sys::arglist) :value))
```

Controlling the Format of trace Output

Tracing output is printed on the stream that is the value of ***trace-output***. This is synonymous with ***terminal-io*** unless you change it. Following is an example of the default form of ***trace*** output:

```
1 Enter FACT 4.
| 2 Enter FACT 3.
|   3 Enter FACT 2.
|   | 4 Enter FACT 1.
|   |   5 Enter FACT 0.
|   |   5 Exit FACT 1.
|   | 4 Exit FACT 1.
|   3 Exit FACT 2.
| 2 Exit FACT 6.
1 Exit FACT 24.
```

You can use the variables **si:*trace-columns-per-level***, **si:*trace-bar-p***, **si:*trace-bar-rate***, and **si:*trace-old-style*** to control the format of **trace** output.

si:*trace-columns-per-level* *Variable*

For **trace** output, controls the number of columns of indentation that are added for each level of function call. The value must be an integer. The default is 2.

si:*trace-bar-p* *Variable*

For **trace** output, controls whether columns of vertical bars are printed. If the value is not **nil**, they are printed; otherwise, spaces are printed instead of the vertical bars. The default is **t** (print the bars).

si:*trace-bar-rate* *Variable*

When **si:*trace-bar-p*** is not **nil**, columns of vertical bars are printed in **trace** output for every *n* levels of function call, where *n* is the value. The value must be an integer. The default is 2.

si:*trace-old-style* *Variable*

If not **nil**, the old, Maclisp-compatible form of printing **trace** output is used. The default is **nil** (use the new style).

Untracing Function Execution

untrace &quote &rest *fns* *Function*

Undoes the effects of **trace** and restore functions *fns* to their normal, untraced state. **untrace** takes multiple specifications, for example, **(untrace foo bar baz)**. Calling **untrace** with no arguments untraces all functions currently being traced.

The arguments should be names of globally defined, traced functions; the names are not evaluated.

```
(untrace delete-duplicates position-if)
 => POSITION-IF
```

Advising a Function

To *advise* a function is to tell a function to do something extra in addition to its actual definition. Advising is achieved by means of the special form **advise**. The something extra is called a piece of advice, and it can be done before, after, or around the definition itself. The advice and the definition are independent, in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function to give the user information about when and how the function is called and when and with what value it returns. Advising is intended for semipermanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing out a change to a function in a way that is easy to retract. In this case, you would call **advise** from the console. It can also be used for customizing a function that is part of a program written by someone else. In this case you would be likely to put a call to **advise** in one of your source files or your login init file rather than modifying the other person's source code. See the section "Logging In".

Advising is implemented with encapsulation, so if the function is redefined (for example, with **defun** or by loading it from a compiled code file), the advice will be transferred from the old definition to the new definition. See the section "Encapsulations".

advise *function class name position* &body *forms* *Special Form*

A function is advised by the special form

```
(advise function class name position
  form1 form2...)
```

None of this is evaluated.

function Specifies the function to put the advice on. It is usually a symbol, but any function spec is allowed. (See the section "Function Specs".)

class Specifies either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). For more information about the meaning of **:around**, **:before**, and **:after** advice: See the section "**:around** Advice".

name Specifies an arbitrary symbol that is remembered as the name of this particular piece of advice. It is used to keep track of multiple pieces of advice on the same function. If you have no name in mind, use **nil**; then we say the piece of advice is anonymous.

A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first.

Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

position Specifies where to put this piece of advice in relation to others of the same class already present on the same function.

Position can have these values:

- *position* can be **nil**. The new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.

- *position* can be a number, which is the number of pieces of advice of the same class to precede this one. For example, 0 means at the beginning; a very large number means at the end.
- *position* can have the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

forms Specifies the advice; they get evaluated when the function is called.

Example: The following form modifies the factorial function so that if it is called with a negative argument it signals an error instead of running forever.

```
(advise factorial :before negative-arg-check nil
  (if (minusp (first arglist))
      (ferror "factorial of negative argument")))
```

Sometimes you use **advise** for per-site customizations. In this case, it is undesirable to use **unadvise**, thus removing even the more "permanent" advice. To specify this more permanent advice: See the function **si:advise-permanently**.

si:advise-permanently *function class name position* &body *forms* *Function*

Identical to **advise**, except that forms advised by **si:advise-permanently** cannot be removed by **unadvise**. They must be removed by **si:unadvise-permanent**. See the function **si:unadvise-permanent**.

function Specifies the function to put the advice on. It is usually a symbol, but any function spec is allowed. (See the section "Function Specs".)

class Specifies either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). For more information about the meaning of **:around**, **:before**, and **:after** advice, see the section "**:around** Advice".

name Specifies an arbitrary symbol that is remembered as the name of this particular piece of advice. It is used to keep track of multiple pieces of advice on the same function. If you have no name in mind, use **nil**; then we say the piece of advice is anonymous.

A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first.

Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

position Specifies where to put this piece of advice in relation to others of the same class already present on the same function.

Position can have these values:

- *position* can be **nil**. The new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.
- *position* can be a number, which is the number of pieces of advice of the same class to precede this one. For example, 0 means at the beginning; a very large number means at the end.
- *position* can have the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

forms Specifies the advice; they get evaluated when the function is called.

si:show-permanent-advice *Function*

Displays all functions which currently have permanent advice.

si:unadvise-permanent *function class* &optional *position* *Function*

Removes pieces of advice whether put there by **advise**, or by **si:advise-permanently**.

function Specifies the function from which to remove the advice. It is usually a symbol, but any function spec is allowed. (See the section "Function Specs".)

class Specifies either **:before**, **:after**, or **:around**, and which advice (before, after, or around the execution of the definition of the function) to remove. For more information about the meaning of **:around**, **:before**, and **:after** advice, see the section "**:around** Advice".

position specifies which piece of advice to remove. It can be the numeric index (0 means the first one) or it can be the name of the piece of advice.

unadvise &optional *function class position* *Function*

Removes pieces of advice. None of its subforms are evaluated. *function* and *class* have the same meaning as they do in the function **advise**. *position* specifies which piece of advice to remove. It can be the numeric index (0 means the first one) or it can be the name of the piece of advice.

unadvise can remove more than one piece of advice if some of its arguments are missing or **nil**. The arguments *function*, *class*, and *position* all act independently. A missing value or **nil** means all possibilities for that aspect of advice. For example, the following form removes all **:before**, **:after**, and **:around** advice named

negative-arg-check on the **factorial** function:

```
(unadvise factorial nil negative-arg-check)
```

In this example **unadvise** removes all **:around** advice on all functions in all positions with all names:

```
(unadvise nil :around)
```

In this example **unadvise** removes all classes of advice named **my-personal-advice** on all functions:

```
(unadvise nil nil my-personal-advice)
```

(unadvise) removes all advice on all functions, since *function*, *class*, and *position* take on all possible values.

The following are the primitive functions for adding and removing advice. Unlike the special forms **advise** and **unadvise**, the following are functions and can be conveniently used by programs. **advise** and **unadvise** are actually macros that expand into calls to these two.

si:advise-1 *function class name position forms* *Function*

Adds advice. The arguments have the same meaning as in **advise**. Note that the *forms* argument is *not* a **&rest** argument.

si:unadvise-1 *function* &optional *class position* *Function*

Removes advice. *function*, *class*, and *position* are independent. If *function*, *class*, or *position* is **nil**, or if *class* or *position* is unspecified, all classes of advice or advice for all functions, at all positions, or with all names is removed.

You can find out manually what advice a function has with **grindef**, which grinds the advice on the function as forms that are calls to **advise**. These are in addition to the definition of the function. See the special form **grindef**.

To poke around in the advice structure with a program, you must work with the encapsulation mechanism's primitives. See the section "Encapsulations".

si:advised-functions *Variable*

A list of all functions that have been advised.

Designing the Advice

For advice to interact usefully with the definition and intended purpose of the function, it needs access to the data flow and control flow through the function. The system provides conventions for doing this.

The list of the arguments to the function can be found in the variable **arglist**. The value of the variable **arglist** sometimes includes internal arguments that should be ignored. In some cases these internal arguments are machine-dependent. If you are advising only ordinary functions defined with **defun**, you need not worry about this. In the following table, the arguments noted as *ignore* are internal:

Function Type	3600 Arglist	Ivory Arglist
Method	(*self ignore ignore args...*)	(*ignore self args...*)
Whopper	(*self ignore ignore args...*)	(*ignore self args...*)
defun-in-flavor	(*self ignore args...*)	(*ignore self args...*)
:internal	(*ignore args...*)	(*ignore args...*)

To find out what the value of the variable **arglist** will be when advising a function, you can evaluate the form (**arglist** '*function-name*). The result always includes the internal arguments.

:before advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with:

```
(setq arglist (copylist arglist))
```

After the function's definition has been executed, the list of the values it returned can be found in the variable **values**. **:after** advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a **prog**, so any piece of advice can exit the entire function and return some values with **return**. No further advice will be executed. If a piece of **:before** advice does this, the function's definition will not even be called.

:around Advice

A piece of **:before** or **:after** advice is executed entirely before or entirely after the definition of the function. **:around** advice is wrapped around the definition; that is, the call to the original definition of the function is done at a specified place inside the piece of **:around** advice. You specify where by putting the symbol **:do-it** in that place.

For example, **(+ 5 :do-it)** as a piece of **:around** advice would add **5** to the value returned by the function. This could also be done by the following:

```
(setq values (list (+ 5 (car values))))
```

as **:after** advice.

When there is more than one piece of **:around** advice, they are stored in a sequence just like **:before** and **:after** advice. Then, the first piece of advice in the sequence is the one started first. The second piece is substituted for **:do-it** in the first one. The third one is substituted for **:do-it** in the second one. The original definition is substituted for **:do-it** in the last piece of advice.

:around advice can access **arglist**, but **values** is not set up until the outermost **:around** advice returns. At that time, it is set to the value returned by the **:around** advice. It is reasonable for the advice to receive the values of the **:do-it** (for example, with **multiple-value-list**) and play with them before returning them (for example, with **values-list**).

:around advice can **return** from the **prog** at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain **:do-it**. Containing two instances of **:do-it** can be useful under peculiar circumstances. If you are careless, however, the original definition might be called twice, but something like the following certainly works reasonably:

```
(if (foo) (+ 5 :do-it) (* 2 :do-it))
```

Advising One Function Within Another

It is possible to advise the function **foo** only when it is called directly from a specific other function **bar**. You do this by advising the function specifier **(:within bar foo)**. That works by finding all occurrences of **foo** in the definition of **bar** and replacing them with **altered-foo-within-bar**. This can be done even if **bar**'s definition is compiled code. The symbol **altered-foo-within-bar** starts off with the symbol **foo** as its definition; then the symbol **altered-foo-within-bar**, rather than **foo** itself, is advised. The system remembers that **foo** has been replaced inside **bar**, so that if you change the definition of **bar**, or advise it, then the replacement is propagated to the new definition or to the advice. If you remove all the advice on **(:within bar foo)**, so that its definition becomes the symbol **foo** again, then the replacement is unmade and everything returns to its original state.

(grindef bar) prints **foo** where it originally appeared, rather than **altered-foo-within-bar**, so the replacement will not be seen. Instead, **grindef** prints calls to **advise** to describe all the advice that has been put on **foo** or anything else within **bar**.

An alternate way of putting on this sort of advice is to use **advise-within**.

advise-within *within-function function-to-advise class name position* &body *forms* *Function*

An **advise-within** form looks like this:

```
(advise-within within-function function-to-advise
                class name position
          forms...)
```

It advises *function-to-advise* only when called directly from the function *within-function*. The other arguments mean the same thing as with **advise**. None of them is evaluated.

To remove advice from **(:within bar foo)**, you can use **unadvise** on that function specifier. Alternatively, you can use **unadvise-within**.

unadvise-within *within-function* &optional *advised-function class position* *Function*

An **unadvise-within** form looks like this:

```
(unadvise-within within-function function-to-advise class position)
```

It removes advice that has been placed on `(:within` *within-function function-to-advise*`)`. The arguments *class* and *position* are interpreted as for **unadvise**.

For example, if those two arguments are omitted, then all advice placed on *function-to-advise* within *within-function* is removed. Additionally, if *function-to-advise* is omitted, all advice on any function within *within-function* is removed. If there are no arguments, than all advice on one function within another is removed. Other pieces of advice, which have been placed on one function and not limited to within another, are not removed.

(unadvise) removes absolutely all advice, including advice for one function within another.

The function versions of **advise-within** and **unadvise-within** are called **si:advise-within-1** and **si:unadvise-within-1** respectively. **advise-within** and **unadvise-within** are macros that expand into calls to the other two.

Compiled Advice

You have the option of whether or not to compile or interpret advice. You can control this globally, by using **si:*advice-compiled-by-default***, or individually (on a function by function basis) by using **si:compile-advice** and **si:interpret-advice**.

si:*advice-compiled-by-default* *Variable*

When this varible is set to **t**, **advise** and **si:advise-permanently** cause the advice to be compiled. When **si:*advice-compiled-by-default*** is set to **nil**, the advice is interpreted.

si:compile-advice *function* *Function*

function must be a function spec of a compiled function that is currently advised. This specification is "sticky" until the next time all advice is removed from *function*. Until then, all advice for *function* is compiled.

si:interpret-advice *Function*

function must be a function spec of a compiled function that is currently advised. This specification is "sticky" until the next time all advice is removed from *function*. Until then, all advice for *function* is interpreted.

Stepping Through an Evaluation

The step facility gives you the ability to follow every step of the evaluation of an interpreted form and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. Use the step facility if your program is behaving strangely, and it is not obvious how it is getting into this strange state.

You can enter the stepper in two ways:

- Use the **step** function.
- Use the **:step** option of **trace**.

step *form*	Evaluates *form* with single stepping.

If a function is traced with the **:step** option, whenever that function is called it will be single stepped. See the section "Options to **trace**". Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be handled by **step**.

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a right-facing arrow (→) character. When a macro is expanded, the expansion is printed out preceded by a double arrow (↔) character. When a form returns a value, the form and the values are printed out preceded by a left-facing arrow (←) character; if more than one value is being returned, an and-sign (∧) character is printed between the values.

Since the forms can be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from you. A variety of commands exist to tell the stepper how to proceed, or to look at what is happening.

`c-N` (Next)	Steps to the next thing. The stepper continues until the next thing to print out, and it accepts another command.
`SPACE`	Goes to the next thing at this level. In other words, it continues to evaluate at this level, but does not step anything at lower levels. In this way you can skip over parts of the evaluation that do not interest you.
`c-U` (Up)	Continues evaluating until we go up one level. Similar to the `SPACE` command; it skips over anything on the current level as well as lower levels.
`c-X` (Exit)	Exits; finishes evaluating without any more stepping.
`c-T` (Type)	Retypes the current form in full (without truncation).
`c-G` (Grind)	Grinds (that is, pretty-prints) the current form.
`c-E` (Editor)	Enters the editor.
`c-B` (Breakpoint)	This command puts you into a breakpoint (that is, a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:

step-form	The current form.
step-values	The list of returned values.
step-value	The first returned value.

You can change the values of these variables within the current environment.

You can also refer to local variables and arguments in the function.

`c-L`	Clears the screen and redisplays the last ten pending forms (forms being evaluated).
`m-L`	Like `c-L`, but does not clear the screen.
`c-m-L`	Like `c-L`, but redisplays all pending forms.
`?` or `HELP`	Prints documentation on these commands.

It is strongly suggested that you write a little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

A Hook Into the Evaluator

The **evalhook** facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses **evalhook**; however, if you want to write your own stepper or something similar, then use this primitive albeit complex facility to do so.

evalhook *Variable*

In your new programs, we recommend that you use the variable ***evalhook*** which is the Common Lisp equivalent of **evalhook**.

If the value of **evalhook** is non-**nil**, then special things happen in the evaluator. When a form (any form, even a number or a symbol) is to be evaluated, **evalhook** is bound to **nil** and the function that was **evalhook**'s value is applied to one argument — the form that was trying to be evaluated. The value it returns is then returned from the evaluator.

evalhook is bound to **nil** by **break** and by the Debugger, and **setq**ed to **nil** when errors are dismissed by throwing to the Lisp top-level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on **evalhook**. It applies only to evaluation — whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function **eval**. It does *not* have any effect on compiled function references, on use of the function **apply**, or on the "mapping" functions.

If you are using CLOE, **evalhook** causes **eval** to pass its argument form to the value of ***evalhook***. This only occurs when the variable is not **nil**. The value of ***evalhook*** should be a function.

evalhook *form evalhook* &optional *applyhook env* *Function*

This function helps exploit the **evalhook** feature. The *form* is evaluated with **evalhook** lambda-bound to the function *evalhook*, and **applyhook** lambda-bound to the function given as applyhook. The checking of **evalhook** is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger. *env* is used as the lexical environment for the operation. *env* defaults to the null environment.

Note: While the Symbolics Common Lisp version of this function does not require the argument *applyhook*, the function as specified in *Common LISP: the Language* and as implemented in CLOE Runtime does.

```
Example:
;; This function evaluates a form while printing debugging
;; information.
(defun hook (x)
   (terpri)
   (evalhook x 'hook-function))

;; Notice how this function calls evalhook to evaluate the
;;  form f, so as to hook the subforms.
(defun hook-function (f)
   (let ((v (evalhook f 'hook-function)))
     (format t "form: ~s~%value: ~s~%" f v)
     v))

;; This isn't a very good program, since if f returns multiple
;; values, it will not work.
```

The following output might be seen from **(hook '(cons (car '(a . b)) 'c))**:

```
form: (quote (a . b))
value: (a . b)
form: (car (quote (a . b)))
value: a
form: (quote c)
value: c
(a . c)
```

Normally after **eval** has evaluated the arguments to a function, it calls the function. If *applyhook* exists, however, **eval** calls the hook with two arguments: the function and its list of arguments. The values returned by the hook constitute the values for the form. The hook could use **zl:apply** on its arguments to do what **eval** would have done normally. This hook is active for special forms as well as for real functions.

Whenever either an evalhook or applyhook is called, both hooks are bound off. The evalhook itself can be **nil** if only an applyhook is needed.

applyhook catches only **apply** operations done by **eval**. It does not catch **apply** called in other parts of the interpreter or **apply** or **funcall** operations done by other functions such as **mapcar**. In general, such uses of **apply** can be dealt with by intercepting the call to **mapcar**, using the applyhook, and substituting a different first argument.

The argument list is like an **&rest** argument: it might be stack-allocated but is not guaranteed to be. Hence you cannot perform side-effects on it and you cannot store it in any place that does not have the same dynamic extent as the call to *applyhook*.

Compatibility Note: In SCL's implementation, the variable name **evalhook** is not available to the user. The incompatibility with the implementation in *Common Lisp the Language* is that:

- **evalhook** is initially bound.
- **evalhook** is **special**.
- **evalhook** should not be altered (via the use of assigning, binding, making unbound, and so on).

A Hook Into zl:apply

applyhook provides a hook into **zl:apply**, much as **evalhook** provides a hook into **eval**.

applyhook *Variable*

In your new programs, we recommend that you use the variable ***applyhook*** which is the Common Lisp equivalent of **applyhook**.

When the value of this variable is not **nil** and **eval** calls **zl:apply**, **applyhook** is bound to **nil** and the function that was its value is applied to two arguments: the function that **eval** gave to **apply** and the list of arguments to that function. The value it returns is returned from the evaluator.

If you are using CLOE, **applyhook** changes the action that takes place when a function is applied to its arguments. This only occurs when the variable is not **nil**. The function and its arguments to the value of **applyhook** (which should be a function) are passed by **zl:apply**.

applyhook *function args evalhook applyhook* &optional *env* *Function*

function is applied to *args* with **evalhook** lambda-bound to the function *evalhook* and with **applyhook** lambda-bound to the function *applyhook*.

Like the **evalhook** function, this bypasses the first place where the relevant hook would normally be triggered. *env* is used as the lexical environment for the operation. *env* defaults to the null environment. *evalhook* or *applyhook* can be nil.

Compatibility Note: In SCL's implementation, the variable name **applyhook** is not available to the user. The incompatibilities with the implementation specified in *Common Lisp: the Language* are:

- **applyhook** is initially bound.
- **applyhook** is **special**.
- **applyhook** should not be altered (via the use of assigning, binding, making unbound, and so on).

The Inspector

How the Inspector Works

The Inspector is a window-oriented program for inspecting data structures. When you ask to inspect a particular object, its components are displayed. The particular components depend on the type of object; for example, the components of a list are its elements, and those of a symbol are its value binding, function definition, and property list.

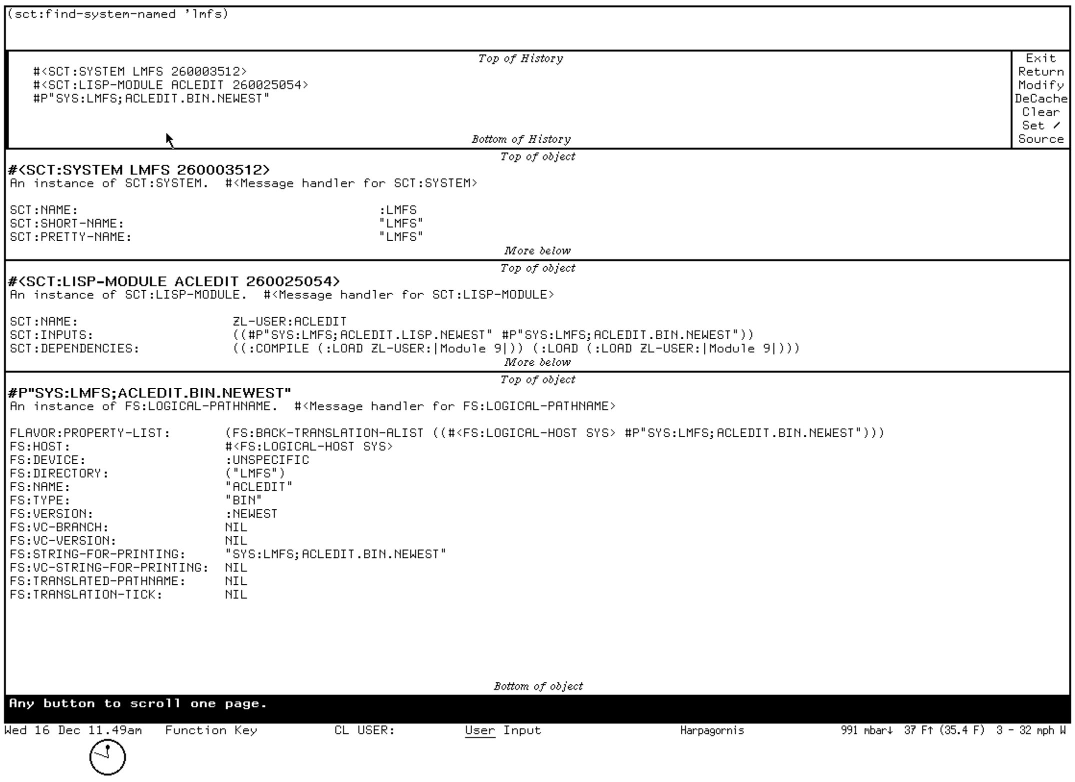
The component objects displayed on the screen by the Inspector are mouse-sensitive, allowing you to do something to that object, such as inspect it, modify it, or give it as the argument to a function. Choose these operations from the menu pane at the top-right part of the screen.

When you click on a component object itself, that component object gets inspected. It expands to fill the window and its components are shown. In this way, you can explore a complex data structure, looking into the relationships between objects and the values of their components.

The Inspector can be part of another program or it can be used standalone. Note, however, that although the display looks the same as that of the standalone Inspector, the handling of the mouse buttons depends upon the particular program being run.

Figure ! shows the standalone Inspector window. The display consists of the following panes, from top to bottom:

- A small interaction pane
- A history pane and menu pane
- Some number of inspection panes (three by default)



```
(sct:find-system-named 'lmfs)

                                    Top of History                                  Exit
   #<SCT:SYSTEM LMFS 260003512>                                                     Return
   #<SCT:LISP-MODULE ACLEDIT 260025054>                                             Modify
   #P"SYS:LMFS;ACLEDIT.BIN.NEWEST"                                                  DeCache
                                                                                    Clear
                                                                                    Set ⁄
                                   Bottom of History                                Source
                                     Top of object
#<SCT:SYSTEM LMFS 260003512>
An instance of SCT:SYSTEM.  #<Message handler for SCT:SYSTEM>

SCT:NAME:                                   :LMFS
SCT:SHORT-NAME:                             "LMFS"
SCT:PRETTY-NAME:                            "LMFS"
                                      More below
                                     Top of object
#<SCT:LISP-MODULE ACLEDIT 260025054>
An instance of SCT:LISP-MODULE.  #<Message handler for SCT:LISP-MODULE>

SCT:NAME:                  ZL-USER:ACLEDIT
SCT:INPUTS:                ((#P"SYS:LMFS;ACLEDIT.LISP.NEWEST" #P"SYS:LMFS;ACLEDIT.BIN.NEWEST"))
SCT:DEPENDENCIES:          ((:COMPILE (:LOAD ZL-USER:|Module 9|)) (:LOAD (:LOAD ZL-USER:|Module 9|)))
                                      More below
                                     Top of object
#P"SYS:LMFS;ACLEDIT.BIN.NEWEST"
An instance of FS:LOGICAL-PATHNAME.  #<Message handler for FS:LOGICAL-PATHNAME>

FLAVOR:PROPERTY-LIST:       (FS:BACK-TRANSLATION-ALIST ((#<FS:LOGICAL-HOST SYS> #P"SYS:LMFS;ACLEDIT.BIN.NEWEST")))
FS:HOST:                    #<FS:LOGICAL-HOST SYS>
FS:DEVICE:                  :UNSPECIFIC
FS:DIRECTORY:               ("LMFS")
FS:NAME:                    "ACLEDIT"
FS:TYPE:                    "BIN"
FS:VERSION:                 :NEWEST
FS:VC-BRANCH:               NIL
FS:VC-VERSION:              NIL
FS:STRING-FOR-PRINTING:     "SYS:LMFS;ACLEDIT.BIN.NEWEST"
FS:VC-STRING-FOR-PRINTING:  NIL
FS:TRANSLATED-PATHNAME:     NIL
FS:TRANSLATION-TICK:        NIL



                                    Bottom of object
Any button to scroll one page.
Wed 16 Dec 11.49am   Function Key      CL USER:     User Input        Harpagornis      991 mbar↓  37 F↑ (35.4 F)  3 - 32 mph W
```

Figure 7. The Inspector

Entering and Leaving the Inspector

You can enter the standalone Inspector via:

- Select Activity Inspector
- `SELECT I`
- [Inspect] in the System menu
- The Inspect command, which inspects its argument, if any
- The **inspect** function, which inspects its argument, if any

Warning: If you enter with the Inspect command or the **inspect** function, the Inspector is not a separate activity from the Lisp Listener in which you invoke it. In this case you cannot use `SELECT L` to return to the Lisp Listener; you should *always* exit via the [Exit] or [Return] option in the Inspector menu. If you forget and exit the Inspector by selecting another activity, you might need to use `c-m-ABORT` to return the Lisp Listener to its normal state.

The Inspector Interaction Pane

The interaction pane has two functions: to prompt you and to receive input. If you are not being asked a question, then a read-eval-inspect loop is active. Any forms you type are echoed in the interaction pane and evaluated. The result is not printed, but rather inspected. When you are prompted for input, usually due to having invoked a menu operation, any input you type at the read-eval-inspect loop is saved away and erased from the interaction pane. When the interaction is finished, the input is re-echoed and you can continue to type the form.

The Inspector History Pane

The history pane maintains a list of all objects that you have inspected, allowing you to back up and continue down another path. The last recently displayed object is at the top of the list, and the most recently displayed object is at the bottom.

You can inspect any mouse-sensitive object in the history pane by clicking on it. In addition, you can perform other operations by placing the mouse cursor in the *line region*, which is the left-hand side of the history pane, the area bounded by the margin on one side and the list of objects on the other. In the line region the shape of the mouse cursor changes to a rightward-pointing arrow.

- Clicking Left in the line region inspects the object. This is sometimes useful when the object is a list and it is inconvenient to position the mouse at the open parenthesis.

- Clicking Middle deletes the object from the history.

The history pane also maintains a cache allowing quick redisplay of previously displayed objects. This means that merely reinspecting an object does not reflect any changes in its state. Clicking Middle in the line region deletes the object from the cache as well as deleting it from the history pane. Use [DeCache] in the menu pane to clear everything from the cache.

The history pane has a scroll bar at the far left, as well as scrolling zones in the middle of its top and bottom edges. The last three lines of the history are always the objects being inspected in the inspection panes.

The Inspector Menu Pane

The menu pane (to the right of the history pane) displays these infrequently used but useful commands:

[Exit]	Equivalent to `c-Z`. Exits the Inspector and deactivates the frame.
[Return]	Similar to [Exit], but allows selection of an object to be returned as the value of the call to **inspect**.
[Modify]	Allows simple editing of objects. Selecting [Modify] changes the mouse sensitivity of items on the screen to only include fields that

are modifiable. In the typical case of named slots, the names are the mouse-sensitive parts. When the field to modify has been selected, a new value can be specified either by typing a form to be evaluated or by using the mouse to select any normally mouse-sensitive object. The object being modified is redisplayed. Clicking Right at any time aborts the modification.

[DeCache]	Flushes all knowledge about the insides of previously displayed objects and redisplays the currently displayed objects.
[Clear]	Clears out the history, the cache, and all the inspection panes.
[Set] \	Sets the value of the symbol \ by choosing an object.

The Inspector Inspection Pane

Each inspection pane can inspect a different object. When you inspect an object it appears in the large inspection pane at the bottom, and the previously inspected objects shift upward.

At the top of an inspection pane is either a label, which is the printed representation of the object being inspected in that window, or the words "a list", which means a list is being inspected. The main body of an inspection pane is a display of the components of the object, labelled with their names, if any. You can scroll this display using the scroll bar on the left or the "more above" and "more below" scrolling zones at the top and bottom.

Clicking on any mouse-sensitive object in an inspection pane inspects that object. The three mouse buttons have distinct meanings, however.

- Clicking Left inspects the object in the bottom pane, pushing the previous objects up.

- Clicking Middle inspects the object but leaves the source (namely, the object being inspected in the window in which the mouse was clicked) in the second pane from the bottom.

- Clicking Right tries to find and inspect the function associated with the selected object (for example, the function binding if a symbol was selected).

Inspection Pane Display

The information that the Inspector displays depends upon the type of the object:

Symbol	The name, value, function, property list, and package of the symbol are displayed. All but the name and the package are modifiable.
List	The list is displayed ground by the system grinder. Any piece of substructure is selectable, and any **car** or atom in the list can be modified.

Instance	The flavor of the instance, the method table, and the names and values of the instance-variable slots are displayed. The instance-variables are modifiable.
Hash Table	The flavor of the hash table, the method table, and the names and values of the instance-variable slots of the hash table are displayed, followed by the key/value pairs for the entries of the hash table. The value for a given key is modifiable.
Closure	The function, and the names and values of the closed variables are displayed. The values of the closed variables are modifiable.
Named structure	The names and values of the slots are displayed. The values are modifiable.
Array	The leader of the array is displayed if present. For one-dimensional arrays, the elements of the array are also displayed. The elements are modifiable.
Compiled code object	The disassembled code is displayed.
Select Method	The keyword/function pairs are shown, in alphabetical order by keyword. The function associated with a keyword is settable via the keyword.
Stack Frame	This is a special internal type used by the Display Debugger. It is displayed as either interpreted code (a list) or as a compiled code object with an arrow pointing to the next instruction to be executed.

Special Characters Recognized by the Inspector

Some special keyboard characters are recognized when not in the middle of typing in a form.

`c-Z`	Exits and deactivates the Inspector.
`BREAK`	Runs a break loop in the typeout window of the bottom-most inspection pane.
`ESCAPE`	Reads a form, evaluates it, and prints the result instead of inspecting it.

Examining a Compiled Code File

To examine a compiled code file, use **si:unbin-file**. The output format from **unbin-file** includes disassembled code for any compiled functions in the compiled code file.

si:unbin-file *file* &optional *outfile* *Function*

Converts the compiled code file *file* to a human-readable file, which you can optionally specify. It includes disassembled code for any compiled functions in the compiled code file.

The Peek Program

You start up Peek by pressing `SELECT P`, by using the Select Activity Peek command, or by evaluating **(zl:peek)**.

Overview of Peek

The Peek program gives a dynamic display of various kinds of system status. When you start up Peek, a menu is displayed at the top, with one item for each system-status 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. Pressing one of the keyboard keys as listed in the Help message also switches Peek to the mode associated with that key. The Help message is a Peek mode; Peek starts out in this mode.

Pressing the `HELP` key displays the Help message.

The `Q` command exits Peek and returns you to the window from which Peek was invoked.

Most of the modes are dynamic: they update some part of the displayed status periodically. The interval between updates is 20 seconds, but if you want more or less frequent updates, you can set it using the `Z` command. Pressing *n*`Z`, where *n* is some number, sets the time interval between updates to *n* seconds. Using the `Z` command does not otherwise affect the mode that is running.

Some of the items displayed in the modes are mouse sensitive. These items, and the operations that can be performed by clicking the mouse on them, vary from mode to mode. Often clicking the mouse on an item gives you a menu of things to do to that object.

The Peek window has scrolling capabilities, for use when the status display is longer than the available display area. `SCROLL` or `c-V` scrolls the window forward (towards the bottom), `m-SCROLL` or `m-V` scrolls it backward (towards the top).

As long as the Peek window is exposed, it continues to update its display. Thus a Peek window can be used to examine things being done in other windows in real time.

zl:peek &optional (*character* **'tv:p**) *Function*

Displays various information about the system, periodically updating it. It has several modes, which are entered by pressing a single key that is the name of the mode. The initial mode is selected by the argument, *character*. If no argument is given, **zl:peek** starts out by explaining what its modes are.

The Help message consists of the following:

```
This is the Peek utility program.  It shows a continually
updating display of status about some aspect of the system,
depending on what mode it is in.  The available modes are listed
below.  Each has a name, followed by a single character in
parentheses, followed by a description.  To put Peek into a given
mode, click on the name of the mode, in the command menu above.
Alternatively, type the single character shown below.

Processes (P):
   Show all active processes, their states, priorities, quanta,
   idle times, etc.

Areas (A):
   Show all the areas in virtual memory, their types, allocation, etc.

File System (F):
   Show all of our connections to various file servers.

Windows (W):
   Show all the active windows and their hierarchical relationships.

Servers (S):
   Show all active network servers and what they are doing.

Network (N):
   Show all local networks, their state and active connections, and
   network interfaces.

Help (HELP):
   Explain how this program works.

Quit (Q):
   Bury PEEK window, exiting PEEK

Hostat (H):
   Show the status of all hosts on the Chaosnet

There are also the following single-character commands:
Z (preceded by a number): Set the amount of time between updates,
  in seconds.  By default, the display is updated every twenty seconds.
<SPACE>: Immediately update the display.
```

The commands `P`, `A`, `F`, `W`, `S`, `H`, and `N` each place you in a different Peek mode, to examine the status of different aspects of Genera.

Peek Modes

Processes (P)

In Processes mode, invoked by pressing P or by clicking on the [Processes] menu item, you see all the processes running in your environment, one line for each. The process names are mouse sensitive; clicking on one of them pops up a menu of operations that can be performed:

Arrest (or Un-Arrest)	Arrest causes the process to stop immediately. Unarrest causes it to pick up where it left off and continue.
Flush	Causes the process to go into the state Wait Forever. This is one way to stop a runaway process that is monopolizing your machine and not responding to any other commands. A process that has been flushed can be looked at with the Debugger or Inspector and can be reset.
Reset	Causes the process to start over in its initialized state. This is one way to get out of stuck states when other commands do not work.
Kill	Causes the process to go away completely.
Debugger	Enters the Debugger to look at the process.
Describe	Displays information about the process.
Inspect	Enters the Inspector to look at the process.

See the section "Introduction to Processes".

Areas (A)

Areas mode, invoked by pressing A or by clicking on [Areas], shows you information about your machine's memory. The first line is hardware information: the amount of physical memory on the machine, the amount of swapping space remaining in virtual memory, and how many wired pages of memory the machine has. The following lines show all the areas in virtual memory, one line for each. For each area you are shown how many regions it contains, what percentage of it is free, and the number of words (of the total) in use. Clicking on an area inserts detailed information about each region: its number, its starting address, its length, how many words are used, its type, and its GC status. See the section "Areas".

Meters (M)

Meters mode, invoked by pressing M or by clicking on [Meters], shows you a list of all the metering variables for storage, the garbage collector, Zwei sectionization, netboot and the disk. There are two types of meters:

Timers	Timers have names that start with ***ms-time-** and keep a total of the milleseconds spent in some activity.

Counts	Counts have names that start with ***count-** and keep a running total of the number of times some event has occurred.

The garbage collector meters fall into two groups according to which part of the garbage collector they pertain to: the scavenger or the transporter. See the section "Theory of Operation of the GC Facilities".

File System (F)

File System mode, invoked by pressing `F` or by clicking on [File System], provides information about your network connections for file operations. For each host the access path, protocol, user-id, host or server unit number, and connection state are listed. For active connections information about the actual packet flow is also given. The various items are mouse sensitive. For hosts, you can get hostat information, do a file reset, log in remotely, find out who is on the remote machine, and send a message to the machine. You can reset, describe, or inspect data channels, and close streams.

Resetting an access path makes the server on a foreign host go away, which might be useful to free resources on that host or if you suspect that the server is not working correctly.

Windows (W)

Windows mode, invoked by pressing `W` or clicking on [Windows], shows you all the active windows in your environment with the panes they contain. This allows you to see the hierarchical structure of your environment. The items are mouse sensitive. Clicking on a window name pops up a menu of operations that you can perform on the window.

Servers (S)

Clicking on [Servers] or pressing `S` puts Peek in Servers mode. If your machine is a server (for example, a file server), Servers mode shows the status of each active server.

Network (N)

Network mode, invoked by pressing `N` or by clicking on [Network], shows information about the networks connected to your machine. For each network there are three headings for information:

Active connections	The data channels that your machine has opened to another machine or machines on the network.
Meters	Information about the data flow (packets) between your machine and other machines on the network.

Routing table	A list of all the subnets and for each the route to take to send packets to a host on that subnet.

To view the information under one of these headings, you click on the heading. The hosts and data channels in the list of active connections are mouse sensitive. For hosts, you can get hostat information, do a file reset, login remotely, find out who is on the remote machine, and send a message to the machine. You can reset, describe, or inspect data channels.

Information about the hardware network interface is also displayed, as well as metering variables for the networks.

Hostat (H)

Clicking on [Hostat] or pressing `H` starts polling all the machines connected to the local network. For each host on the network a line of information is displayed. Those machines that do not respond to the poll are marked as "Host not responding". You terminate the display by pressing `c-ABORT`.

Help and Quit

Clicking on the [Help] menu item or pressing `HELP` displays the help information that is displayed when Peek is selected the first time.

Clicking on [Quit] or pressing `Q` buries the Peek window and returns you to the window from which you invoked Peek.

Files

Naming of Files

A Symbolics computer generally has access to many file systems. While it can have its own file system on its own disks, a community of Symbolics users often has many shared file systems accessible by any of the Symbolics computers over a network. These shared file systems can be implemented by any computers that are capable of providing file system service. A *file server computer* might be a special-purpose computer that does nothing but service file system requests from computers on a network, or it might be an existing timesharing system.

Programs, at the behest of users, need to use names to designate files within these file systems. The main difficulty in dealing with names of files is that different file systems have different naming conventions and formats for files. For example, in the UNIX system, a typical name looks like:

```
/usr2/george/foo.bn
```

In this example, /usr2/george is the *directory name*, foo is the *file name* and bn is the *file type*. However, in TOPS-20, a similar file name is expressed as follows:

```
    PS:<GEORGE>FOO.BIN
```

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists; in fact, new formats could be added in the future, and existing programs should retain their abilities to manipulate files in a system-independent fashion.

The functions, flavors, and messages described in this chapter exist to solve this problem. They provide an interface through which a program can deal with files and manipulate them without depending on their syntax. This lets a program deal with multiple remote file systems simultaneously, using a uniform set of conventions.

Pathnames

All file systems dealt with by the Symbolics computer are mapped into a common model, in which files are named by a conceptual object called a *pathname*. The Symbolics computer system, in fact, represents pathnames by objects of flavor **pathname**, and the flavors built upon it. A pathname always has six conceptual components, described below. These components provide the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. This mapping is described elsewhere for each file system. See the section "The Character Set".

The following are the conceptual components of a pathname. They will be clarified by examples below.

Host	The computer system, the machine, on which the file resides.
Device	Corresponds to the "device" or "file structure" concept in many host file systems. Often, it designates a group of disks, or removable storage media, or one of several different media of differing storage densities or costs.
Directory	An organizational structure in which files are "contained" on almost all file systems. Files are "stored in", or "reside in" directories. The directories have names; the files' names are only valid within the context of a given directory. Some systems (*hierarchical* file systems) allow directories to be contained in other directories; others do not.
Name	The name of a group of files that can be thought of as conceptually the "same" file. In many systems, this is the "first name" of the file. For instance, source and object files for the same program generally have the same *name*, but differing *type*.
Type	Corresponds to the "filetype" or "extension" concept in many host file systems. This usually indicates the kind of data stored in the file, for example, binary object code, a Lisp source program, a FORTRAN source program, and so forth.

Version	Corresponds to the "version number" concept in many host file systems. Some systems implement this concept, others do not. A version number is a number, part of the conceptual name of the file, that distinguishes succeeding versions of a file from each other. When you write out a file on such a file system, you are not modifying the file on the host computer but writing a *new version*, that is, one with a higher version number, automatically. The Symbolics computer system allows a version component of "newest" or "oldest", represented by the keyword symbols **:newest** and **:oldest**, respectively, to designate "the newest (oldest) version of the file, whichever that might be".

As an example, consider a TOPS-20 user named "George", who writes a Lisp program that he thinks of as being named "conch". If George uses the TOPS-20 host named FISH, the source for his program might be in a file on the host FISH with the following name:

```
<GEORGE>CONCH.LISP.17
```

In this case, the host is FISH, the device would be some appropriate default, and the directory would be <GEORGE>. This directory would probably contain a number of files related to the "conch" program. The source code for this program would live in a file with name CONCH, type LISP, and versions 1, 2, 3, and so on. The compiled form of the program would live in a file named CONCH with type BIN.

Now suppose George is a UNIX user, using the UNIX host BIRD. The source for his program would probably be in a file on the host BIRD with the following name:

```
/usr2/george/conch.l
```

In this case, the host is BIRD, and the directory would be /usr2/george. This directory would probably contain a number of files related to the "conch" program. The source code for this program would live in a file with name conch, type l. The compiled form of the program would live in a file named conch, with type bn. There are no version numbers on UNIX.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; a pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with a version of "newest" will refer to the same file as a pathname with the same components except a certain number as the version. In systems with links, multiple file names, logical devices, and so forth, two pathnames that look quite different can turn out to designate the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

Basic Use of the Pathname System

The pathname system can be very easy to use if you know a few simple techniques. It often seems that there are many different ways to do anything, and that only one of these is right for any circumstance, but most of these features only exist for special needs. This section shows you how to easily do some of the simple things.

Getting a Filename From the User

The simplest and most common application for using a pathname is simply to read or write a file. For example, a program to do some *very* simple processing of a database (it reads the file and ignores it):

```
(defun process-example-database (database-pathname)
  (with-open-file (database-stream database-pathname)
    (format t "~&Ignoring database ~A ..." (send database-stream :truename))
    (stream-copy-until-eof stream #'si:null-stream)
    (format t " ignored.~%")))
```

This simple example is adequate for a program interface, but for a user, it is rather awkward. You must supply all components of the pathname, plus the quotation marks around the strings. Also, you have no completion available. In this example, you do not have to parse the pathname; **open** will do that for you. (Sometimes we are not be so lucky).

Your job can be made easier by providing a function to read a pathname and pass it to **process-example-database**. To do this, **zl-user:accept** is used.

In our first version, we just ask the user for the pathname.

```
(defun run-example ()
  (let ((pathname (prompt-and-read :pathname "Where is your database? ")))
    (process-example-database pathname)))

Where is your database?  Y:>user>databases>dummy.database
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

zl-user:accept does much of what we are looking for. It provides the following:

- Parsing
- Completion
- Merging with defaults

In this case, we supplied no default, so the "default default", **fs:*default-pathname-defaults*** is used. But this default is not very helpful, because it is not visible; it could even be confusing if you expected one default and got another. Good practice dictates telling the user what the default is. **prompt-and-read** makes this easy with the **:visible-default** suboption to **:pathname**, **:pathname-or-nil**, and **:pathname-list**.

```
(defun run-example ()
  (let ((pathname (prompt-and-read
                    `(:pathname :visible-default ,fs:*default-pathname-defaults*)
                    "Where is your database? ")))
    (process-example-database pathname)))
Where is your database?  (Default Y:>user>foo.lisp) databases>dummy.database
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

Now that you can see the defaults, you can make use of them. Note that in the above example, you did not have to type the "Y:>user>", because the default was available.

Tailoring Pathname Defaults

fs:*default-pathname-defaults* is a global default, with nothing particularly appropriate to any specific application. Often, when an application is writing or reading a file, it knows more about the file than is implied by **fs:*default-pathname-defaults***. This information can be used to help prompt the user for a suitable filename and help reduce the amount of typing needed to specify a suitable filename.

For example, consider our example of reading a database. (See the section "Getting a Filename From the User".) In this example, we are just prompting for the filename and ignoring the actual database.

```
(defun run-example ()
  (let ((pathname (prompt-and-read
                    `(:pathname :visible-default ,fs:*default-pathname-defaults*)
                    "Where is your database? ")))
    (process-example-database pathname)))
Where is your database?  (Default Y:>user>foo.lisp) databases>dummy.database
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

First, if we are going to seriously use our own special file type, we need to define the type so that it can be used successfully on different systems. See the special form **fs:define-canonical-type**.

```
(fs:define-canonical-type :database "DATABASE"
  ((:vms :vms4) "DBS")
  (:unix "DB"))
```

Now this type can be used as the default type for our example databases.

```
(defun run-example ()
  (let* ((default (fs:default-pathname fs:*default-pathname-defaults*
                                       nil              ;Host
                                       :database))      ;Type
         (pathname (prompt-and-read `(:pathname :visible-default ,default)
                                    "Where is your database?~%")))
    (process-example-database pathname)))

Where is your database?  (Default Y:>user>foo.database) databases>dummy
Ignoring Y:>user>databases>dummy.database.7 ... ignored.
```

More About Defaults

Most simple programs use **fs:*default-pathname-defaults*** as the source for their defaults. However, as a program makes more use of pathname reading and defaults, there are some things we can do to make things easier for the user.

- Provide a default based on other files in an operation, for example, defaulting an output file pathname from the input file.

- Provide "sticky" defaults, where the new default is based on the last file the user gave.

- Provide a default based on the current context, as in "pathname of the current buffer" in Zmacs.

Defaulting An Output File Pathname From An Input File

Perhaps the most common defaulting situation is that of defaulting an output file pathname from the input file. Usually, the output file differs from the input file only in file type and version, and we would like to have users provide explicit information only when they want something differ from the usual case.

```
(defun my-compile-file (input-file output-file)
  (format t "~&Compiling ~A into ~A.~%"
          input-file output-file)
  (compiler:compile-file input-file output-file))
```

```
(defun comp-it ()
  (let* ((input-default (fs:default-pathname nil nil :lisp :newest))
         (input-file (prompt-and-read
                       '(:pathname :visible-default ,input-default)
                       "Input file: "))
         (output-default (fs:default-pathname input-file nil :bin :newest))
         (output-file (prompt-and-read
                        '(:pathname :visible-default ,output-default)
                        "Output file: ")))
    (my-compile-file input-file output-file)))
```

The above example works well for single files, but it does not handle wildcards. To handle wildcards, we need to introduce the use of **:translate-wild-pathname** and **fs:directory-link-opaque-dirlist**. **:translate-wild-pathname** does the work of interpreting how a given input file is to be mapped to its corresponding output file, and **fs:directory-link-opaque-dirlist** takes care of finding all the input files.

Note that we use **fs:directory-link-opaque-dirlist** rather than **fs:directory-list**. In general, this is necessary whenever the **:translate-wild-pathname** message is used. **:translate-wild-pathname** expects the input pathname to match the input pattern. **fs:directory-list**, in the presence of directory links or VAX/VMS logical devices, can have a different directory or a different device.

If the input pattern has wildcards in its directory component, **fs:directory-link-opaque-dirlist** currently does no better than **fs:directory-list**. This is a difficult problem still under investigation.

```
(defun comp-one-file (input-file-pattern output-file-pattern input-file)
  (let ((output-file (send input-file-pattern :translate-wild-pathname
                           output-file-pattern input-file)))
    (my-compile-file input-file output-file)))

(defun comp-files ()
  (let* ((input-default (fs:default-pathname nil nil :lisp :newest))
         (input-pattern (prompt-and-read
                          '(:pathname :visible-default ,input-default)
                          "Input file: "))
         (output-default (fs:default-pathname input-file nil :bin :newest))
         (output-pattern (prompt-and-read
                           '(:pathname :visible-default ,output-default)
                           "Output file: ")))
    (if (not (send input-file :wild-p))
        (comp-one-file input-pattern output-pattern input-pattern)
      (loop for (file) in (cdr (fs:directory-link-opaque-dirlist
                                 input-pattern :fast))
            do (comp-one-file input-pattern output-pattern file)))))
```

Note that in the above example, we just call **comp-one-file** directly if the input pathname is not wild. While it is not strictly necessary to do this (**fs:directory-link-opaque-dirlist** works on non-wildcard pathnames), it does eliminate an unneeded operation.

Sticky Pathname Defaults

Often, when a single command or a related set of commands are to be repeated, the next command should operate on a file related to the one the current command is operating on. In this case, it would be most convenient for the default to be the previous pathname. This is called *sticky defaulting*.

For example, consider a simple user-written tool to either show or delete files.

```
(defun show-or-delete ()
  (loop with default = (fs:default-pathname)
        for ch = (prompt-and-read :character "Cmd>")
        do (multiple-value-bind (prompt function)
               (selector char-equal ch
                 (#\S (values "Show File" #'viewf))
                 (#\D (values "Delete File" #'deletef))
                 (#\Q (return nil))
                 (#\Help (format t "~&S = Show File~@
                                    D = Delete File~@
                                    Q = Quit~%")
                  (values nil nil))
                 (otherwise
                  (tv:beep)
                  (format t "~&~:C is an unknown command.~%" ch)))
             (when prompt
               (let ((file (prompt-and-read
                             `(:pathname :visible-default ,default)
                             prompt)))
                 ; The following is done for us by prompt-and-read
                 ;(setq default (fs:merge-pathnames file default))
                 (funcall function file))))))
```

Each time around the loop, when you specify a file, it is remembered to serve as the default the next time around. Note the commented out **(setq default (fs:merge-pathnames file default))**. This isn't needed in this example, since **prompt-and-read** does this for us, but if we were reading pathnames via some other mechanism, it is important to keep the default as a fully specified pathname. Otherwise, the second time around the loop, we could end up with defaults like "Q:", which is not of much use if you are then forced to type all the components of the pathname and may get an error if you do not.

If you wish to use a default such as this and not keep it in a local variable, you should use a defaults alist. This serves as a registered place to remember a pathname, so that if the world is moved to another site, it can be reset. Defaults alists can be passed to **fs:default-pathname** to extract a fully-merged default. See the function **fs:set-default-pathname**. See the function **fs:make-pathname-defaults**.

Pathname Defaulting From the Current Context

Often, an application program involves the user working on a single context for an extended time. For example, in the editor, the user is working on a single named buffer. In the font editor, the user is working on a single named font.

Often, the object being worked on was read in from a file. This file can serve as a default for further file operations, such as listing the directory, or resaving the object. Consider a picture editor, which lets the user edit multiple pictures, as the Zmacs editor lets the user edit multiple buffers. This picture editor stores its files in .BIN files.

```
(defflavor picture (name
                    (pathname sys:fdefine-file-pathname)
                    (array (make-array '(100. 100.) :type 'art-1b)))
           ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(defvar *pictures* nil
       "List of pictures being edited")

(defvar *current-picture* nil)

(defvar *picture-defaults* (fs:make-pathname-defaults))

(defun add-picture (picture)
  (setq *pictures* (del #'(lambda (p1 p2)
                             (string-equal (send p1 :name) (send p2 :name)))
                        picture
                        *pictures*))
  (push picture *pictures*)
  (setq *current-picture* picture))

(defmethod (picture :fasd-form) ()
  `(make-instance ',(typep self)
                  :name ',name
                  :array ',array))
```

```
(defun picture-default-pathname (&key type (version :newest))
  (let ((bare-default (fs:default-pathname *picture-defaults*
                                            nil type version))
        (path (when *current-picture*
                (send *current-picture* :pathname))))
    (if (not *current-picture*)
        bare-default
      (if path
          (setq path (fs:merge-pathnames path bare-default version))
        ;; A new picture, so no pathname.  Let's make a guess from the name.
        (let ((name (send *current-picture* :name)))
          (setq path
                (condition-case ()
                    (fs:merge-pathnames name bare-default version)
                  ;; If name isn't parsable, just use the bare default.
                  (error bare-default)))))
      path)))

(defun com-create-picture ()
  (let ((name (prompt-and-read :string "Picture name: ")))
    (add-picture (make-instance 'picture :name name))))

(defun com-save-picture ()
  (let* ((default (picture-default-pathname :type :bin))
         (file (prompt-and-read
                 `(:pathname :visible-default ,default)
                 "Save to picture file: ")))
    ;; Remember the pathname given, so the next time we
    ;; get a new picture, we can have a better default.
    (fs:set-default-pathname file *picture-defaults*)
    (sys:dump-forms-to-file
      file
      `((add-picture ',*current-picture*)))))
```

In this example, **picture-default-pathname** computes the default. If the current picture has a file associated with it, that serves as the default. If there is no pathname with the current picture, we attempt to make a pathname using the name. If that fails (or if there is no current picture), we just use the bare default.

Finally, the pathname we read is remembered, so the next time a default is needed for a new picture, we will have a more recent default.

Note that when the picture is loaded, **sys:fdefine-file-pathname** is used to get the file being loaded. This works well when the file being loaded is a .bin file, since **zl:load** binds this variable. However, in other situations, you need to make other arrangements to set the pathname.

Lozenge Character is Reserved in Pathnames

The lozenge character (◊) is a reserved character in pathnames.

Host Determination In Pathnames

Two important operations of the pathname system are *parsing* and *merging*. Parsing is the conversion of a string, which might have been typed by the user when asked to supply the name of a file, into a pathname object. This involves finding out for which host the pathname is intended, using the file name syntax conventions of that host to parse the string into the standard pathname components, and constructing such a pathname. Merging is the operation that takes a pathname with missing components and supplies values for those components from a set of defaults.

Since each kind of file system has its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and determining the value of each component. The parsers, therefore, all work differently. How does the parsing operation know which parser to use? It determines for which host the pathname is intended, and uses the appropriate parser. A filename character string can specify a host explicitly, by having the name of the host, followed by a colon, at the beginning of the string, or it can assume a default, if there is no host name followed by a colon at the beginning of the string.

Here is how the pathname system determines for which host a pathname being parsed is intended. The first colon in a pathname being parsed *always* delimits the host name. You can also enter pathname strings that are for a specific host and do not contain any host name. In that case, a *default host* is used. Normally, the identity of the default host is displayed to the user entering a pathname. See the section "Pathname Defaults and Merging".

However, pathnames can have colons in them that do not designate hosts, such as filenames constructed from clock times, and the like. Some systems use the colon character to delimit devices. This creates a problem in parsing such pathnames. See the function **fs:parse-pathname**. The standard Symbolics computer user interface does not use such pathnames, but they can be used by other programs, particularly those that deal with files whose format is defined by a foreign operating system.

The rule for parsing file names containing colons is, again, that any string used before a colon is *unconditionally* interpreted as a file computer. If the string cannot be interpreted as a host, an error is signalled.

If you must type a pathname that has an embedded colon *not* meaning a host, you omit the host and place a colon at the beginning of the string. This "null host" tells the parser that it should *not* look further for a colon, but instead assume the host from the defaults. Examples:

- SS:<FOO>BAR refers to a host named "SS". :SS:<FOO>BAR refers to no explicit host; if parsed relative to a TOPS-20 default, "SS" probably refers to a device.

- 09:25:14.data refers to a host named "09". :09:25:14.data refers to no explicit host.
- AI: COMMON; GEE WHIZ refers to a host named "AI".
- AI: ARC: USERS1; FOO BAR refers to a host named "AI". "ARC" is the name of a device in the ITS operating system.
- EE:PS:<COMMON>GEE.WHIZ.5 specifies host EE (TOPS-20).
- PS:<COMMON>GEE.WHIZ.5 specifies a host named PS, which is almost certainly not what is intended! The user probably intended the "PS" device on some TOPS-20 host.
- :PS:<COMMON>GEE.WHIZ.5, assuming that the default host is some TOPS-20, specifies a device named "PS" on that host.

There are a few "pseudohost" names, which are recognized as host names even though they are not actually the names of hosts:

"local"	This pseudohost name always refers to the local file system (LMFS) of the machine that you are using. It does not matter whether or not a local file system actually exists on that machine; an attempt will be made to reference it. "Local" is always equivalent to the name of the local host.
"FEP"	This pseudohost name always refers to a FEP (front-end processor) file system on the machine you are using, specifically, the one on the disk unit from which the system was booted.
"FEP*n*"	This pseudo name always refers to a FEP file system on the machine you are using. The single digit *n* specifies the disk unit number; there is a separate FEP file system on each drive. This can access the boot unit, or any other disk unit, when multiple units are present.
"*host*|FEP*n*"	*host* must be a valid host name. This pseudohost name refers to a FEP file system on a remote Symbolics computer. The syntax "*host*|FEP" is not acceptable: you cannot access the "boot unit" of a remote machine in this fashion. You must know the disk unit number. The disk unit number of a host having only one disk unit is **0**.

If the string to be parsed does not specify a host explicitly, the parser assumes that some particular host is the one in question, and it uses the parser for that host's file system. The optional arguments passed to the parsing function (**fs:parse-pathname**) tell it which host to assume.

Interning of Pathnames

Pathnames, like symbols, are *interned*. This means that there is only one pathname object with a given set of components. If a character string is parsed into components, and some pathname object with exactly those components already exists, then the parser returns the existing pathname object rather than creating a new one. The main reason for this is that a pathname has a property list. See the section "Property Lists". The system stores properties on pathnames to remember information about the file or family of files to which that pathname refers. (In fact, some of the *properties* stored on a generic pathname come from the file's *attribute* list when the file is edited or loaded, so they can be retrieved later without having to perform I/O on the file.) So you can parse a character string that represents a filename, and then look at its property list to get various information known about that pathname. The components of a pathname are never modified once the pathname has been created, just as the print name of a symbol is never modified. The only thing that can be modified is the property list.

When using property lists of pathnames, you have to be very careful which pathname you use to hold properties, in order to avoid a subtle problem: many different pathnames can refer to the same file, because of the **:newest** component, file system links, multiple naming in the file system, and so on. If you put a property on one of these pathnames because you want to associate some information with the file itself, somebody else might look at another pathname that refers to the same file, and not find the information there. If you really want to associate information with the file itself rather than some particular pathname, you can get a canonical pathname for the file by using the **:truename** message to a stream opened to that file. See the message **:truename**. You might also want to store properties on "generic" pathnames. See the section "Generic Pathnames".

Printing Pathnames

A pathname can be converted back into a string, which is in the file name syntax of its host's file system. Although such a string (the *string for host*) can be produced from a pathname (by sending it the **:string-for-host** message), we discourage this practice. The Genera user interface prefers a string called the *string for printing*, which is the same as the string for host, except that it is preceded by the host name and a colon. This leaves no ambiguity about the host on which the file resides, when seen by a user. It is also capable of being reparsed, unambiguously, back into a pathname. **prin1** of a pathname (**~S** in **zl:format**) prints it like a Lisp object (using the usual "#<" syntax), while **princ** of a pathname (**~A** in **zl:format**) prints the string for printing. The **string** function, applied to a pathname, also returns the string for printing.

Not all the components of a pathname need to be specified. If a component pathname is missing, its value is **nil**. Before a file server can utilize a pathname to manipulate or otherwise access a file, all the pathname's missing components must be filled in from appropriate defaults. Pathnames with missing components are nevertheless often passed around by programs, since almost all pathnames typed by users do not specify all the components explicitly. The host is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to explicitly designate a host. Every pathname has a host attribute, even if the string that was parsed to create it did not specify one explicitly.

All pathname parsers support the cross-system convention that the double-shafted arrow character (↔) can be used to specify a null directory, name, type, or version component explicitly. Thus, for LMFS or TOPS-20, you can type the following:

```
↔.↔.5
```

This example specifies a version of 5, but no name or type. This is useful when typing against the default and attempting to change just the version of that default.

The keyword symbol **:unspecific** can also be a component of a pathname. This means that the component is not meaningful on the type of file system concerned. For example, UNIX pathnames do not have a concept of "version", so the version component of every UNIX pathname is **:unspecific**. When a pathname is converted to a string, **nil** and **:unspecific** both cause the component not to appear in the string. The difference occurs in the merging operation, where **nil** is replaced with the default for that component, while **:unspecific** is left alone.

The special symbol **:wild** can also be a component of a pathname. This is only useful when the pathname is being used with a directory listing primitive such as **fs:directory-list** or **fs:all-directories**, where it means that this pathname component matches anything. See the function **fs:directory-list**. The printed representation of a pathname usually designates **:wild** with an asterisk; however, this is host-dependent.

:wild is one of several possible *wildcard* components, which are given to directory-listing primitives to filter file names. Many systems support other wildcard components, such as the string "foo*". This string, when supplied as a file name to a directory list operation on any of several system types, specifies all files whose name starts with "foo". In other contexts, it might not represent a wildcard at all. The component **:wild** matches all possible values for any component for which it appears. Other wildcard possibilities for directories exist, but they are more complicated, and are explained elsewhere. See the section "Values of Pathname Components". See the section "Directory Pathnames and Directory Pathnames as Files".

Pathname Translation

A translations file contains the form **fs:set-logical-pathname-host** and a translations list. The list describes logical pathnames by providing their corresponding physical pathnames. Each logical/physical pathname pair is called a translation pair.

The Logical Pathname Translations File

Here is a sample translations file (note the translations list, containing three translation pairs, following the `:translations` keyword at the end of the file):

```
;;; -*- Mode: LISP; Package: FS; Syntax: ZetaLisp; Base: 10; -*-
(fs:set-logical-pathname-host "SYS"
   :physical-host "ACME-YUKON"
   :translations '(("SYS:DOC;**;*.*.*" "ACME-YUKON:>sys>doc>**>*.*.*")
                   ("SYS:FONTS;**;*.*.*" "ACME-RIVERSIDE:>sys>fonts>**>*.*.*")
                   ("SYS:**;*.*.*" "ACME-QUABBIN:>sys>**>*.*.*")))
```

In this sample, the logical pathname `SYS:DOC;` (and all its inferior directories) maps to the physical host ACME-YUKON. If this translations file were loaded, the logical pathname,

```
SYS:DOC;SITE;SITE7.SAB.NEWEST
```

would resolve to the file described by this physical pathname:

```
ACME-YUKON:>sys>doc>site>site7.sab.newest
```

Likewise, the logical pathname, `SYS:FONTS;` (and all its inferior directories) would map to the physical host ACME-RIVERSIDE, and the logical pathname,

```
SYS:FONTS;TV;CPTFONTB.BFD.NEWEST
```

would resolve to the file described by this physical pathname:

```
ACME-RIVERSIDE:>sys>fonts>tv>cptfontb.bfd.newest
```

All other logical pathnames beginning with `SYS:` (and all their inferior directories) would map to the physical host ACME-QUABBIN. For example,

```
SYS:FLAVOR;CTYPES.LISP.NEWEST
```

would resolve to the file described by this physical pathname:

```
ACME-QUABBIN:>sys>flavor>ctypes.lisp.newest
```

The Logical Pathname Translation Process

There are two phases to the translation process. In the first phase, using the **:pathname-match** message, the pathname resolver matches a logical pathname (the pathname to be translated) against the logical pathnames listed successively in the translations file. Once the pathname resolver finds a match, it uses the appropriate logical/physical pathname pair from the list.

Note: Because the translation list is searched in sequence, it should provide the most specific pathnames first, and the most general pathname last.

In the second phase, the pathname resolver processes the selected translation pair according to translation rules. There are three sets of translation rules for each logical host:

Permanent	The permanent translation rules are special purpose rules that cannot be overridden. They provide for such things as the translation of patch file pathnames. This set is searched first.
Site	The site translation rule is determined by the Site-Directory attribute for each object of class "site" in the namespace. A site's translation rule cannot be overridden. This set is searched second.

Supplied	The normal, supplied translation rules are provided by the author of the software using the logical host. This set is searched third.

Additionally, the pathname resolver uses these host-independent rules:

Global	This set is not currently used for anything, but it is provided for future extension.
Default	This is the **:translate-wild** rule, which uses **:translate-wild-pathname-reversible**. This rule is used when no other rule is applicable.

The second phase (in which the the pathname resolver processes the selected translation pair according to translation rules) is potentially more complex. In its simplest form, the pathname resolver uses the default rule. Before using the default rule, though, the pathname resolver searches for a more suitable one.

The default rule produces a physical pathname by sending the **:translate-wild-pathname-reversible** message to the logical pathname, where the first element of the translation pair is the source pattern, and the second element of the translation pair is the target pattern. For more information about source and target patterns:

See the section "Wildcard Pathname Mapping". See the section "Wildcard Directory Mapping". See the section "Reversible Wildcard Pathname Translation".

Logical Translations to Multiple Physical Hosts

A logical host can translate to more than one physical host when the translations list given to **fs:set-logical-pathname-host** contains explicit pointers to more than one host.

For example:

```
(fs:set-logical-pathname-host "SYS"
  :translations '(("SYS:DOC;**;*.*.*" "ACME-LISPM:>Rel-8-0>doc>**>*.*.*")
                  ("SYS:**;*.*.*" "ACMEVAX:SYMBOLICS:[REL8-0...]*.*.*"))
  :no-translate nil)
```

Note: it is not necessary to specify the **:physical-host** argument to **fs:set-logical-pathname-host** as long as host names are specified in the translations list. If a **:physical-host** argument is specified, however, it serves as the default.

Translation Rules

The logical host SYS uses heuristics that eliminate characters illegal in VAX/VMS file specifications and deal with filename-length limitations on foreign hosts.

For example, some filenames can be shortened without changing their meanings:

```
sys:io;pathnm-cometh.lisp
```

might translate to

```
    acmevax:symbolics[rel7-I.io]pthnmcmth.lsp
```

on a VAX/VMS physical host.

The system does not allow two logical pathnames to translate to the same place. An error is signalled when the system attempts to translate a logical pathname to a physical pathname already found as the result of a logical-pathname translation (for example, if a typographical error is made when a user types in a logical pathname).

However, when **:no-translate nil** is used in the **fs:set-logical-pathname-host** form, the system translates all its logical pathnames when setting the logical system host (thus eliminating the possibility of incorrect translations being entered by mistake). For more information about this, see the function **fs:set-logical-pathname-host**.

Reversible Wildcard Pathname Translation

Reversible wild pathname translation is a special version of wild pathname translation. The difference between wild pathname translation and reversible translation lies in their treatment of a target wildcard pattern consisting solely of *. In regular translation, a target pattern of **:wild** causes the source component to be copied verbatim. This is a useful user-interface feature, but it causes dropping of information and resultant noninvertibility of the transformation. In reversible mapping, this feature is not present. Logical pathname translation and back-translation is done in this mode.

Example:

	Source	*Source*	*Target*	
Type	*pattern*	*instance*	*pattern*	*Result*
Regular	foo*	foolish	*	foolish
Reversible	foo*	foolish	*	lish
Either	*	bar	foo-*	foo-bar

Note that the inverse translation of foo-bar to bar cannot be accomplished under regular translation.

Defining a Translation Rule

Translation rules are defined using the **fs:set-logical-pathname-host** function, using the **:rules** or **:site-rules** argument. (The other rule tables are not normally set by the user). These arguments should be an a list of system type and translation rule specifications.

((:vms *VMS rule specifications ...*)
 (:vms4 *VMS4 rule specifications ...*)
 (:unix *UNIX rule specifications ...*))

Each rule specification consists of a *pattern*, a *rule type*, and optional arguments, as in the following example.

```
("PICTURE:EDITOR;LINE-DRAWING-COMMANDS.*.*" :vms-new-pathname :name "LINECMNDS")
```

In this example, **"PICTURE:EDITOR;LINE-DRAWING-COMMANDS.*.*"** is the pattern, **:vms-new-pathname** is the rule type, and **:name** and **"LINECMNDS"** form a keyword/value pair of arguments to the **:vms-new-pathname** rule type.

Normally, translation rules are defined in the system definition file before a **defsystem** form, so that the rules are loaded before they are needed. If you wish to override the translation rules provided either by Symbolics or another vendor, you can use the **:site-rules** argument to the call to **fs:set-logical-pathname-host**, normally placed in the translation file.

The following sections describe the various rule types that exist and their arguments.

:translate-wild &rest options *Translation rule*

The default translation rule's type is **:translate-wild**. This simply sends the source pattern a **:translate-wild-pathname-reversible** with the target pattern as target and the pathname being translated as the source pathname. For example:

contents of sys.translations *file:*

```
(fs:set-logical-pathname-host "SYS"
  :translations '(("SYS:DOC;**;*.*.*" "S:>Rel-6>doc>*.*.*")
                  ("SYS:**;*.*.*" "Q:>Rel-6>**>*.*.*")))
```

pathname to translate:

```
SYS:IO;PATHNM.LISP.23
```

translation pair found in phase 1:

```
("SYS:**;*.*.*" "ACME:>Rel-6>**>*.*.*")
```

result of translation:

```
ACME:>Rel-6>io>pathnm.lisp.23
```

In other words, the default is for the translation to occur according to the wildcard mapping given in the translations.

:new-pathname &key device directory name type version *Translation rule*

Similar to **:translate-wild**, but replaces the *directory*, *name*, *type*, or *version*. Any components not specified in the argument list will not be replaced, and will be derived via **:translate-wild-pathname-reversible** as for the **:translate-wild** translation rule type.

:vms-heuristicate &optional substitute *Translation rule*

Tries to make understandable VMS pathnames out of longer, hyphenated filenames. It produces usually understandable, hopefully unique, legal names and directories. In operation, it is similar to the **:translate-wild** type, but the components translated by wildcards are subjected to heuristics if needed to fit VMS's pathname syntax.

The *substitute* argument is used to perform character substitutions. For example, for VMS, it can be used to substitute "_" for "-".

```
("SYS:**;*.*.*" :vms-heuristicate ((#\- #\_)))
```

:vms-heuristicate-name &optional substitute *Translation rule*

Similar to **:vms-heuristicate**, but heuristicates only the name.

:vms-heuristicate-directory &optional substitute *Translation rule*

Similar to **:vms-heuristicate**, but heuristicates only the directory name.

:vms-new-pathname &key device directory name type version *Translation rule*

A cross between **:new-pathname** and **:vms-heuristicate**. Components not explicitly specified in the argument list are supplied by wildcard mapping plus heuristics as for **:vms-heuristicate**.

:vms-font &optional renamings *Translation Rule*

Parses the name component of the logical pathname as a font spec. For example, in **fix.roman.normal**, the *family* is **roman**, the *size* is **normal**, and the *style* is **fix**. (The style is optional). The font family is subjected to the VMS heuristics to fit in a smaller space (to allow room for the size and style). The result is concatenated with the size and style to construct a new name.

If the *renamings* argument is supplied, it is an alist of font names and replacement to be used instead of the one produced by the heuristics. This is useful in cases where the heuristic produces a confusing name, or where there would otherwise be name conflicts. For example, the following translation rule is used with the SYS: host for VMS hosts.

```
("SYS: FONTS; LGP-2; *.BFD.*" :vms-font
 (("DANG-MATH" "DANGM")
  ("GHELVETICA" "GHLVT")
  ("HELVETICA" "HELVT")
  ("TIMESROMAN" "TIMSR")
  ("XGP-VGV" "XGPVV")))
```

This translation rule serves to encode the relevant information that makes each font distinct.

In addition, **:vms-font** performs full VMS heuristics on the directory.

:vms-microcode *Translation rule*

Encodes the microcode names in such a way as to be sure to retain the information that distinguishes different microcodes.

The name component of the logical pathname is parsed into words. Each word is looked up in the alist **fs:*vms-microcode-translation-alist***. (The alist is shared with the equivalent translation for UNIX). If found, it is replaced with the replacement (a single character, except "MIC" maps to "") found in the second element of the alist bucket. This sequence of characters is then concatenated to produce the new filename.

The directory component is subject to full heuristication.

:tops20-heuristicate-directory &optional (levels **fs:*default-tops20-directory-levels****)* *Translation rule*

Compensates for the fact that TOPS20 directories are limited to a size of **fs:*tops-20-max-field-size***, including the "." characters as directory-level separators. Each level of directory is allocated a share of the available space, and is compressed to fit in that space as needed. In determining how much space to allocate to each level, the rule assumes that no more than *levels* directory levels will be needed. The default is **fs:*default-tops20-directory-levels***, or **fs:*default-tops20-directory-levels*** levels.

:unix-microcode *Translation rule*

Encodes the microcode names in such a way as to be sure to retain the information that distinguishes different microcodes.

The name component of the logical pathname is parsed into words. Each word is looked up in the alist **fs:*unix-microcode-translation-alist***. (The alist is shared with the equivalent translation for VMS). If found, it is replaced with the replacement (a single character, except "MIC" maps to "") found in the second element of the alist bucket. This sequence of characters is then concatenated to produce the new filename.

:unix-font &optional renamings *Translation rule*

Parses the name component of the logical pathname as a font spec. For example, in **fix.roman.normal**, the *family* is **roman**, the *size* is **normal**, and the *style* is **fix**. (The style is optional). The font family is subjected to the VMS heuristics to fit in a smaller space (to allow room for the size and style). The result is concatenated with the size and style to construct a new name.

If the *renamings* argument is supplied, it is an alist of font names and replacement to be used instead of the one produced by the heuristics. This is useful in cases where the heuristic produces a confusing name, or where there would otherwise be name conflicts. For example, the following translation rule is used with the SYS: host for UNIX hosts.

```
("SYS: FONTS; LGP-1; *.BFD.*" :unix-font
 (("DANG-MATH" "DANGMT")
  ("GHELVETICA" "GHELVT")
  ("HELVETICA" "HELVET")
  ("TIMESROMAN" "TIMESR")
  ("XGP-VGV" "XGPVGV")))
```

This translation rule serves to encode the relevant information that makes each font distinct.

:unix-type-and-version &optional renamings *Translation rule*

Used in situations where you need to retain both the type and version. This is usually needed where differing versions of the file need to coexist.

The name component is matched against the *renamings* alist. If it is found, the second element of the alist bucket is used instead. Then, if the last character of the name (or the replacement) is a digit, a "+" is added to the end. Then, the version number (or "" if **nil** or "*" if **:wild**) is added to the end. This is then used as the name component. The type is handled via the normal mechanisms.

The version is added to the name rather than the end of the type, so that the type field can be recognized by programs that look at the type (or canonical type).

:site-directory &key device directory name type version *Translation rule*

Substitutes the **:site-directory** attribute from the local site object for the host and directory. The arglist is like for **:new-pathname**. This is used to translate SYS:SITE;.

As a special feature, this rule can be overridden by an explicit entry for SYS: SITE; in the translations. This can be useful when debugging, to get a different site directory without modifying your site namespace object.

fs:patch-file system-name &optional file-type *Translation rule*

fs:patch-file rules, which will often be seen when doing a **fs:describe-logical-host**, are internal to the patch system. They provide for the translation of patch file logical names to physical files, in a system-dependent manner. These rules are added as a result of defining a system to be patchable.

fs:describe-logical-host *host* *Function*

Provides various information about the host, which can the a logical host or the name of a logical host. The information includes:

- Default physical host
- Translations
- Translation rules sorted by search order
- Translation rules sorted by group

This translation rule can be useful for determining what went wrong with a translation file.

fs:make-logical-pathname-host *name* &key *no-search-for-shadowed-physical* *Function*

Defines *name* (a string or symbol) to be the name of a logical pathname host. *Name* should not conflict with the name of any existing host, logical or physical. An **fs:make-logical-pathname-host** form often appears in the file `sys:site;`*system-name*`.system`.

fs:make-logical-pathname-host loads the file `sys:site;`*name*`.translations`. **load-patches** checks the translations file for each logical host that is defined in the current world; if any translations file has been changed it is reloaded (if and only if no specific systems are specified in its arguments).

The argument **:no-search-for-shadowed-physical** (default **nil**) means to look only in the existing pathname hosts for a host with the same name as the logical host. This saves time by not asking the namespace server whether the name of the newly defined logical host conflicts with the names of any physical hosts, but it prevents you from seeing the following warnings:

```
Warning: the host ~A must now be referred to as ~A: in pathnames,
                       since ~A is now a logical pathname host.
                       This affects ~[no~:;~:*~D~] extant pathnames.

Warning: the nickname ~A: for the physical host ~A
                        will now refer instead to the
                        logical pathname host ~A.
                        Use ~A: in pathnames.
```

Note: fs:add-logical-pathname-host is an obsolete name for this function.

More information is available about using the function **fs:make-logical-pathname-host** in system files. See the section "System Files".

fs:set-logical-pathname-host *logical-host* &key *:physical-host :translations :rules :site-rules (:no-translate* **t**) *:no-search-for-shadowed-physical* *Function*

Creates a logical host named "*logical-host*" if one does not already exist. This form appears in `sys:site;`*logical-host*`.translations` files. It establishes the translations of logical directories on *logical-host* to physical directories on one or more physical hosts. The machine specified by the *:physical-host* keyword serves as the default physical host.

The *:translations* keyword specifies the list of translations from logical to physical directories.

- For more information about translations lists: See the section "Translations Files".

- For the format of the lists and the translation rules: See the section "Pathname Translation".

- For a discussion of the *:rules* and *:site-rules* keywords: See the section "Defining a Translation Rule".

If *no-translate* is **nil**, the translation of every interned logical pathname is checked. Properties are copied from the old physical pathname to the the new one, and logical pathnames that now have no corresponding physical pathnames are uninterned.

If *no-translate* is not **nil** or not supplied, this mapping is suppressed, and some physical pathnames might not get the properties of the logical pathname. This is not normally of any consequence, so *no-translate* defaults to **t**.

The argument *no-search-for-shadowed-physical* (default **nil**) means to look only in the existing pathname hosts for a host with the same name as the logical host. This saves time by not asking the namespace server whether the name of the newly defined logical host conflicts with the names of any physical hosts, but it prevents you from seeing the following warnings:

```
Warning: the host ~A must now be referred to as ~A: in pathnames,
                       since ~A is now a logical pathname host.
                       This affects ~[no~:;~:*~D~] extant pathnames.

Warning: the nickname ~A: for the physical host ~A
                        will now refer instead to the
                        logical pathname host ~A.
                        Use ~A: in pathnames.
```

For more information about `sys.translations` files, see the section "Pathname Translation". Also see the section "Translations Files".

(flavor:method :translated-pathname fs:logical-pathname) *Method*

Converts a logical pathname to a physical pathname. It returns the translated pathname of this instance: a pathname whose **:host** component is the physical host that corresponds to this instance's logical host. See the section "Syntax for Logical Pathnames".

If this message is sent to a physical pathname, it simply returns itself.

(flavor:method :back-translated-pathname fs:logical-pathname) *pathname* *Method*

Converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*. See the section "Syntax for Logical Pathnames".

This message might be used in connection with truenames. Given a stream that was obtained by opening a logical pathname,

```
(send stream :pathname)
```

returns the logical pathname that was opened.

```
(send stream :truename)
```

returns the true name of the file that is open, which of course is a pathname on the physical host. To get this in the form of a logical pathname, you would do the following:

```
(send (send stream :pathname)
      :back-translated-pathname
      (send stream :truename))
```

If this message is sent to a physical pathname, it simply returns its argument. Thus the above example works no matter what kind of pathname was opened to create the stream. However, it is important to note two situations in which back translation can fail to do what you expect:

Links	If opening the file involved following a link, the truename will no longer match, and back translation might not be able to convert it to a physical pathname at all.
File-system restrictions	If the translation involved compressing or modifying a name to adapt to a file-system's rules, the physical pathname might be translated to a logical pathname different from the one originally used.

Back translation is useful only in cases where the logical pathname is wanted for informational, not operational, purposes. For example, if you remember a back translation to reopen the file, you might end up with physical instead of logical pathnames in your program. Physical pathnames are not transportable between sites.

One way to avoid this problem is to avoid back translation. Often, all that is needed is the version number, in which case the following code will serve:

```
(send (send stream :pathname)
      :new-default-pathname
      :version (send (send stream :truename) :version))
```

Note that **:new-default-pathname** is used rather than **:new-pathname**. This is necessary because the logical host and the physical host are of different types. When copying components between host types, you need to allow for certain substitutions. In this case, if the physical host is a UNIX system, the version will be **:unspecific**, and **:new-default-pathname** will convert this to the nearest equivalent for logical pathnames: **:newest**.

Values of Pathname Components

The set of permissible values for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually treated as things chosen by the user that need to be preserved and passed around.

Most programs do not use or specify the components of a pathname explicitly, or only in a very limited way. In this way, they can remain operating-system-independent, while letting the pathname system take care of most issues of compatibility. In general, you should avoid where possible using specific values of pathname components in your programs. The descriptions here are illustrative but not complete, and programs should be written to expect component values other than those given here.

It is important to remember that not all pathname flavors accept all the values indicated here. For example, UNIX pathnames accept a type or version of **:unspecific**; few other pathnames do. Some systems do not allow certain characters or limit certain fields to a certain length.

It is generally *not* possible to simply copy components from one flavor of pathname to another. It is often necessary to perform substitutions in order to produce a legal pathname. The **:new-default-pathname** message can be used instead of **:new-pathname** to get this substitution where necessary. The **:new-default-pathname** message attempts to substitute something as close as possible in meaning to the original component; however, the substitution can be arbitrary if necessary. For this reason, it is better to avoid copying components between pathnames of differing flavor, where possible.

The type is always a string (unless it is one of the special symbols **nil**, **:unspecific**, or **:wild**). Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are "lisp", compiled Lisp programs (on, for example, a LMFS host) are "bin", text files are "text", and so on. The set of characters allowed in the type, and the number of characters, are system-dependent. In order to process file types in a system-independent way, the *canonical type* mechanism has been devised. A canonical type is a system-independent keyword symbol representing the conceptual type of a file. For instance, a Lisp source file on VMS has a file type of "LSP", and one on UNIX has a file type of "l". When we ask pathnames of either of these natures for their canonical type, we receive the keyword symbol **:lisp**. See the section "Canonical Types in Pathnames".

The version is either a number (specifically, a positive fixnum), or one of the symbols **nil**, **:unspecific**, **:wild**, **:newest**, or **:oldest**. **nil**, **:unspecific**, and **:wild** have been explained above. **:newest** refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. **:oldest** refers to the smallest existing version number.

The host component of a pathname is always a host object. See the section "Attributes for Objects of Class "Host"".

The device component of a pathname can be one of the symbols **nil** or **:unspecific**, or a string designating some device, for those file systems that support such a notion.

The file name can be **nil** or a string, or **:wild**.

The directory component is highly system-specific. While it can be **nil** for any type of host, values designating actual directories, or partially wild specifications for directories, are more complicated. On nonhierarchical file systems, the directory component is usually a string such as "LMDOC", designating the name of the directory.

On hierarchical file systems, the directory component, when not **nil**, is a list of *directory level components*. For example:

LMFS pathname	*Directory component*
`>sys>io>qfile.lisp.2357`	`("sys" "io")`

"sys" and **"io"** are the directory level components. Since the "root directory" of hierarchical file systems has no directory level components, it would be represented as **nil**, but this is impermissible, since **nil** already means that the directory component has not been specified. Thus, **:root** is used as the directory component in that case.

Relative pathnames on hierarchical file systems are represented by directory components having the level component **:relative**, followed by a number of occurrences of the symbol **:up** equal to the number of "upward relativization symbols", followed by the remaining directory level components. For example:

LMFS relative pathname	*Directory component*
`<<x>y>z.lisp`	`(:relative :up :up "x" "y")`

Directory components of pathnames for hierarchical file systems, on some systems, can also have the symbol **:wild** or a partially wild string (such as "foo*") as directory level components, to do level-by-level matching of level components. Also, on some systems, the level component **:wild-inferiors** (which is printed as "**" on LMFS and logical pathnames, and "..." on VMS, currently the only ones supporting it) to designate "any number, including zero of directory levels" to a directory list operation.

Note that some systems (currently VMS) do not allow using zero directory levels to denote their root directory. In this case, **:wild-inferiors** cannot stand alone, but must follow some other directory spec. For example: "[FOO...]" or "[*...]".

Directory Pathnames and Directory Pathnames as Files

In almost all systems having hierarchical directories, and certainly all the ones supported by the Symbolics computer as file server systems, the directories are implemented internally as special files, known about by the operating system. The data in these files is not accessible to the user except through the defined operating system interfaces for dealing with directories.

Typically, listings of the contents of directories on hierarchical directory systems display names of both files and directories contained in the listed directory (as well as of links, on systems that support links).

Directories on hierarchical directory systems and files thus have some things have in common. Both appear in directories. Also, directories can usually be renamed (as can files) or deleted, when the appropriate restrictions of the operating system are met, such as being empty. You can ask about the properties of a directory, or change some of them, with **fs:file-properties** and **fs:change-file-properties**, respectively, just as you do with a file.

Using LMFS as an example, consider the directory named "bar", which is contained in the directory named "foo", which itself is contained in the ROOT. A file in this directory named "tables.lisp.6" would have the following pathname:

```
>foo>bar>tables.lisp.6
```

The directory in which it is contained, bar, has the following pathname:

```
>foo>bar.directory.1
```

The file type of a directory, on LMFS, is "directory", and the version number of all directories is 1. The file types of directories, and their versions, if appropriate, vary among operating systems. If you wanted to rename, delete, or deal with the properties of the directory bar, you would have to present the above filename for this directory. A pathname of this type, which names a directory, as though it were a file, is called a *directory pathname as file.*

Directory pathnames as files are appropriate only to systems with hierarchical directories. On other systems, you cannot address directories directly.

The most common use of directories, however, is to reference files in them. The following pathname mentions the directory "bar" in this way:

```
>foo>bar>tables.lisp.6
```

This filename, when parsed into a pathname for the appropriate LMFS host, has a name component of "tables", a type component of "lisp", a version component of 6, and a directory component (in fact **("foo" "bar")**) that designates the directory bar, inferior of foo, inferior of the ROOT. Such a pathname, which designates a given directory via its directory component, is called a *pathname as directory* for that directory. Of course, since the file name, type, and version are irrelevant to the specification of the directory, it is only one of many possible "pathnames as directory" for the directory bar.

The concept of pathname as directory is more general than the concept of directory pathname as file, since directories on nonhierarchical systems be described by their pathnames as directories as well. For instance, the following TENEX pathname, which describes a file in the "LMDOC" directory, is a pathname as directory for the LMDOC directory:

```
<LMDOC>CHFILE.TEXT;7
```

Note, also, that any pathname whose directory component is not **nil** is a pathname as directory for *some* directory.

Therefore, the Symbolics Common Lisp primitives and operations that deal with directories explicitly (for example, **fs:expunge-directory** and **fs:all-directories**) expect pathnames of directories to be represented in the "pathname as directory" form. It is the canonical, system-independent way to represent pathnames of directories in the Symbolics system.

The following two messages convert between directory pathnames as files and pathnames as directories:

(flavor:method :directory-pathname-as-file pathname) *Method*

Every pathname whose directory component is not **nil** is a pathname as directory for *some* directory. This method returns the directory pathname as file for that directory.

```
(setq p (fs:parse-pathname "Quabbin:>sys>lmfs>fsstr.lisp.243"))
#P"Q:>sys>lmfs>fsstr.lisp.243"

(send p :directory-pathname-as-file)
#P"Q:>sys>lmfs.directory.1"
T
```

See the method **(flavor:method :pathname-as-directory pathname)**.

(flavor:method :pathname-as-directory pathname) *Method*

This method is intended to be sent to a pathname that is the valid directory pathname as file for some directory. It produces one of many possible pathnames as directory for that directory, namely, the one whose name, type, and version are all **nil**.

```
(setq p1 (fs:parse-pathname "Quabbin:>sys>io.directory.1"))
#P"Q:>sys>io.directory.1"
(setq p2 (send p1 :pathname-as-directory))
#P"Q:>sys>io>"
(send p2 :directory-pathname-as-file)
#P"Q:>sys>io.directory.1"
```

See the method **(flavor:method :directory-pathname-as-file pathname)**.

If you are used to other systems' file-naming conventions, you may be confused by pathnames that have real directory components, but no name, type, or version. When typed in or printed, they look like the following:

```
>jones>book>examples>
```

Users who are familiar with Multics or UNIX immediately see such pathnames as invalid, even though they are often used on the Symbolics computer to access Multics and UNIX. When parsed for LMFS or Multics, the above filename string produces a pathname whose directory component designates the directory "examples", which is contained in "book", which itself is contained in "jones", an inferior of the ROOT. The name, type, and version components of this pathname are **nil**. This pathname is equivalent to the following:

```
>jones>book>examples>↔.↔.↔
```

Either of these is a canonical pathname as directory for the directory "examples". Typing such pathnames as input is exceedingly common, since the merging process, given such a pathname as its unmerged input, replaces the directory component of the default with a directory component specifying the directory named by the "pathname as directory". See the section "Pathname Defaults and Merging". For example:

```
Default:        Q:>abel>baker>cakes.list
User Typein:    >Romanoli>weddings>
Merged output:  >Romanoli>weddings>cakes.list
```

Compare this with the following:

```
Default:        Q:>abel>baker>cakes.list
User Typein:    >Romanoli>weddings
Merged output:  >Romanoli>weddings.list

Default:        Q:>abel>baker>cakes.list
User Typein:    >Romanoli>weddings>↔.↔.73
Merged output:  >Romanoli>weddings>cakes.list.73
```

All the Symbolics hierarchical directory parsers recognize a trailing directory delimiter as an instruction to construct a pathname with **nil** name, type, and version, for the directory designated—a "pathname as directory". (The version component, however, remains **:unspecific** for systems not supporting file versions.) This is true even of the parsers for UNIX and Multics, on which systems such syntax is never seen.

This mode of directory naming is usually familiar to users of nonhierarchical systems. The following TENEX pathname results, when parsed, in a pathname as directory for the LMDOC directory (on the appropriate TENEX host), with name, type, and version of **nil**, that can be used in merging operations in a way similar to that shown in the above LMFS example.

```
<LMDOC>
```

As a side-effect of these conventions, the following kinds of pathnames occasionally occur on LMFS or Multics:

```
<lmdoc>
```

As explained above, this is a valid way of entering the following relative pathname:

```
<lmdoc>↔.↔.↔
```

Case in Pathnames

The pathname system handles alphabetic case in pathnames and transferring of pathname components between hosts with different preferred alphabetic cases.

The components of a pathname (directory, name, type, and so on) have two possible representations for case, *raw* (also called *native*) and *interchange*. The raw case representation keeps the case in whatever form is normal for that system (for example, lowercase for UNIX, uppercase for TOPS-20). Interchange representation is a format for manipulating pathname components in a host-independent manner. All pathname defaulting and cross-host translation functions use the interchange form of pathname messages.

All the standard messages to pathnames (for example, **:directory**, **:name**) return pathname components in interchange case rather than raw case.

The components are stored internally in raw case, that is, the actual alphabetic case in which the names of the files are stored, or to be stored, in the host's file system. It is possible to access the raw case representation via the set of messages **:raw-directory**, **:raw-name**, and so forth. However, programs seeking to be system-independent should not use these messages, but the standard ones, **:directory**, **:name**, and so forth. Doing so ensures that pathname components transferred between system types stay in the preferred case for each of the systems concerned.

The raw forms of the messages are provided for writing host-specific code or for manipulating several pathname objects known to be on the same host.

Interchange case form	*Raw case form*
:device	**:raw-device**
:directory	**:raw-directory**
:name	**:raw-name**
:type	**:raw-type**

The interchange form of the message specifies the following effect:

Case of component	*Translated case returned*
System default	Uppercase
Mixed case	Mixed case
Opposite to default	Lowercase

Uppercase was chosen as the interchange case because strings like "LISP", representing pathname components, appear in many programs. Either choice (upper or lower) would have been natural for some hosts and not for others.

This facility provides more features for dealing with pathname components independent of the case-sensitivity of file names of different hosts. The following table shows some examples for different host types.

Host	*Message*	*Applied to raw form*	*Returns interchange form*
UNIX	**:name**	"foo-bar.baz"	"FOO-BAR"
	:name	"FOO-BAR.BAZ"	"foo-bar"
	:name	"Foo-Bar.Baz"	"Foo-Bar"
Lisp Machine	**:name**	"foo-bar.baz"	"FOO-BAR"
File System	**:name**	"FOO-BAR.BAZ"	"FOO-BAR"
	:name	"Foo-Bar.Baz"	"FOO-BAR"
TOPS-20	**:name**	"FOO-BAR.BAZ"	"FOO-BAR"
	:name	"foo-bar.baz"	"foo-bar"
	:name	"Foo-Bar.Baz"	"Foo-Bar"
VMS4	**:name**	"FOO_BAR.BAZ"	"FOO-BAR"

Note that VMS has only one example; that is because VMS supports uppercase only. VMS uses the underscore character "_" where other operating systems use the hyphen "-".

Note that the Lisp Machine File System (LMFS) appears not to follow the interchange case rules. This is because, for LMFS, case is usually maintained but is not significant ("foo", "Foo", and "FOO" are all the same). Thus any mixture of cases in a file name satisfies the "system default" condition and returns all uppercase for the interchange form.

Functions that manipulate pathnames, such as **fs:make-pathname**, **fs:merge-pathnames**, and **fs:merge-pathname-and-set-defaults**, manipulate components in interchange case.

Pathname-constructing functions such as **fs:make-pathname** and pathname messages such as **:new-pathname** and **:new-default-pathname** accept both **:directory** and **:raw-directory**, to allow specification of components in either interchange case or raw case.

Pathname Defaults and Merging

It is unreasonable to require the user to type a complete pathname, containing all components. Instead the program is expected to supply a *default pathname*, from which values of components not specified by the user can be taken.

Every program that prompts the user for a pathname should maintain some default pathname, display it to the user when prompting for a pathname, and merge the parsed input from the user with that default. The function **prompt-and-read** provides easy ways to do all of these things. See the function **prompt-and-read**. No program should use any pathname obtained from user input without merging it against *some* default. Since it is impossible for a user to type a pathname correctly without knowing against which default it will be merged, the default must be displayed to the user.

A *default default* is available for programs that have no better idea of a default pathname, and a function (**fs:default-pathname**) for customizing default pathnames.

Typically, a program might take the default default, customize it, perhaps by supplying a specific file type (usually via the canonical type mechanism), prompt the user for the name of a file, displaying that default, and merge the user's parsed input against that default.

A more complex program, one that requires an input file and an output file, might proceed as follows: It obtains the pathname of its input file as above, and prepares a default pathname for its output file by customizing the input file pathname, usually by supplying a new type, and presents and uses that as a default for the prompt for the output file pathname.

The *merging* operation is performed by the function **fs:merge-pathnames**. It takes as input an unmerged pathname and a default pathname and returns a *merged pathname*, which has no missing components. Basically, the missing components in the unmerged pathname are filled in from the default pathname. The merging operation also takes a *default version* argument, which specifies the version number of the output pathname, *if there is no version mentioned in the unmerged pathname*.

That is, the version number is almost never defaulted from the default pathname. If the default version argument is not supplied, it is assumed to be **:newest**. The version number of the default is used as a default version in the following cases:

- Neither name, type, nor version is specified by the unmerged pathname.
- The unmerged pathname does not have a version, and the value of the default version argument is **:default**.

The full details of the merging rules are as follows.

1. If the unmerged pathname does not supply a device, the device is the default file device for that host.
2. If the unmerged pathname does not specify a host, device, directory, name, or type, that component comes from the defaults.
3. If the unmerged pathname supplies a version, it is used.
4. If it does not supply a version, the default version as explained above is used.

Thus, if the user supplies just a name, the host, device, directory and type will come from the default, with the default version argument (or **:newest** if there was none). If the user supplies nothing, or just a directory, the name, type, and version comes over from the default together. If the host's file name syntax provides a way to input a type or version without a name, the user can let the name default but supply a different type or version than the ones in the default.

The system also defines an object called a *defaults alist*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults alist is basically an object containing a replaceable pathname. **fs:merge-pathnames** accepts a defaults alist as its default pathname argument as well as a pathname. **fs:merge-pathnames-and-set-defaults** is like **fs:merge-pathnames** but *requires* a defaults alist as its default pathname argument. When it has completed its merge, it stores the result back into the defaults alist before returning it. See the function **fs:merge-pathnames-and-set-defaults**. It is important that you do not attempt to construct a defaults alist, but instead use the primitives provided. See the function **fs:make-pathname-defaults**. See the function **fs:copy-pathname-defaults**.

See the function **fs:set-default-pathname**.

The following special variables are parts of the pathname interface that are relevant to defaults.

fs:*default-pathname-defaults* *Variable*

The default defaults alist; if the pathname primitives that need a set of defaults are not given one, they use this one. Most programs, however, should have their own defaults rather than using these.

fs:load-pathname-defaults *Variable*

In your new programs, we recommend that you use the variable ***load-pathname-defaults*** which is the Symbolics Common Lisp equivalent of **fs:load-pathname-defaults**.

The defaults alist for the **zl:load** and **compiler:compile-file** functions. Other functions can share these defaults.

Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the file attribute list line of a source file in the family. See the section "File Attribute Lists". All types of files with that name, in that directory, belong together. They are different members of the same family; for example, they might be source code, compiled code, and documentation for a program. All versions of files with that name, in that directory, belong together.

The generic pathname of pathname *p* has the same host, device, directory, and name as *p* does. However, it has a version of **nil**. Furthermore, if the canonical type of *p* is one of the elements of **fs:*known-types***, then it has a type of **nil**; otherwise it has the same type as *p*. The reason that the type of the generic pathname works this way is that in some file systems, such as that of ITS, the type component can actually be part of the file name; ITS files named "DIRECT IONS" and "DIRECT ORY" do not belong together.

The **:generic-pathname** message to a pathname returns its corresponding generic pathname. See the method **(flavor:method :generic-pathname pathname)**.

fs:*known-types* *Variable*

A list of the canonical file types that are "not important"; constructing a generic pathname will strip off the file type if it is in this list. File types not in this list are really part of the name in some sense. The following is the initial list:

```
(:LISP :QBIN :BIN :IBIN NIL :UNSPECIFIC)
```

Some users might need to add to this list. See the section "Canonical Types in Pathnames".

Relative Pathnames

Many operating systems support a notion called *relative pathnames* in order to simplify the typing of filenames by their users. Typically, a user on a system such as Multics or UNIX tells the system what directory on the system is his or her *working directory*. These operating systems assume the working directory as the default directory for filenames whose directory is not specified. For example, when the user types a filename, perhaps as an argument to a command (such as "print foo") the system assumes that the name foo refers to a file named foo in the working directory, as long as the user did not specify another directory (for instance, by saying "print >sources>c>foo").

On hierarchical systems, such as UNIX and Multics, the working directory can often be several levels deep, and have a full name that is therefore cumbersome to type. The concept of working directory is all the more powerful in these cases. Since the hierarchical organization of directories exists to facilitate relating files by placing them in directories in common subtrees, it is common for users working on such systems to want to reference files in "siblings" of their working directory, or "uncles", or even "inferiors" or "inferiors of inferiors", that is, directories near in the directory hierarchy to their working directory.

In order to facilitate the referencing of files in directories "near" the working directory, without having to type full pathnames of directories, these systems support *relative pathnames*, which are interpreted relative to the working directory. Relative pathnames are always syntactically distinguishable from other pathnames. For instance, on Multics, if the working directory is >udd>Proj>Username, the pathname

```
<Othername>stuff>x.pl1
```

refers to the file

```
>udd>Proj>Othername>stuff>x.pl1
```

Although it supports relative pathnames, the Lisp Machine File System does not support a concept of working directory. One rationale for this is the fact that the user might be communicating with many systems at once, and might have several working directories to remember. The merging and defaulting system takes the place of the working directory concept. See the section "Pathname Defaults and Merging". The default pathname, which is displayed when a user is asked to enter a pathname, determines the default directory for a pathname having no directory explicitly specified. What is more, it specifies the default values of other components as well.

Systems supporting relative pathnames usually have some special syntax to indicate a pathname that is relative to a superior of the working directory, and another to indicate pathnames relative to superiors of the working directory. We call these "upward relativization" and "downward relativization". In this context, a pathname with an explicit directory specified is called an *absolute pathname*, and one without an explicit directory, a relative pathname. However, since specification of no directory at all is a very common case handled by systems that do not otherwise support relative directories, namely, by simply defaulting an entire directory component, this is not considered a relative pathname by the Symbolics system.

The Symbolics system supports relative directories for those hierarchical systems that support it themselves. As might be expected, the "resolution" of relative pathnames entered by the user is performed relative to the default pathname, as opposed to any working directory. Resolution of relative pathnames is performed by **fs:merge-pathnames** as part of its normal operation.

The following examples, using LMFS pathnames, show some examples of relative pathnames and their resolution via merging:

```
Default:     >sys>lmfs>new>xst.lisp
Unmerged:    test>xst.lisp
Merged:      >sys>lmfs>new>test>xst.lisp
Default:     >sys>lmfs>new>xst.lisp
Unmerged:    <test>thing.lisp                ;upward relativization
Merged:      >sys>lmfs>test>thing.lisp
Default:     >sys>lmfs>new>xst.lisp
Unmerged:    <<test>                        ;upward relativization
Merged:      >sys>test>xst.lisp
Default:     >sys>lmfs>new>xst.lisp
Unmerged:    test>best>                     ;downward relativization
Merged:      >sys>lmfs>new>test>best>xst.lisp
Default:     >sys>lmfs>new>xst.lisp
Unmerged:    <xst.lisp
Merged:      >sys>lmfs>xst.lisp
Default:     >sys>lmfs>new>xst.lisp
Unmerged:    <<abel>baker>foo.lisp
Merged:      >sys>abel>baker>foo.lisp
```

Canonical Types in Pathnames

A *canonical type* for a pathname is a symbol that indicates the nature of a file's contents. To compare the types of two files, particularly when they could be on different kinds of hosts, you compare their canonical types. (**fs:*default-canonical-types*** and **fs:*canonical-types-alist*** show the canonical types and the default surface types for various hosts.)

Some terminology:

canonical type	A host-independent name for a certain type of file, for example, Lisp compiled code files or LGP font files. A canonical type is a keyword symbol.
file specification	What you type when you are prompted to supply a string for the system to build a pathname object.
surface type	The appearance of the type component in a file specification. This is a string in native case.
default surface type	Each canonical type has as part of its definition a representation for the type when it has to be used in a string. Default surface type is the string (in interchange case) that would be used in a string being generated by the system and shown to the user. See the special form **fs:define-canonical-type**.
preferred surface type	Some canonical types have several different possible surface

representations. The definition for the type designates one of these as the preferred surface type. It is a string in interchange case. ("Default surface type" implies "preferred surface type" when one has been defined.)

Each canonical type has a default surface representation, which can be different from the surface file type actually appearing in a file specification. **:lisp** is a canonical type for files containing Lisp source code. For example, on UNIX, the default surface representation of the type for **:lisp** files is "L". (Remember, the default surface representation is kept in interchange case.) The surface type in a file specification containing lisp code is different on different systems, "LISP" for Lisp Machine file system, "l" for UNIX. You can find out from a pathname object both the canonical type for the pathname and the surface form of the type for the pathname by using the **:canonical-type** message. See the method **(flavor:method :canonical-type pathname)**.

The following tables illustrate the terminology.

UNIX

Surface type	"l"	"lisp"	"foo"
Raw type	"l"	"lisp"	"foo"
Type	"L"	"LISP"	"FOO"
Canonical type	**:lisp**	**:lisp**	"FOO"
Original type	**nil**	"LISP"	"FOO"

Genera

Surface type	"l"	"lisp"	"foo"
Raw type	"l"	"lisp"	"foo"
Type	"L"	"LISP"	"FOO"
Canonical type	"L"	**:lisp**	"FOO"
Original type	"L"	**nil**	"FOO"

To translate the type field of a pathname from one host to another, determine the canonical type, using the surface type on the original host. Then find a surface type on the new host for that canonical type.

Copying operations can preserve the surface type of the file through translations and defaulting rather than by converting it to the surface form for the canonical type. For example:

```
(multiple-value-bind (ctype otype)
    (send p ':canonical-type)
  (send p ':new-pathname
          ':canonical-type ctype
          ':original-type otype
          ':name "temp-p"))
```

Correspondence of Canonical Types and Editor Modes

fs:*file-type-mode-alist* is an alist that associates canonical types (in the car) with editor major modes (in the cdr).

```
((:LISP . :LISP) (:SYSTEM . :LISP) (:TEXT . :TEXT) ...)
```

Wildcard Pathname Mapping

In the Symbolics system, as in some other systems, wildcard pathnames are used not only to specify groups of files, but to specify mappings between pairs of pathnames, for operations such as renaming and copying files.

For example, you might ask to copy *foo*.lisp to *bar*.lisp. All the files to be copied match the wildcard name *foo*.lisp. *bar*.lisp is a specification of how to construct the pathname of the new file. The two wildcard pathnames, as in the above example, are called the *source pattern* and *target pattern*. The original name of any file to be copied is called the *starting instance*. Here is an example:

Source pattern:	`f:>fie>*old*.lisp`
Target pattern:	`vx:/usr2/fum/*older*.l`
Starting instance:	`f:>fie>--oldfoo.lisp`
Target instance:	`vx:/usr2/fum/--olderfoo.l`

A more abstract description of this terminology:

Source pattern	A pathname containing wild components.
Target pattern	A pathname containing wild components.
Source instance	A pathname that matches the source pattern.
Target instance	A pathname specified by applying the common sequences between the source and target patterns to the source instance.

Two Zmacs commands accept pairs of wildcard file specifications:

Copy File (m-X)
Rename File (m-X)

The components of the target instance are determined component-by-component for all components except the host. (The host component is always determined literally from the source and target patterns; it cannot be wild.) The mapping of pathnames is done in the native case of the target host. The source pattern and source instance are coerced to the target host via the **:new-default-pathname** message before the mapping takes place. See the method **(flavor:method :new-default-pathname pathname)**.

When the type of the target pattern is **:wild**, it uses the canonical type for the target, regardless of the surface form for the type in the source pattern and instance. One implication is that the resulting translation is not reversible. See the section "Reversible Wildcard Pathname Translation".

Note: In LMFS, * as the directory portion of a file specification specifies a relative pathname. You must use >** to indicate a wild directory component that matches any directory at all. See the section "LMFS Pathnames".

Here are the rules used in constructing a target instance, given the source and target patterns and a particular source instance. This set of rules is applied separately to each component in the pathname. In the mapping rules, a * character as the only contents of a component of a file spec is considered to be the same as the keyword symbol **:wild**. The rule uses the patterns from the example above.

1. If the target pattern does not contain *, copy the target pattern component literally to the target instance.

2. If the target pattern is **:wild**, copy the source component to the target literally with no further analysis. The type component is handled somewhat differently — when source and target hosts are of different system types, it uses the canonical-type mechanism to translate the type. This does not apply when the target pattern is **:wild-inferiors**, in directory specifications.

3. Find the positions of all * characters in the source and target patterns. Take the characters intervening between * characters as a literal value. Literal values for the name component:
   Source: `old`
   Target: `older`

4. Find each literal value from the source pattern in the source instance. Take the characters intervening between literal values as a matching value for the * from the source pattern. The matching value could be any number of characters, including zero. Matching values for the name component:
   `--` and `foo`

5. Create the component by assembling the literal and matching values in left to right order, substituting the matching values where * appears in the target pattern. For the name component:
   `--olderfoo`

   When not enough matching values are available (due to too few * in the source pattern) use the null string as the matching value. When the source pattern has too many *, ignore the first extra * and everything following it.

Some examples:

Source pattern	*Source instance*	*Target pattern*	*Target instance*
`*report`	`6802-report`	`*summary`	`6802-summary`
`lmfs-*`	`lmfs-errors`	`*`	`lmfs-errors`
`l*`	`l`	`l*`	`l`
`l*`	`lisp`	`l*`	`lisp`
`OLD-DIR`	`OLD-DIR`	`NEW-PLACE`	`NEW-PLACE`
`*`	`doc`	`*-extract`	`doc-extract`
`doc`	`doc`	`doc-extract`	`doc-extract`

Wildcard Directory Mapping

The rules for mapping directory components between two wildcard pathnames and a starting instance are parallel to the rules for single names. Directory-level components play roughly the roles of characters in the name-translating algorithm. See the section "Wildcard Pathname Mapping".

Consider a directory component as a sequence of directory level components. The levels are separated by level delimiters (> in LMFS). Example: In the pathname >foo>bar>*>mumble*>x>**>y>a.b.3, the directory-level components are foo, bar, *, mumble*, x, **, and y. The source and target patterns, as well as the starting instance, are considered as sequences of directory-level components, and are matched and translated level by level.

For this purpose, each directory-level component can be classified as one of three types:

Type	*Directory representation*
constant	String containing no *'s
wild-inferiors	** in LMFS, ... in VMS
must-match	* or string containing at least one * (but not the string representing wild-inferiors)

The matching and mapping of constant and wild-inferiors levels proceeds in a manner identical to the matching and mapping of constant substrings and *s for single names. See the section "Wildcard Pathname Mapping". Constant directory level components act as constant substrings in that algorithm, and wild-inferiors levels as *s. That is, wild-inferiors level components match and, on the target side, carry, zero to any number of constant directory-level components.

Examples:

Source pattern:	`>sys>**>*.*.newest`
Target pattern:	`>old-systems>release-5>**>*.*.*`
Starting instance:	`>sys>lmfs>patch>lmfs-33.patch-dir.66`
Target instance:	`>old-systems>release-5>lmfs>patch>lmfs-33.patch-dir.66`

Source pattern:	`>a>b>c>**>d>e>**>x.y.*`
Target pattern:	`>t>u>**>m>**>w>*.*.*`
Starting instance:	`>a>b>c>p>q>d>e>f>g>x.y.1`
Target instance:	`>t>u>p>q>m>f>g>w>x.y.1`

Must-match components are matched with exactly one directory-level component, which must be present. They are mapped according to the string-mapping rules in the name-translating algorithm. See the section "Wildcard Pathname Mapping".

Example:

Source pattern:	`>a>b>c>foo*>d>*>*.*.*`
Target pattern:	`>x>*bar>y>*man>*.*.*`
Starting instance:	`>a>b>c>foolish>d>yow>a.lisp.1`
Target instance:	`>x>lishbar>y>yowman>a.lisp.1`

You can intersperse constants, must-matches, and wild-inferiors directory-level components, as long as the sequence of wildcard types is the same in both patterns.

Example:

Source pattern:	`>a>*>c>**>*.lisp.*`
Target pattern:	`>bsg>sub>new-*>q>**>*.*.*`
Starting instance:	`>a>bb>c>d>e>p1.lisp.6`
Target instance:	`>bsg>sub>new-bb>q>d>e>p1.lisp.6`

Pathname Functions

The following functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

pathname *x* &optional *(defaults* ***default-pathname-defaults****)* *Function*

Converts *x* into a pathname. The argument can be a pathname, a string, or a stream. **pathname** always returns a pathname.

Compatibility Note: The optional argument *defaults* is an extension to Symbolics Common Lisp, which might not work in other implementations of Common Lisp, and is not supported in CLOE. See the function **fs:parse-pathname**.

parse-namestring *thing* &optional *host* (*defaults* ***default-pathname-defaults***) &key (*start* **0**) *end junk-allowed* *Function*

Turns *thing* into a pathname. The *thing* is usually a string (that is, a namestring), but it can be a symbol (in which case the print name is used) or a pathname or stream (in which case no parsing is needed, but an error check can be made for matching hosts).

This function does *not*, in general, do any defaulting of pathname components, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are present because in some cases a namestring can be parsed only with reference to a particular file name syntax of several available ones. If *host* is non-**nil**, it must be a host name that could appear in the host component of a pathname. If *host* is **nil**, then the host name is extracted from the default pathname in *defaults* and used to determine the syntax convention.

For a string or symbol argument, **parse-namestring** parses a file name within it in the range delimited by the **:start** and **:end** arguments. **:start** is an integer index into *thing* and defaults to the beginning of the string. **:end** is also an integer index, and defaults to the end of the string.

If **:junk-allowed** is non-**nil**, the first value returned is the pathname parsed, or **nil** if no syntactically correct pathname was seen. If **:junk-allowed** is **nil** (the default), the entire substring is scanned, and the returned value is the pathname parsed.

An error is signalled if the substring does not consist entirely of the representation of a pathname, possibly surrounded on either side by whitespace characters.

In either case, the second value returned is the index into the string of the delimiter that terminated the parse, or the index beyond the substring if the parse terminated at the end of the substring (as is always the case if **:junk-allowed** is **nil**).

If *thing* is not a string or a symbol, the value of the **:start** keyword is always returned as the second value.

Parsing an empty string alway succeeds, producing a pathname with all components (except the host) equal to **nil**.

Note that if *host* is specified and non-**nil**, and *thing* contains a manifest host name, an error is signalled if the hosts do not match.

The *host* and *default* arguments are not used in the CLOE implementation except with respect to logical pathnames.

```
(parse-namestring "/tmp/ASD567") => #P"/tmp/ASD567"
```

fs:parse-pathname *thing* &optional *with-respect-to (defaults* **fs:*default-pathname-defaults***) *Function*

Turns *thing*, which can be a pathname, a string, or a Maclisp-style name list, into a pathname. Most functions that take a pathname argument call **fs:parse-pathname** on it so that they accept anything that can be turned into a pathname. Some, however, do it indirectly, by calling **fs:merge-pathnames**.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *with-respect-to* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host.

If *with-respect-to* is supplied, it should be a host or a string to be parsed as the name of a host. If *thing* is a string, it is then parsed as a true string for that host; host names specified as part of *thing* are not removed. Thus, when *with-respect-to* is not **nil**, *thing* should not contain a host name.

If *with-respect-to* is not supplied or is **nil**, any host name inside *thing* is parsed and used as the host. If *with-respect-to* is **nil** and no host is specified as part of *thing*, the host is taken from *defaults*.

Examples, using a LMFS host named Q:

```
(fs:parse-pathname "a:>b.c" "q") => #P"Q:a:>b.c"  ;(wrong)
(fs:parse-pathname "q:>b.c" "q") => #P"Q:q:>b.c"  ;(wrong)
(fs:parse-pathname "q:>b.c")     => #P"Q:>b.c"
(fs:parse-pathname ">b.c" "q")   => #P"Q:>b.c"
```

Note that this causes correct parsing of a TOPS-20 pathname when *thing* contains a device but no host and when *with-respect-to* is not **nil**. (Warning: If *thing* contains a device but no host and if *with-respect-to* is **nil** or not supplied, the device is interpreted as a host.) In the following example, X is a TOPS-20 host and A is a device:

```
(fs:parse-pathname "a:<b>c.d" "x") => #<TOPS20-PATHNAME "X:A:<B>C.D">
(fs:parse-pathname "a:<b>c.d")     => Error: "a" is not a known file
                                          server host.
```

In the same TOPS-20 example, if *with-respect-to* is **nil** and the host is be to taken from *defaults*, the pathname string must be preceded by a colon to be parsed correctly:

```
(fs:parse-pathname ":a:<b>c.d" nil "x:") => #<TOPS20-PATHNAME "X:A:<B>C.D">
(fs:parse-pathname "a:<b>c.d" nil "x:")  => Error: "a" is not a known file
                                                server host.
```

If *thing* is a list, *with-respect-to* is specified, and *thing* contains a host name, an error is signalled if the hosts from *with-respect-to* and *thing* are not the same.

merge-pathnames *pathname* &optional (*defaults* ***default-pathname-defaults***) (*default-version* **:newest**) *Function*

This function should be called to process a file name supplied by a user. It fills in unspecified components of the *pathname* from the *defaults* and returns a new pathname. Both *pathname* and *defaults* can be a pathname, stream, string, or symbol. The value returned is always a pathname.

defaults defaults to the value of ***default-pathname-defaults***. *default-version* defaults to **:newest**.

```
(setq myfile (pathname "myfile"))

 => #P"myfile"

(setq *tmp-dir-default-path*
      (make-pathname
        :host :local :device nil :directory "tmp" :name "foo" :type "lisp"))

 => #P"/tmp/foo.lisp"

(merge-pathnames myfile *tmp-dir-default-path*)

 => #P"/tmp/myfile.lisp"
```

fs:merge-pathnames *pathname* &optional (*defaults* **fs:*default-pathname-defaults***) (*default-version* **':newest**) *Function*

Fills in unspecified components of *pathname* from the defaults, and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, or a Maclisp name list. The returned value is always a pathname. The merging rules are documented elsewhere: See the section "Pathname Defaults and Merging".

If *pathname* is a string, it is parsed before merging. The default pathname is presented to **fs:parse-pathname** as a default pathname, from which the latter defaults the host if there is no explicit host named in the string.

defaults can be a pathname, a defaults alist, or a string. If it is a string, it is parsed against the default defaults. *defaults* defaults to the value of **fs:*default-pathname-defaults*** if unsupplied.

fs:merge-pathnames-and-set-defaults *pathname* &optional (*defaults* **fs:*default-pathname-defaults***) (*default-version* **':newest**) *Function*

The same as **fs:merge-pathnames** except that after it is done the result is stored back into *defaults*. This is handy for programs that have "sticky" defaults. (If *defaults* is a pathname rather than a defaults alist, then no storing back is done.) The optional arguments default the same way as in **fs:merge-pathnames**.

default-pathname-defaults *Variable*

If any pathname primitive needs a set of defaults and is not given one, it uses this set. In other words, this is the final source of pathname defaults. As a general rule, however, each program should have its own pathname defaults rather than relying on these defaults.

In CLOE, contains the system default pathname that includes a host component of **:local**, a null device field, an indicator for the current working directory in the directory field, default name and type of foo and lisp, respectively, and a version field of **:newest**.

```
(setq *default-pathname-defaults*
      (make-pathname
        :host :local :device nil :directory nil :name "myfile" :type "lsp"))
```

The following function is what programs use to complete a partially typed-in pathname.

fs:complete-pathname *defaults string type version* &rest *options* *Function*

string is a partially specified file name. (Presumably it was typed in by a user and terminated with the COMPLETE or END to request completion.) **fs:complete-pathname** looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded in a host-dependent fashion.

string is completed relative to a default pathname constructed from *defaults*, the host (if any) specified by *string*, *type*, and *version*, using the function **fs:default-pathname**. See the function **fs:default-pathname**. If *string* does not contain a colon, the host comes from *defaults*; otherwise the host name precedes the first colon in *string*.

options are keywords (without following values) that control how the completion will be performed. The following option keywords are allowed. Their meanings are explained more fully below.

:deleted	Look for files that have been deleted but not yet expunged. The default is to ignore such files.
:read or **:in**	The file is going to be read. This is the default. The name **:in** is obsolete and should not be used in new programs.
:write or **:print** or **:out**	The file is going to be written (that is, a new version is going to be created). The names **:print** and **:out** are obsolete and should not be used in new programs.
:old	Look only for files that already exist. This is the default. **:old** is not meaningful when **:write** is specified.
:new-ok	Allow either a file that already exists, or a file that does not yet exist. **:new-ok** is not meaningful when **:write** is specified. The **:new-ok** option is no longer used by any system software, because users found its effects (in the Zmacs command Find File (`c-X c-F`)) to be too confusing. It remains available, but programmers should consider this experience when deciding whether to use it.

The first value returned is always a string containing a file name; either the original string, or a new, more specific string. The second value returned indicates the status of the completion. It is non-**nil** if it was completely successful. The following values are possible:

:old The string completed to the name of a file that exists.

:new The string completed to the name of a file that could be created.

nil The operation failed for one of the following reasons:

- The file is on a file system that does not support completion. The original string is returned unchanged.
- There is no possible completion. The original string is returned unchanged.
- There is more than one possible completion. The string is completed up to the first point of ambiguity.
- A directory name was completed. Completion was not successful because additional components to the right of this directory remain to be specified. The string is completed through the directory name and the delimiter that follows it.

Although completion is a host-dependent operation, the following guidelines are generally followed:

When a pathname component is left completely unspecified by *string*, it is generally taken from the default pathname. However, the name and type are defaulted in a special way described below and the version is not defaulted at all; it remains unspecified.

When a pathname component is specified by *string*, it can be recognized as an abbreviation and completed by replacing it with the expansion of the abbreviation. This usually occurs only in the rightmost specified component of *string*. All files that exist in a certain portion of the file system and match this component are considered. The portion of the file system is determined by the specified, defaulted, or completed components to the left of this component. A file's component *x* matches a specified component *y* if *x* consists of the characters in *y* followed by zero or more additional characters; in other words, *y* is a left substring of *x*. If no matching files are found, completion fails. If all matching files have the same component *x*, it is the completion. If there is more than one possible completion, that is, more than one distinct value of *x*, there is an ambiguity and completion fails unless one of the possible values of *x* is equal to *y*.

If completion of a component succeeds, the system attempts to complete any additional components to the right. If completion of a component fails, additional components to the right are not completed.

A blank component is generally treated the same as a missing component; for example, if the host is a LMFS, completion of the strings "foo" and "foo." deals with the type component in the same way. The strings are not completed identically; completion of "foo" attempts to complete the name component, but completion of "foo." leaves the name component alone since it is not the rightmost.

If *string* does not specify a name, then the name of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, completion fails.

- With the **:write** option, the default name is always used when *string* does not specify a name, regardless of what files exist.

- With the **:new-ok** option, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, the default name is used.

The special treatment of the case where all files in the directory have the same name is not very useful and is not implemented by all file systems.

If *string* does not specify a type, then the type of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, completion fails.

- With the **:write** option, the default type is always used when *string* does not specify a type, regardless of what files exist.

- With the **:new-ok** option, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, the default type is used.

In file systems such as LMFS and UNIX that require a trailing delimiter (> or /) to distinguish a directory component from a name component, the system heuristically decides whether the rightmost component was meant to be a directory or a name, and inserts the directory delimiter if necessary.

If *string* contains a relative directory specification for a host with a hierarchical file system, it is assumed to be relative to the directory in the default pathname and is expanded into an absolute directory specification.

The host and device components generally are not completed; they must be fully specified if they are specified at all. This might change in the future.

If *string* does not specify a version, the returned string does not specify a version either. This differs from file name completion in TOPS-20; TOPS-20 completes an implied version of "newest" to a specific number. This is possible in TOPS-20 because completing a file name also attaches a "handle" to a file. In Genera, the version number of the newest file might change between the time the file name is completed and the time the actual file operation (open, rename, or delete) is performed.

A pathname component must satisfy the following rules in order to appear in a successful completion:

- The host, device, and directory must actually exist.

- The name must be the name of an existing file in the specified directory, unless **:write** or **:new-ok** is included in *options*.

- The type must be the type of an existing file with the specified name in the specified directory, unless **:write** or **:new-ok** is included in *options*.

- A pathname component always completes successfully if it is **:wild**.

When the rules are not satisfied by a component taken from the default pathname, completion fails and that component remains unspecified in the resulting string. When the rules are not satisfied by a component taken from *string*, completion fails and that part of *string* remains unchanged (other components of *string* can still be expanded).

These functions yield a pathname, given its components.

make-pathname &key *host device directory name type version defaults* *Function*

Constructs a pathname. It uses whatever components are supplied and fills in the missing components following the merging rules used by **merge-pathnames** and using the defaults from the **:defaults** argument.

The default value of the **:defaults** argument is a pathname whose host component is the same as the host component of ***default-pathname-defaults*** and whose other components are all **nil**.

```
(make-pathname :name "MYFILE" :type "LSP" :defaults my-defaults)
```

fs:make-pathname &rest *options* *Function*

Constructs a pathname. *options* are alternating keywords and values that specify the components of the pathname. Missing components default to **nil**, except the host (all pathnames must have a host). The **:defaults** option specifies the defaults to get the host from if none are specified. The other options allowed are **:host**, **:device**, **:directory**, **:name**, **:type**, **:version**, **:raw-device**, **:raw-directory**, **:raw-name**, **:raw-type**, **:canonical-type**.

The following functions are used to manipulate defaults alists directly.

fs:make-pathname-defaults *Function*

Creates a defaults alist initially containing no defaults. Asking this empty set of defaults for its default pathname before anything has been stored into it returns the file FOO on the user's home directory on the host to which the user logged in.

Defaults alists created with **fs:make-pathname-defaults** are remembered, and reset whenever the site is changed. This prevents remembered defaults from pointing to unknown hosts when world load files are moved between sites.

fs:copy-pathname-defaults *defaults* *Function*

Creates a defaults alist, initially a copy of *defaults*.

fs:default-pathname &optional *defaults host default-type default-version sample-p* *Function*

Obtains a pathname suitable for use as a default pathname and customizes it by modification of its type and version. It also extracts pathnames out of default alists.

The pathname returned by **fs:default-pathname** is always fully specified; that is, all components have non-**nil** values. This is needed when defaulting a pathname with **fs:merge-pathnames** to pass to **open** or other file-system operations, as these operations should always receive fully specified pathnames.

Specifying the optional arguments *host*, *default-type*, and *default-version* as not **nil** forces those fields of the returned pathname to contain those values. If *defaults*, which can be a pathname or a defaults alist, is not specified, the default defaults are used.

If *default-type* is a symbol representing a canonical type, that canonical type is used as the canonical type of the pathname returned. That is, the pathname has a type component that is the correct representation of that canonical type for the host.

Users should never supply the optional argument *sample-p*.

fs:set-default-pathname *pathname* &optional *defaults* *Function*

Updates a defaults alist. It stores *pathname* into *defaults*. If *defaults* is not specified, the default defaults are used.

The following functions return useful information.

pathnamep *x* *Function*

This predicate is **t** if *x* is a pathname and **nil** otherwise.

```
(pathnamep (pathname "foo")) => t

(pathnamep (open "foo" :direction :output)) => nil
```

directory *pathname* &key *:deleted* *Function*

Returns a list of pathnames that match the given *pathname*. The *pathname* argument can be a pathname, string, or a stream associated with a file. For a file that matches, the **truename** is returned. If no file matches, the function returns **nil**. Keywords such as **:wild** and **:newest** can be used in *pathname* to indicate the search space.

If **:deleted** is **nil**, which is the default, only pathnames of undeleted files are listed. If **:deleted** is **t**, then pathnames of deleted files are also listed.

```
(directory "/tmp")
 => (#P"/tmp/G5001.lisp" #P"/tmp/G5003.lisp" #P"/tmp/G5005.lisp")
```

:deleted is a Symbolics extension to Common Lisp, not available in CLOE.

pathname-device *pathname* *Function*

Returns the device component of *pathname*. *pathname* can be a pathname, string, or stream.

The returned value is a string or the symbol **:unspecific.** See the method **(flavor:method :device pathname)**.

Converts *object*,which may be a string, symbol, file-stream or already a pathname, to a pathname. The value of the converted *object* is returned.

```
(pathname "/usr/bin/appl.lisp") → #P"/usr/bin/appl.lisp"
```

See Also: CLtL 413, **truename**

pathname-directory *pathname* *Function*

Returns the directory component of *pathname*. *pathname* can be a pathname, string, or stream.

The returned value is a list of strings or keyword symbols. See the method **(flavor:method :directory pathname)**.

pathname-host *pathname* *Function*

Returns the host component of *pathname*. *pathname* can be a pathname, string, or stream.

The returned value is a host object. See the method **(flavor:method :host pathname)**.

pathname-name *pathname* *Function*

Returns the name component of *pathname*. *pathname* can be a pathname, string, or stream.

The returned value is a string or the symbol **:wild.** See the method **(flavor:method :name pathname)**.

pathname-type *pathname* *Function*

Returns the type component of *pathname*. *pathname* can be a pathname, string, or stream.

The returned value strings or the symbol **:wild**. See the method **(flavor:method :type pathname)**.

pathname-version *pathname* *Function*

Returns the version component of *pathname*. *pathname* can be a pathname, string, or stream.

The returned value is a number or symbol. See the method **(flavor:method :version pathname)**.

truename *x* *Function*

Finds the "truename" of the file associated with *x* within the file system. If *x* is an open stream already associated with a file in the file system, that file is used. The "truename" is returned as a pathname. An error is signalled if an appropriate file cannot be located within the file system for the given pathname.

truename can return a value different than **pathname**, since *truename* takes file links, logical devices, mapping of the "newest" version to a real version number, and other things into account.

An error is signalled if the file does not exist. The argument may be either a string, file-stream, symbol or pathname.

For example, a file may be opened with a partial string name for the file. The true pathname of the file associated with stream is returned, and **namestring** can be used for the string conversion.

```
(setq myfile (open "myfile" :direction :output))

(namestring (truename myfile)) => "/usr/me/myfile.lisp"
```

namestring *pathname* *Function*

Returns the full form of *pathname* as a string.

pathname can be a pathname, a string or symbol, or a stream that is or was open to a file. If it is a stream, the name returned represents the name used to open the file, which might not be the actual name of the file. It returns the result of **:string-for-printing** to **pathname**.

```
(setq bar (parse-namestring "/usr/myfile"))

 => #P"/usr/myfile"

(namestring bar) => "/usr/myfile"

(setq foo (open "myfile" :direction :output))

(namestring foo) => "/usr/me/myfile.lisp"
```

For more information see the function **truename**.

enough-namestring *pathname* &optional (*defaults* ***default-pathname-defaults***)
Function

Returns an abbreviated namestring that identifies the file named by *pathname* relative to *defaults*, which defaults to the value of ***default-pathname-defaults***. The following two forms must be equivalent in all cases:

```
(merge-pathnames (enough-namestring pathname defaults)
                 defaults)

(merge-pathnames (parse-namestring pathname nil defaults)
                 defaults)
```

The namestring returned by **enough-namestring** is generally the shortest string that satisfies this condition.

```
(setq foo (open "myfile" :direction :output))

(directory-namestring foo) => "myfile"
```

file-namestring *pathname* *Function*

Returns a string that represents the name, type, and version components of *pathname.*

pathname can be a pathname, a string or symbol, or a stream that is or was open to a file. If it is a stream, the name returned represents the name used to open the file, which might not be the actual name of the file. See the function **truename**. The name represented by *pathname* is returned as a namelist in canonical form.

Note that you cannot necessarily construct a valid namestring simply by concatenating the results of **file-namestring**, **directory-namestring**, and **host-namestring**. To obtain a full namestring: See the function **namestring**.

Under CLOE, there is no string representationof the version component.

```
(setq foo (open "myfile" :direction :output))

(file-namestring foo) => "myfile.lisp"
```

directory-namestring *pathname* *Function*

Returns a string that represents the directory-name component of *pathname.*

pathname can be a pathname, a string, a symbol, or a stream that is or was open to a file. If it is a stream, the name returned represents the name used to open the file, which might not be the actual name of the directory. See the function **truename**.

Note that you cannot necessarily construct a valid namestring simply by concatenating the results of **directory-namestring**, **file-namestring**, and **host-namestring**. To obtain a full namestring: See the function **namestring**.

```
(setq foo (open "myfile" :direction :output))

(directory-namestring foo) => "/usr/user-homedir"
```

host-namestring *pathname* *Function*

Returns a string that represents the host-name component of *pathname*. Note that this string can specify a logical rather than a physical host.

pathname can be a pathname, a string, or a stream that is or was open to a file. See the function **truename**.

Note that you cannot necessarily construct a valid namestring simply by concatenating the results of **host-namestring**, **file-namestring**, and **directory-namestring**. To obtain a full namestring: See the function **namestring**.

```
(setq foo (open "myfile" :direction :output))

(host-namestring foo) => "LOCAL"
```

file-author *file* *Function*

Returns the name of *file*'s author as a string, or **nil** if this cannot be determined. *file* can be a pathname, a string, or a stream that is open to a file.

```
(file-author "myfile.lisp") => "juser"
```

file-length *stream* *Function*

Returns a file's length as an integer, or **nil** if the length cannot be determined. *stream* must be a stream that is open to a file. For a binary file, the length is measured in units of the **:element-type** specified when the file was opened.

This function is also the maximum value for a file position.

```
(with-open-file (myfile "myfile.lisp" :direction :io)
  (format myfile "The length of ~A is ~A.~%"
          myfile (file-length myfile)))
```

file-write-date *file* *Function*

Returns the time, in universal time format, at which *file* was created or last written, or **nil** if this cannot be determined. *file* can be a pathname, a string, or a stream that is open to a file.

```
(multiple-value-bind (seconds minutes hours date month)
  (decode-universal-time (file-write-date "myfile.lisp"))
  (declare (ignore seconds minutes hours))
  (format nil "File write date myfile.lisp: ~A/~A" month date))
```

user-homedir-pathname &optional (*host* **fs:user-login-machine**) *Function*

Returns a pathname for the user's home directory on *host*. The home directory is the directory in which the user keeps personal files, such as lispm-init.lisp and mail files. If it is impossible to determine this information, the function returns **nil**. The function never returns **nil** when the *host* argument is not specified. The function returns a pathname without any name, type or version component. Those components are all **nil**.

```
(user-homedir-pathname) => #P"/usr/staff/techies/joeuser"
```

fs:user-homedir &optional *(host* **fs:user-login-machine***)* *Function*

Returns the pathname of the logged-in user's home directory on *host*, which defaults to the host the user logged in to. For a registered user (one who logged in without using the **:host** argument to **login**), the host is the user's **home-host** attribute. Home directory is a somewhat system-dependent concept, but from the point of view of the Symbolics computer it is usually the directory where the user keeps personal files such as init files and mail. This function returns a pathname without any name, type, or version component (those components are all **nil**).

fs:init-file-pathname *program-name* &optional *(canonical-type* **nil***)* *(host* **fs:user-login-machine***)* *Function*

Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the **login** function. *canonical-type* is the canonical type of the init file. It should be **nil** when the returned pathname is being passed to **load** so that **load** can look for a file of the appropriate type.

The following function defines a canonical file type.

fs:define-canonical-type *canonical-type default* &body *specs* *Special Form*

Defines a new canonical type. *canonical-type* is the symbol for the new type; *default* is a string containing the default surface type for any kind of host not mentioned explicitly. The body contains a list of specs that define the surface types that indicate the new canonical type for each host. The following example would define the canonical type **:lisp**.

```
(fs:define-canonical-type :lisp "LISP"
  ((:tops-20 :tenex) "LISP" "LSP")
  (:unix "L" "LISP")
  (:vms "LSP"))
```

For systems with more than one possible default surface form, the form that appears first becomes the preferred form for the type. Always use the interchange case.

Define new canonical types carefully so that they are valid for all host types. For example "com-map" would not be valid on VMS because it is both too long and contains an invalid character. You must define them so that the surface types are unique. That is, the same surface type cannot be defined to mean two different canonical types.

Canonical types that specify binary files must specify the byte size for files of the type. This helps **zl:copyf** and other system tools determine the correct byte size and character mode for files. You specify the byte size by attaching a **:binary-file-byte-size** property to the canonical type symbol. For example, the system defines the byte size of press files as follows.

```
(defprop :press 8. :binary-file-byte-size)
```

The following function is useful when dealing with canonical types. Unlike other functions described here, this function actually accesses and searches a host file system. This description is provided here for completeness. For functions and messages that actually access host file systems: See the section "Streams".

fs:find-file-with-type *pathname canonical-type* *Function*

Searches the file system to determine the actual surface form for a pathname object. Like **probef**, it returns the truename for *pathname*. When no file can be found to correspond to a pathname, it returns **nil**.

If *pathname* is a string, it is parsed against the default defaults to obtain an actual pathname object before processing.

canonical-type applies only when *pathname* has **nil** as its type component. **fs:find-file-with-type** searches the file system for any matching file with *canonical-type*. For example, on a TOPS-20 host, this would look first for ps:<gcw>toolkit.lisp and then for ps:<gcw>toolkit.lsp:

```
(fs:find-file-with-type (fs:parse-pathname "sc:<gcw>toolkit") ':lisp)
```

If it finds more than one file, it returns the one with the preferred surface type for *canonical-type* (or chooses arbitrarily if none of the files has the preferred surface type).

If *pathname* already had a type supplied explicitly, that overrides *canonical-type*. You can ensure that *canonical-type* applies by first setting the type explicitly:

```
(fs:find-file-with-type (send p ':new-type nil) ':lisp)
```

System programs that supply a default type for input files (for example, **load**) could use this mechanism for finding their input files.

The following functions are useful for poking around.

fs:describe-pathname *pathname* *Function*

If *pathname* is a pathname object, this describes it, showing you its properties (if any) and information about files with that name that have been loaded into the machine. If *pathname* is a string, this describes all interned pathnames that match that string, ignoring components not specified in the string. This is useful for finding the directory of a file whose name you remember. Giving **describe** a pathname object does the same thing as this function.

fs:pathname-plist *pathname* *Function*

Parses and defaults *pathname* then returns the list of properties of that pathname.

Pathname Messages

This section documents some of the messages a user can send to a pathname object. These messages are known as the *passive messages* to pathnames. They deal with inspecting and extracting components, constructing new pathnames based on old pathnames and new components, matching pathnames, and so forth. None of these messages actually interact with any host file system; they deal only with pathname objects within the Symbolics computer.

The other common, useful class of messages to pathnames are those that open, delete, and rename files, list directories, find and change file properties, and so forth. These are the *active messages* to pathnames. You usually do not send these messages directly, but use interface functions, such as **open**, **zl:probef**, **zl:deletef**, **zl:renamef**, **fs:directory-list**, **fs:file-properties**, and **fs:change-file-properties**. Neither these functions and messages, nor additional similar ones, are documented here. See the section "Streams".

Pathnames handle some additional messages that are intended to be sent only by the pathname system itself, and therefore are not documented here. Only someone who wanted to add a new type of file host to the system would need to understand those internal messages. This section also does not document messages that are peculiar to pathnames of a particular type of host.

(flavor:method :host pathname) *Method*

Returns the host component of the pathname. The returned value is always a host object. If the pathname is a logical pathname, the logical host is returned. It is an error to send **:host** to a logical host.

(flavor:method :device pathname) *Method*

Returns the device component of the pathname. The returned value can be **nil**, **:unspecific**, or a string. The string is in interchange case.

(flavor:method :directory pathname) *Method*

Returns the directory component of the pathname. The returned value can be **nil**, **:wild**, or a list of strings and symbols, each representing a directory level. These symbols can be **:wild** or **:wild-inferiors**. Single names of directories in nonhierarchical file systems are returned as single element lists. The strings are in interchange case.

(flavor:method :name pathname) *Method*

Returns the name component of the pathname. The returned value can be **nil**, **:wild**, or a string. The string is in interchange case.

(flavor:method :type pathname) *Method*

Returns the type component of the pathname. The returned value is always **nil**, **:unspecific**, **:wild**, or a string. The string is in interchange case.

(flavor:method :version pathname) *Method*

Returns the version component of the pathname. The returned value is always **nil**, **:wild**, **:unspecific**, **:oldest**, **:newest**, or a number.

(flavor:method :raw-device pathname) *Method*

Returns the device component of the pathname. The returned value can be **nil**, **:unspecific**, or a string. The string is in its raw case.

(flavor:method :raw-directory pathname) *Method*

Returns the directory component of the pathname. The returned value can be **nil**, **:wild**, or a list of strings and symbols, each representing a directory level. These symbols can be **:wild** or **:wild-inferiors**. Single names of directories in nonhierarchical file systems will be returned as single element lists. The strings are in their raw case.

(flavor:method :raw-name pathname) *Method*

Returns the name component of the pathname. The returned value can be **nil**, **:wild**, or a string. The string is in its raw case.

(flavor:method :raw-type pathname) *Method*

Returns the type component of the pathname. The returned value is always **nil**, **:unspecific**, **:wild**, or a string. The string is in its raw case.

(flavor:method :canonical-type pathname) *Method*

Determines the canonical type of a pathname and a surface representation for the type. It returns two values:

Value	*Meaning*
canonical type	This is either a keyword symbol from the set of known canonical types or a string (when the type component of the pathname is not a known canonical type). The string contains the type component from the pathname, in interchange case. original type This is **nil** when the type of the pathname is the same as the preferred surface type for the canonical type. See the special form **fs:define-canonical-type**. Otherwise, when the type differs from the preferred or default surface type, it is the original type in interchange case.

For example, for a UNIX pathname, sending the message **:canonical-type** to the following pathnames has these results:

Pathname	*Results from* **:canonical-type** *message*		
foo.l	**:lisp**	**nil**	Preferred surface type
foo.lisp	**:lisp**	"LISP"	Alternate surface type
foo.L	"l"	"l"	Not recognized
foo.LISP	"lisp"	"lisp"	Not recognized

Keep in mind that the **:canonical-type** message returns the type string in the interchange case rather than in the raw case.

(flavor:method :new-device pathname) *new-device* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the device component has been changed. The valid set of arguments to the **:new-device** message is the set of possible outputs of **:device**. See the method **(flavor:method :device pathname)**. A string value is expected to be in interchange case.

(flavor:method :new-directory pathname) *new-directory* *Method*

Returns a new pathname which is the same as the pathname it is sent to except that the value of the directory component has been changed. The valid set of arguments to the **:new-directory** message is the set of possible outputs of **:directory**. See the method **(flavor:method :directory pathname)**. String values are expected to be in interchange case.

(flavor:method :new-name pathname) *new-name* *Method*

Returns a new pathname which is the same as the pathname it is sent to except that the value of the name component has been changed. The valid set of arguments to the **:new-name** message is the set of possible outputs of **:name**. See the method **(flavor:method :name pathname)**. String values are expected to be in interchange case.

(flavor:method :new-type pathname) *new-type* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the type component has been changed. The valid set of arguments to the **:new-type** message is the set of possible outputs of **:type**. See the method **(flavor:method :type pathname)**. String values are expected to be in interchange case.

(flavor:method :new-version pathname) *new-version* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the version component has been changed. The valid set of arguments to the **:new-version** message is the set of possible outputs of **:version**. See the method **(flavor:method :version pathname)**.

(flavor:method :system-type pathname) *Method*

Returns the type of host that the pathname is intended for. This value is a keyword from the following set:

:its, **:lispm**, **:multics**, **:tenex**, **:tops-20**, **:unix**, **:vms**, **:logical**

This is the same set as returned by the **:system-type** message to a host object. It is not likely that you need to use this message directly.

(flavor:method :new-raw-device pathname) *dev* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the device component has been changed. The valid set of arguments to the **:new-raw-device** message is the set of possible outputs of **:raw-device**. See the method **(flavor:method :raw-device pathname)**. A string value is expected to be in its raw case.

(flavor:method :new-raw-directory pathname) *new-directory* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the directory component has been changed. The valid set of arguments to the **:new-raw-directory** message is the set of possible outputs of **:raw-directory**. See the method **(flavor:method :raw-directory pathname)**. String values are expected to be in their raw case.

(flavor:method :new-raw-name pathname) *new-name* *Method*

Returns a new pathname which is the same as the pathname it is sent to except that the value of the name component has been changed. The valid set of arguments to the **:new-raw-name** message is the set of possible outputs of **:raw-name**. See the method **(flavor:method :raw-name pathname)**. String values are expected to be in their raw case.

(flavor:method :new-raw-type pathname) *new-type* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the value of the type component has been changed. The valid set of arguments to the **:new-raw-type** message is the set of possible outputs of **:raw-type**. See the method **(flavor:method :raw-type pathname)**. String values are expected to be in their raw case.

(flavor:method :new-canonical-type pathname) *canonical-type* &optional *original-type* *Method*

Returns a new pathname based on the old one but with a new canonical type. *canonical-type* specifies the canonical type for the new pathname. The surface type of the new pathname is based on the default surface type of the canonical type, unless the pathname already had the correct type.

When the pathname object receiving the message already has the correct canonical type, the surface type in the new pathname depends on the presence of *original-type*. When *original-type* is omitted, the new pathname type has the same surface type as the old pathname. When *original-type* is supplied, the surface type for the new pathname is *original-type*. This assumes that *original-type* is a valid representation for *canonical-type*; if that assumption is not met, the *canonical-type* prevails and its default surface type is used.

canonical-type is a symbol for a known type, **:unspecific**, **nil**, or a string. Use a string for *canonical-type* to make pathnames with types that are not known canonical types.

The following examples assume that a pathname object for the file specification "vixen:/usr2/jwalker/mild.new" is the value of **m**.

```
(send m ':new-canonical-type ':lisp) =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
(send m ':new-canonical-type ':lisp "LISP") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.lisp">
(send m ':new-canonical-type ':lisp "MSS") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
(send m ':new-canonical-type "BAR" "BAR") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.bar">
(send m ':new-canonical-type ':lisp "lisp") =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
(send m ':new-canonical-type ':lisp nil) =>
#<UNIX-PATHNAME "VIXEN: //usr2//jwalker//mild.l">
```

(flavor:method :types-for-canonical-type pathname) *canonical-type* *Method*

The internal primitive for finding which surface types correspond to *canonical-type*. Normally you would not use this directly. To determine what form of a pathname exists in a file system: See the function **fs:find-file-with-type**.

(flavor:method :new-pathname pathname) &rest *options* *Method*

Returns a new pathname that is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are **:host**, **:device**, **:directory**, **:name**, **:type**, **:version**, **:raw-name**, **:raw-device**, **:raw-type**, **:raw-directory**, and **:canonical-type**. The **:type** argument also accepts a symbol as an argument, implying canonical type. See the section "Canonical Types in Pathnames".

(flavor:method :new-default-pathname pathname) &rest *options* *Method*

Returns a new valid pathname based on the one receiving the message, using the pathname components supplied by *options*. The components do not need to be known to be valid on a particular host. The method uses the components "as suggestions" for building the new pathname; it is free to make substitutions as necessary to create a valid pathname. It is heuristic, not algorithmic, so it does not necessarily yield valid semantics. The heuristics used, however, seem to produce pathnames that match what many people expect from cross-host defaulting.

It always produces a pathname with valid syntax but not necessarily valid semantics. For example, when it tries to map between a hierarchical file system and a nonhierarchical file system, it uses the least significant of the hierarchical components as the directory component. Sometimes this is not correct, but in all cases it is syntactically valid. The main applications for **:new-default-pathname** are in producing defaults to offer to the user and in copying components from one kind of pathname to another.

Application notes: **:new-pathname** always does what its arguments specify; it never uses heuristics. Thus **:new-pathname** could signal an error in certain cross-host situations where **:new-default-pathname** would not have any problems. Usually, user programs should use **fs:default-pathname**, which sends **:new-default-pathname** as part of its operation. However, if you are copying a single component from one kind of pathname to another, **:new-default-pathname** is the right tool.

For example, the right way to copy the version from an input pathname to an output pathname is as follows:

```
(defun copy-version (input-pathname output-pathname)
  (send output-pathname :new-default-pathname
        :version (send input-pathname :version)))
```

If the above example used **:new-pathname** or **:new-version**, the input pathname were a UNIX pathname, and the output were a LMFS pathname, this example would signal an error, since **:unspecific** is not a valid version in a LMFS pathname. However, using **:new-default-pathname**, the closest equivalent is substituted, namely **:newest**.

(flavor:method :parse-truename pathname) *string* &optional (*from-filesystem* **t**) *Method*

Returns the pathname corresponding to the *string* argument. The *string* is parsed, with the pathname supplying the defaults (notably, the host). The method is useful when, for example, a remote file system produces a string naming a file, and you want the corresponding pathname.

(flavor:method :generic-pathname pathname) *Method*

Returns the generic pathname for the family of files of which this pathname is a member. See the section "Generic Pathnames".

The following messages get a pathname string out of a pathname object:

(flavor:method :string-for-printing pathname) *Method*

Returns a string that is the printed representation of the pathname. This is the same as what you get if use **princ** or **string** on the pathname. It is the native host form of the pathname string, preceded by the name of the host and colon. This is the preferred user-visible printed representation of pathnames.

(flavor:method :string-for-wholine pathname) *Method*

Returns a string that can be compressed in order to fit in the status line.

(flavor:method :string-for-editor pathname) *Method*

Returns a string that is the pathname with its components rearranged so that the name is first. The editor uses this form to name its buffers.

(flavor:method :string-for-dired pathname) *Method*

Returns a string to be used by the directory editor. The string contains only the name, type, and version.

(flavor:method :string-for-host pathname) *Method*

Returns a string that is the pathname in the form preferred by the host file system.

(flavor:method :string-for-directory pathname) *Method*

Returns a string suitable for describing the directory portion of the pathname, in the format that users of the host system are used to seeing it. The host name is not included.

The following messages manipulate the property list of a pathname:

(flavor:method :get pathname) *indicator* *Method*

Manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists".

Be careful using property lists of pathnames. See the section "Pathnames".

(flavor:method :getl pathname) *list-of-indicators* *Method*

Manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists". Please take care in using property lists of pathnames. See the section "Pathnames".

(flavor:method :putprop pathname) *value indicator* *Method*

Manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists". Please take care in using property lists of pathnames. See the section "Pathnames".

(flavor:method :remprop pathname) *indicator* *Method*

Manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists". Please take care in using property lists of pathnames. See the section "Pathnames".

(flavor:method :plist pathname) *Method*

Manipulates the pathname's property list analogously to the function of the same name, which does not (currently) work on instances. See the section "Property Lists".

The following messages can be sent to pathnames having wildcard components or suspected of having wildcard components:

(flavor:method :pathname-match pathname) *candidate-pathname* &optional *(match-host t)* *Method*

Determines whether *candidate-pathname* would satisfy the wildcard pattern of the pathname receiving the message. (The pathname receiving the message is assumed to be one that would satisfy **:wild-p**.) It compares corresponding components in the pattern pathname and *candidate-pathname*. It returns **nil** when *candidate-pathname* does not satisfy the pattern; otherwise it returns something other than **nil**.

match-host determines whether it requires the host component of the pattern to match as well. When *match-host* is **nil**, it ignores the host component. By default, it does require that the host component match.

A pattern pathname containing no wild components matches only itself.

If the *candidate-pathname* specifies a physical host, and the message is sent to a logical pathname, the physical host is "back-translated," if possible.

(flavor:method :wild-p pathname) *Method*

A predicate that determines whether the pathname is syntactically a wildcard pathname. This means that any component is **:wild**, or, for most systems, contains the character *, or that the directory component has any of the valid forms of directory wildcard in it. See the method **(flavor:method :directory-wild-p pathname)**.

Value	*Meaning*
nil	No component of the name is syntactically a wildcard.
not **nil**	One or more components of the name are syntactically wild. The actual value in this case is the symbol for the most significant wild component: **:device**, **:directory**, and so on.

(flavor:method :device-wild-p pathname) *Method*

If the device component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned.

(flavor:method :directory-wild-p pathname) *Method*

If the directory component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. All forms of wildcard at each directory level for hierarchical file systems, as well as **:wild-inferiors**, are recognized as constituting a wildcard directory component. Otherwise, **nil** is returned.

(flavor:method :name-wild-p pathname) *Method*

If the name component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. Otherwise, **nil** is returned.

(flavor:method :type-wild-p pathname) *Method*

If the type component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. Otherwise, **nil** is returned.

(flavor:method :version-wild-p pathname) *Method*

If the version component of this pathname is a recognized wildcard for the system type concerned, or **:wild**, a non-**nil** is returned. Otherwise, **nil** is returned.

(flavor:method :translate-wild-pathname pathname) *target-pattern-pathname starting-pathname* *Method*

Produces a new pathname based on *starting-pathname* and the analogies between the pathname receiving the message and *target-pattern-pathname*.

:translate-wild-pathname examines the correspondences between *target-pattern-pathname* and the pathname receiving the message. It then does whatever is necessary to *starting-pathname* to transform it into the target pathname.

It checks to be sure *starting-pathname* matches the pathname receiving the message and signals **zl:ferror** if they do not match. A standard way for generating *starting-pathname* is to send **:directory-list** to the source pattern pathname to generate a set of starting pathnames.

Pathnames on Supported Host File Systems

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncrasies.

LMFS Pathnames

LMFS is an acronym for Lisp Machine File System, which is the native file system of the Symbolics computer. It is only one of many possible file systems accessible from the Symbolics computer.

LMFS is a hierarchical file system. Every file has a name, type, and version. Names are virtually unlimited in length (hundreds of characters), but a performance penalty is imposed for names of over 30 characters. Types are limited to 14 characters. File versions are supported. There is no limit to the depth of directories. There are no devices (**:device** to a LMFS pathname always returns **:unspecific**).

A LMFS pathname looks as follows:

```
>dir>ectory>name.type.version
```

The greater-than (">") character separates directory levels. Absolute pathnames always start with greater-than's. Pathnames that specify no directory, relative or otherwise, contain no greater-than's. For example:

```
foo.bar.7
```

The topmost directory of the directory tree (the *ROOT* directory) is indicated by the absence of directory names but the continued presence of a greater-than. For example, the following is a file named foo.bar, version 7, in the ROOT directory:

```
>foo.bar.7
```

No file type abbreviations are needed for LMFS.

File and directory names in LMFS can be stored in upper, lower, or mixed case. Lowercase is the preferred case. Case is ignored on lookup.

Due to problems with interning of pathnames it is sometimes difficult to control the casing of a LMFS pathname, and it is almost always impossible to change it once established. See the section "Interning of Pathnames".

A version component of **:newest** is represented by the string "newest". A version component of **:oldest** is represented by the string "oldest".

Upward relativization in relative directory specifications is designated by a pathname starting with the character less-than ("<"). All and only all absolute pathnames start with the character greater-than (">"). Downward relativization is indicated by a pathname, which although it contains greater-than's, does not start with one. For example, the following specifies a directory named foo, inferior to the superior directory of the directory of the default pathname with which it is merged.

```
<foo>x.y
```

LMFS directories, when referenced as files, have a file type of "directory" and a version of 1. See the section "Directory Pathnames and Directory Pathnames as Files".

The following example specifies a directory named bar, inferior to the directory of the default pathname with which it is merged.

```
bar>x.y
```

LMFS supports recursive directory level matching (**:wild-inferiors**). The representation of **:wild-inferiors** in LMFS is **. Any number of ** components can appear in wildcard pathnames as directory levels, and need not be in trailing positions. (The further it gets from the trailing end of the directory name, however, the more expensive it gets to compute.) Here are some examples of the use of **:

Pathname	*What it means*
>**>*.lisp.newest	All the newest lisp files on the whole file system.
>**>*>secret>*.*.*	All files in subdirectories (but not top-level directories) named "secret".
>lmach>**>*.*.newest	All the newest files in >lmach and all its subdirectories.

A component of **:wild**, in any component except the directory component, is represented by *. *, when accompanied by other characters, such as in foo*bar*, matches zero or more characters, as a wildcard. Although * or names containing * are valid as directory-level component names, a directory component of **:wild** cannot be specified through pathname syntax. This is because "any directory at all" is represented by **(:wild-inferiors)**. A directory name given as * is a specification for a relative pathname, any subdirectory of the directory of the pathname which is merged. That is represented internally as **(:wild)**, not **:wild**.

The name of the ROOT directory, as a file (its "directory pathname as file") is

```
>The Root Directory.directory.1
```

Names of files stored in the Lisp Machine File System can not contain *. This restriction is necessary because * is used consistently to indicate wildcards in pathnames.

You can not access files whose names contain * as a character. A special function allows you to rename any file or directories whose names contain *.

lmfs:rename-local-file-tool *from-path to-path* *Function*

Renames a file in which * appears in one of the pathname components. This function works locally only; you must run it on the machine in whose file system the file is stored. It does not rename a file across the network.

from-path and *to-path* must be pathnames or strings coercible to pathnames. *from-path* is parsed against a default on the local host. *to-path* is parsed against *from-path* as the default. The version number for *to-path* is inherited from the file being renamed. Any version number appearing in *to-path* is ignored.

```
(lmfs:rename-local-file-tool ">AUser>*secret-stuff*" "-secret-stuff-")
(lmfs:rename-local-file-tool ">*special*.directory.1" "-special-")
```

FEP File System Pathnames

The syntax of FEP file system pathnames is identical to that of LMFS pathnames, and the semantics are the same as well. For more information: See the section "LMFS Pathnames".The following differences are to be noted.

- The maximum length of a file name is 32 characters.
- The maximum length of file types is 4 characters.
- The type of directories is "DIR".

The name of the ROOT directory, as a file (its "directory pathname as file") is:

```
>ROOT-DIRECTORY.DIR.1
```

UNIX Pathnames

Since UNIX file names can be no longer than 14 characters, the representations of most canonical types are stored in abbreviated form, according to the following table. Other values are represented as they are.

Canonical type	*UNIX abbreviation(s)*
:LISP	"l" "lisp"
:TEXT	"tx" "text" "txt"
:MIDAS	"md"
:QFASL	"qf" "qfasl"
:QBIN	"qb" "qbin"

:BIN	"bn" "bin"
:PRESS	"pr" "press"
:LGP	"lg" "lgp"
:PATCH-SYSTEM-DIRECTORY	"sd"
:PATCH-VERSION-DIRECTORY	"pd"
:BABYL	"bb" "babyl"
:XMAIL	"xm" "xmail"
:MAIL	"ma" "mail"
:RMAIL	"rm"
:ZMAIL-TEMP	"_z" "_zmail"
:GMSGS-TEMP	"_g" "_gmsgs"
:UNFASL	"uf" "unfasl"
:OUTPUT	"ot" "output"
:ULOAD	"ul" "uload"
:MCR	"mc" "mcr"
:SYM	"sm" "sym"
:TBL	"tb" "tbl"
:MICROCODE	"mic"
:ERROR-TABLE	"err"
:FEP-LOAD	"flod"
:SYNC-PROGRAM	"sn" "sync"
:CWARNS	"cw" "cwarns"
:SYSTEM	"sy" "system"
:FONT-WIDTHS	"wd" "widths"
:BFD	"bfd"
:KST	"kt" "kst"
:AST	"at" "ast"
:PLT	"pl" "plt"
:DRW	"drw"
:WD	"wd"
:DIP	"dip"
:SAV	"sav"
:MAP	"map"
:CONSOLIDATED-MAP	"cm"
:TAGS	"tg" "tags"
:PALX-BIN	"pb" "pbin"
:XGP	"xg" "xgp"
:LIL	"ll" "lil"
:SAR	"sar"
:SAB	"sab"
:MSS	"mss" "ms"
:FORTRAN	"f"
:LOGICAL-PATHNAME-TRANSLATIONS	"lt" "logtran"

:LOGICAL-PATHNAME-DIRECTORY-TRANSLATIONS	
	"ld" "logdir"
:NULL-TYPE	:unspecific ""
:FILES	"fl"
:COLD-LOAD	"load"
:PXL	"px" "pxl"
:IMAGE	"im" "image"
:DUMP	"dm" "dump"

As is true with the canonical type mechanism in general, files having the canonical type spelled in full are also recognized as being of that canonical type.

Logical pathname translation must get around the restrictions in UNIX pathnames. When translating logical pathnames an extra translation step is invoked, in some cases, as for VAX/VMS pathnames.

The preferred case on UNIX is lowercase. Pathname components presented to **:new-directory**, **:new-name**, and so forth, are case-inverted in most instances. See the section "Case in Pathnames".

UNIX 4.2 Pathnames

UNIX 4.2 uses slightly different representations of some canonical types than do other versions of UNIX. In most cases, the representations are the same as for LMFS, but the UNIX versions are also allowed.

Canonical type	*UNIX 4.2 abbreviation(s)*
:LISP	"lisp" "l"
:TEXT	"text" "tx" "txt"
:MIDAS	"midas" "md"
:QFASL	"qfasl" "qf"
:QBIN	"qbin" "qb"
:BIN	"bin" "bn"
:PRESS	"pr" "press"
:LGP	"lgp" "lg"
:PATCH-SYSTEM-DIRECTORY	
	"system-dir" "sd"
:PATCH-VERSION-DIRECTORY	
	"patch-dir" "pd"
:BABYL	"babyl" "bb"
:XMAIL	"xmail" "xm"
:MAIL	"mail" "ma"
:RMAIL	"rmail" "rm"
:ZMAIL-TEMP	"_zmail" "_z"
:GMSGS-TEMP	"_gmsgs" "_g"
:UNFASL	"unfasl" "uf"
:OUTPUT	"output" "ot"
:ULOAD	"uload" "ul"
:MCR	"mcr" "mc"

:SYM	"sym" "sm"
:TBL	"tbl" "tb"
:MICROCODE	"mic"
:ERROR-TABLE	"err"
:FEP-LOAD	"flod"
:SYNC-PROGRAM	"sync" "sn"
:CWARNS	"cwarns" "cw"
:SYSTEM	"system" "sy"
:FONT-WIDTHS	"widths" "wd"
:BFD	"bfd"
:AC	"ac"
:AL	"al"
:KS	"ks"
:KST	"kst" "kt"
:AST	"ast" "at"
:PLT	"pl" "plt"
:DRW	"drw"
:WD	"wd"
:DIP	"dip"
:SAV	"sav"
:MAP	"map"
:CONSOLIDATED-MAP	"con-map" "cm"
:TAGS	"tags" "tg"
:PALX-BIN	"palx_bin" "pbin" "pb"
:XGP	"xgp" "xg"
:LIL	"lil" "ll"
:FORTRAN	"f"
:SAR	"sar"
:SAB	"sab"
:MSS	"mss" "ms"
:LOGICAL-PATHNAME-TRANSLATIONS	"logtran" "lt"
:LOGICAL-PATHNAME-DIRECTORY-TRANSLATIONS	"translations" "logdir" "ld"
:NULL-TYPE	:unspecific ""
:COLD-LOAD	"load"
:FILES	"files" "fl"
:PXL	"pxl" "px"
:IMAGE	"image" "im"
:DUMP	"dump" "dm"

As is true with the canonical type mechanism in general, files having the canonical type spelled in full are also recognized as being of that canonical type.

Logical pathname translation must get around the restrictions in UNIX pathnames. When translating logical pathnames, an extra translation step is invoked as for VAX/VMS pathnames.

The preferred case on UNIX is lowercase. Pathname components presented to **:new-directory**, **:new-name**, and so forth, are case-inverted in most instances. See the section "Case in Pathnames".

VAX/VMS Pathnames

A VAX/VMS version 4.4 pathname looks as follows:

```
[DIR.ECTORY.COM.PONENTS]NAME.TYP;VERSION
```

The semicolon character is the standard delimiter for the version number. Because of it, a version can be specified even though the name and type are omitted. For compatibility with other Digital Equipment Corporation systems, however, a period is also accepted as a version delimiter when name and type are supplied.

Device is specified by a device name followed by a colon preceding the pathname. You must take great caution with pathnames specifying devices so as not to confuse the pathname parser about host identity. See the section "Host Determination In Pathnames".

Uppercase is the only supported alphabetic case. Pathnames typed in lowercase are converted to uppercase on input.

Here is a list of canonical types, their VAX/VMS representations, their default byte-size used for a binary transfer, and whether records are stored in fixed- or variable-length format:

Canonical type	*VMS representation*	*Byte-size*	*Format*
:LISP	"LSP"		
:TEXT	"TEXT" "TXT"		
:MIDAS	"MID"		
:QFASL	"QFS"	16	var
:QBIN	"QBN"	16	var
:BIN	"BIN"	16	var
:PRESS	"PRS"	8	fix
:PATCH-SYSTEM-DIRECTORY	"SPD"		
:PATCH-VERSION-DIRECTORY	"VPD"		
:BABYL	"BAB"		
:XMAIL	"XML"		
:MAIL	"MAI"		
:RMAIL	"RML"		
:ZMAIL-TEMP	"ZMT"		
:GMSGS-TEMP	"GMT"		
:UNFASL	"UNF"		
:OUTPUT	"OUT"		
:ULOAD	"ULD"		
:MCR	"MCR"		
:SYM	"SYM"		

:TBL	"TBL"		
:MICROCODE	"MIC"	8	var
:ERROR-TABLE	"ERR"		
:FEP-LOAD	"FLD"		
:SYNC-PROGRAM	"SYN"		
:CWARNS	"CWN"		
:SYSTEM	"SYD"		
:FONT-WIDTHS	"WID"	16	fix
:BFD	"BFD"	16	var
:KST	"KST"	9	
:AC	"AC"	16	
:AL	"AL"	16	
:KS	"KS"	16	
:AST	"AST"		
:PLT	"PLT"	9	
:DRW	"DRW"	12	
:WD	"WD"	12	
:DIP	"DIP"	12	
:SAV	"SAV"	12	
:MAP	"MAP"		
:CONSOLIDATED-MAP	"CON"		
:TAGS	"TAG"		
:PALX-BIN	"PXB"	8	var
:XGP	"XGP"		
:LIL	"LIL"		
:FOR	"FOR"		
:SAR	"SAR"		
:SAB	"SAB"	8	
:MSS	"MSS"		
:LOGICAL-PATHNAME-TRANSLATIONS	"LTR"		
:LOGICAL-PATHNAME-DIRECTORY-TRANSLATIONS	"LDT"		
:NULL-TYPE	""		
:COLD-LOAD	"LOD"	16	var
:FILES	"FLS"		
:PXL	"PXL"	8	
:IMAGE	"IMG"		
:DUMP	"IDM"	16	

Different versions of VAX/VMS have different restrictions on pathnames. For example, VAX/VMS version 3 allows neither the underscore character nor a hyphen in pathnames. Versions 4.0 through 4.3 allow the underscore but not the hyphen. Version 4.4 allows the hyphen. Also, version 3 had a strict restriction on the length of filenames, which later versions relaxed. The Symbolics logical pathname translation works differently depending on the version of VAX/VMS running on the host. This information is stored in the System Type attribute of the host object for the VAX. Briefly, the differences are as follows:

System Type	*Logical Pathname Translation to VAX/VMS*
vms4.4	VAX/VMS version 4.4 and later versions: translation does not compress file names, and does not make changes for the hyphen or underscore, since both are supported by VAX/VMS.
vms4	VAX/VMS versions 4.0, 4.1, 4.2, and 4.3: translation substitutes hyphens for the underscore character, but does not compress file names.
vms	VMS versions prior to version 4: translation to the VAX/VMS side compresses file names, and removes underscores and hyphens.

The VAX/VMS pathname mechanism supports recursive directory matching (**:wild-inferiors**). The representation for a directory level component of **:wild-inferiors** is ".."; however, it can appear only at the end of a directory name. Thus, the following matches any file in [A.B] or any of its subdirectories:

```
[A.B...]*.*.*
```

Upward relativization in pathnames is specified by one or more minuses ("-") as the first directory name. Downward relativization is represented by a null (0-character) first directory name. For example, the following specifies a directory named FOO, inferior to the superior directory of the directory of the default pathname with which it is merged.

```
[-.FOO]X.Y
```

A pathname version component of **:newest** is specified by a version of 0 in the filename string. There is no VAX/VMS implementation of **:oldest**.

The percent sign (%) can be used in VAX/VMS wildcards to specify the matching of a single character.

The pathname system does not recognize logical device names. They are specified as device names and are resolved by VAX/VMS, not the pathname system. Defaulting the directory specification of VAX/VMS pathnames when logical devices are used can cause problems.

VAX/VMS directories, when referenced as files, have a type of "DIR" and a version of 1. See the section "Directory Pathnames and Directory Pathnames as Files".

TOPS-20 and TENEX Pathnames

A TOPS-20 pathname has the form:

```
HOST:DEVICE:<DIRECTORY>NAME.TYPE.VERSION
```

The default device is PS:.

TOPS-20 pathnames are mapped to uppercase. Special characters (including lowercase letters) are quoted with the circle-X (⊗) character, which has the same character code in the Symbolics character set as control-V in the TOPS-20 character set.

TOPS-20 pathnames represent versions of **:oldest** and **:newest** by the strings "..-2" and "..0", respectively.

The directory component of a TOPS-20 pathname is a list of directory level components. The directory <FOO.BAR> is represented as the list ("FOO" "BAR").

The TOPS-20 init file naming convention is "<user>program.INIT".

When there is not enough room in the status line to display an entire TOPS-20 file name, the name is truncated and followed by a center-dot character to indicate that there is more to the name than can be displayed.

TENEX pathnames are almost the same as TOPS-20 pathnames, except that the version is preceded by a semicolon instead of a period, the default device is DSK instead of PS, and the quoting requirements are slightly different.

ISO 9660 Pathnames

An ISO 9660 pathname looks like:

host`|CDROM`*n*`:>`*dir1*`>`*dir2*`>...>`*file*`.`*type*`;`*version*

Directory names can be up to 31 characters long. The file name and type together may be up to 30 characters long. If the CD-ROM drive is attached to the local machine, "*hostl*" is optional. The version is an integer between 1 and 32767, or "NEWEST", or "OLDEST". You can use "*" in any position for a wild card. All names must consist entirely of digits, upper-case letters, and underscore "_" characters.

CD-ROM character files are expected to be in "Unix ASCII" format with NL characters separating the lines of text.

Multics Pathnames

Multics possesses a hierarchical file system. Every file has a name, and might or might not have a type. Multics does not support file versions. The sum of the lengths of name and type and the period required to separate them must not exceed 32 characters. A maximum of 16 directory levels is supported. There are no devices (**:device** to a Multics pathname always returns **:unspecific**). A Multics pathname looks as follows:

```
>dir>ectory>name.type
```

The greater-than (">") character separates directory levels. Absolute pathnames always start with greater-than's. Pathnames that specify no directory, relative or otherwise, contain no greater-than's, for example:

```
foo.bar
```

The topmost directory of the directory tree (the *ROOT* directory) is indicated by the absence of directory names but the continued presence of a greater-than. For example, the following is a file named foo.bar, in the ROOT directory:

```
>foo.bar
```

No file type abbreviations are needed for Multics.

File and directory names can be stored in upper, lower, or mixed case. Lowercase is the preferred case. Case is significant: Foo, FOO, and foo could be the names of three different files in the same directory.

Upward relativization in relative directory specifications is designated by a pathname starting with the character less-than ("<"). All and only all absolute pathnames start with the character greater-than (">"). Downward relativization is indicated by a pathname, which although it contains greater-than's, does not start with one. For example, the following specifies a directory named foo, inferior to the superior directory of the directory of the default pathname with which it is merged.

```
<foo>x.y
```

Multics directories, when referenced as files, have no specific type; they need not have any type at all. See the section "Directory Pathnames and Directory Pathnames as Files".

The following example specifies a directory named bar, inferior to the directory of the default pathname with which it is merged.

```
bar>x.y
```

Multics does not support **:wild-inferiors**, that is, recursive directory-level matching. For that matter, Multics does not support *any* form of wildcard in the directory component of a pathname. (Although :pathname-match matches such components, Multics does not support them in directory lists.) A component of **:wild**, in any component except the directory component, is represented by *. *, when accompanied by other characters, such as in foo*bar*, matches zero or more characters, as a wildcard.

ITS Pathnames

An ITS pathname looks like "HOST: DEVICE: DIR; FOO 69". The default device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: can be used.

ITS does not exactly fit the virtual file system model, in that a file name has two components (FN1 and FN2) rather than three (name, type, and version). Consequently to map any virtual pathname into an ITS filename, it is necessary to choose whether the FN2 will be the type or the version. The rule is that usually the type goes in the FN2 and the version is ignored; however, certain types (LISP and TEXT) are ignored and instead the version goes in the FN2. Also if the type is **:unspecific** the FN2 is the version.

An ITS filename is converted into a pathname by making the FN2 the version if it is "<", ">", or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols **:oldest** and **:newest**, which correspond to "<" and ">" respectively. If a version is specified, the type is always **:unspecific**. If a type is specified, the version is **:unspecific** so that it does not override the type.

Each component of an ITS pathname is mapped to uppercase and truncated to six characters.

Special characters (space, colon, and semicolon) in a component of an ITS pathname can be quoted by prefixing them with right horseshoe (⊃) or equivalence sign (≡). Right horseshoe is the same character code in the Symbolics character set as control-Q in the ITS character set.

The ITS init file naming convention is "homedir; user program".

fs:*its-uninteresting-types* *Variable*

The ITS file system does not have separate file types and version numbers; both components are stored in the "FN2". This variable is a list of the file types that are "not important"; files with these types use the FN2 for a version number. Files with other types use the FN2 for the type and do not have a version number.

It is not possible to have two ITS pathnames with the same meaning that differ in an ignored component. **fs:*its-uninteresting-types*** controls which types are ignored in favor of retaining version numbers. The following table summarizes the interaction of type and version components for ITS pathnames.

Type	*Version*	*Result*
supplied	omitted	type is retained, version is **:unspecific**
omitted	supplied	type is **:unspecific**, version is retained
"interesting"	supplied	type is retained, version is **:unspecific**
"uninteresting"	supplied	type is **:unspecific**, version is retained

(flavor:method :fn1 fs:its-pathname) *Method*

Returns a string that is the FN1 host-dependent component of the pathname.

(flavor:method :fn2 fs:its-pathname) *Method*

Returns a string that is the FN2 host-dependent component of the pathname.

MS-DOS Pathnames

An MS-DOS pathname looks like this:

```
    HOST:DEVICE:\DIR\ECTORY\NAME.TYPE
```

The default device is C:. Uppercase is the only supported case. Pathnames typed in lowercase are converted to uppercase on input.

File names and directory components are restricted to eight characters. File types are restricted to three characters. The canonical types for MS-DOS are the same as for VAX/VMS.

Relative pathnames are permitted. Upward-level changes are signalled with "..". For example:

```
PC:A:..\..\DIR\FILE.LSP
```

Syntax for Logical Pathnames

A logical pathname has the form

```
HOST: DIRECTORY; NAME.TYPE.VERSION
```

In logical pathnames, dots separate the filename, type, and version. There is no way to specify a device within a logical pathname. When a logical pathname is parsed, a pathname is returned whose device component is **:unspecific**. Logical pathnames can be hierarchical; use semicolons to separate directory levels.

Logical pathnames can also be relative. That is, they can contain a directory component whose meaning is "when merging against a default, append this". The syntax for this is

```
HOST: ; DIRECTORY; NAME.TYPE.VERSION
```

Notice the semicolon [;] that is placed before the directory component. The previous pathname, merged against a default of

```
HOST: USER; FOO.LISP.NEWEST
```

would yield this:

```
HOST: USER; DIRECTORY; NAME.TYPE.VERSION
```

The equivalence-sign character (≡) can be used for quoting special characters such as spaces and semicolons. (The use of this character is discouraged, however, as files named using it will probably not be transportable). The double-arrow character (↔) can be used as a place-holder for unspecified components. The **:newest**, **:oldest**, and **:wild** values for versions are specified with the strings NEWEST, OLDEST, and * respectively. On input, **:newest** can be represented by > and **:oldest** by <.

There is no init file naming convention for logical hosts; you cannot log in to them. The **:string-for-host**, **:string-for-wholine**, **:string-for-dired**, and **:string-for-editor** messages are all passed on to the translated pathname, but the **:string-for-printing** is handled by the **fs:logical-pathname** flavor itself and shows the logical name.

Wildcard Matching in Logical Pathnames

The system can match any directory or subdirectory, at any level. For example, you can ask the Show Directory command to list all font files anywhere in the SYS hierarchy like this:

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

Wildcards in logical pathnames correspond to the >**> syntax for LMFS pathnames, the [name...] syntax for VAX/VMS file specifications, and the /**/ syntax in UNIX file specifications. See the section "LMFS Pathnames". This makes it easy to specify logical pathname translations on Symbolics computers, VAX/VMS, and

UNIX. For example:

```
;;; -*- Mode: LISP; Syntax:  Common-lisp; Package:  USER -*-

(fs:set-logical-pathname-host "SYS" :translations
   '(("**;" "ACME-SMBX:>Rel-8-0>sys>**>")))

(fs:set-logical-pathname-host "SYS"
  :translations
  '(("SYS:**;*.*.*" "ACME-VMS:SYMBOLICS:[REL8-0...]*.*;*"))
  :no-translate nil)
```

Consider this example:

```
;;; -*- Mode: LISP; Syntax:  Common-lisp; Package:  USER -*-

(fs:set-logical-pathname-host "SYS" :translations
  '(("**;" "ACME-VMS:[SYMBOLICS.REL-8-0.SYS...]")))
```

Consider the following UNIX example:

```
;;; -*- Mode:  LISP; Syntax:  Common-lisp; Package: USER -*-

(fs:set-logical-pathname-host "SYS" :translations
  '(("**;" "ACME-UNIX:/usr/share/symbolics/rel-8-0/sys.sct/**/")))
```

Note: Wherever a double asterisk [**] appears in a logical-host's pathname, a corresponding "wild-inferiors" pathname must exist in the physical-host's pathname.

For more information about LMFS and VAX/VMS pathnames, see the section "LMFS Pathnames" and see the section "VAX/VMS Pathnames".

Init-File Naming Conventions

Init files are of canonical type **:lisp** for source files and **:bin** for compiled files. For hosts that support long file names, the init file name consists of *program-name* with "-INIT" appended. Thus, the standard file name for a Genera init file is LISPM-INIT; for a Zmail init file, it is ZMAIL-INIT. Hosts that do not support long file names have conventions peculiar to each system.

Following are the names of lispm init source files on some hosts:

Host system	*File name*
LMFS/TOPS-20	LISPM-INIT.LISP
UNIX	lispm-init.l
VMS	LISPMINI.LSP
Multics	lispm-init.lisp
ITS	If user has own directory: LISPM >. If user does not have own directory: *USER* LISPM.

File and Directory Access

Accessing Files

Genera lets you access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Symbolics computer itself, if the machine has its own file system. This section tells you how to get a stream that reads or writes a given file, and what the device-dependent operations on that stream are. Files are named with *pathnames*. Since pathnames are quite complex they have their own chapter. See the section "Naming of Files".

File-Opening Options

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. The file-opening options control whether the stream is for input from an existing file or output to a new file, whether the file is text or binary, and so on.

The following option keywords are recognized. Unless otherwise noted, they are supported generically. Additional keywords can be implemented by particular file system hosts.

:byte-size The possible values are **nil** (the default), a number in the range 1 to 16 inclusive, which is the number of bits per byte, and **:default**, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, **nil** selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, **nil** selects the default byte size of 16 bits. The preferred way to specify the byte-size for files is to use the **:element-type** keyword.

:characters

This option specifies whether the objects contained in the file are characters or fixnums. The preferred way to specify character files is to use the **:element-type** keyword.

Value	*Meaning*
t	Specifies that the file contains character objects. This is the default.
nil	Specifies that the file is a binary file.
:default	On output, **:default** is always **t**, as character files are created by default. On input, **:default** specifies that the file system determine from the file properties for LMFS

files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:deleted	The default is **nil**. If **t** is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.
:direct	The default is **nil**. **t** specifies a direct access stream. See the section "Direct Access File Streams".
:direction	

The **:direction** option allows the following values:

:input	The file is being opened for input. This is the default.
:output	The file is being opened for output.
:block	This is a special case of **:output** that is used for the FEP File System.
:io	The file is being opened for intermixed input and output. Bidirectionality is supported only if the stream is to be a direct stream, that is, **:direct t** is given as well. See the section "Direct Access File Streams".
:probe	A "probe" opening; no data are to be transferred, and the file is being opened to determine whether the file exists, or to gain access to or change its properties. Returns the truename of the object at the end of a link or chain of links. If the value of **:direction** is **:probe** and the value of **:error** is **nil**, then **open** will return the error object instead of **nil**. If the value of **:if-does-not-exist** is **nil**, the error object will still be returned.
:probe-link	The same as **:probe** except that links are not chased. Returns the truename of the object named, even if it is a link.
:probe-directory	The pathname is being opened to find out about the existence of its *directory* component. Otherwise, the semantics are the same as **:probe**. If the directory is not found, a file lookup error is signalled.

nil	This is the same as probe. No data are transferred, and the file is being opened only to gain access to or change its properties. If the value of **:direction** is **nil** and the value of **:error** is **nil**, then **open** will return the error object instead of **nil**. If the value of **:if-does-not-exist** is **nil**, the error object will still be returned.

:element-type

This argument specifies the type of Lisp object transferred by the stream. Anything that can be recognized as being a finite subtype of **character** or **integer** is acceptable. In particular, the following types are recognized:

character	The object being transferred is any character, not just a string-character. The functions **read-char** and/or **write-char** can be used on the stream. This is the default. Note that **file-position** does not work on a stream of characters. You can speed up the reading of the first byte of a long character stream by specifying **zl-user:string-char** as the stream's **:element-type**.
string-char	The object being transferred is a string-character. The functions **read-char** and/or **write-char** can be used on the stream. Note that you can use **file-position**. Note that epsilon coding of fonts is not interpreted in a stream of **zl-user:string-chars**.
(unsigned-byte *n***)**	The object being transferred is an unsigned byte (a non-negative integer) of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
unsigned-byte	The object being transferred is an unsigned byte (a non-negative integer) whose size is determined by the file system. The functions **read-byte** and/or **write-byte** can be used on the stream.
(signed-byte *n***)**	The object being transferred is a signed byte of size *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
signed-byte	The object being transferred is a signed byte whose size is determined by the file

	system. The functions **read-byte** and/or **write-byte** can be used on the stream.
bit	The object being transferred is a bit (values 0 and 1). The functions **read-byte** and/or **write-byte** can be used on the stream.
(mod *n***)**	The object being transferred is a non-negative integer less than *n*. The functions **read-byte** and/or **write-byte** can be used on the stream.
:default	On output, **:default** is always **character**, as character files are created by default. On input, **:default** specifies that the file system determine from the file properties for LMFS files and the canonical type definition for other files what type of objects are stored in the file; then **open** opens it in the appropriate mode.

:error

This option controls what happens when any **fs:file-operation-failure** condition is signalled. **t** is the recommended value for this option. The others have been provided for compatibility with previous systems to aid in converting programs. See the section "File-System Errors".

The option has three possible values:

Value	*Meaning*
t	Signals the error normally. **t** is both the default and the recommended value.
nil	Returns the error object. If the value of either **:if-exists** or **:if-does-not-exist** is **nil**, the error object is still returned.
:reprompt	Reprompts the user for another file name and tries **open** again. When you use this option, remember that the **:pathname** message sent to the stream finds out what file name was really opened. The alternative to **:reprompt** is to use **:error t** and set up a condition handler for **fs:file-operation-failure** that explains the condition and prompts the user.

:estimated-length

The value of the **:estimated-length** option can be **nil** (the default), which means there is no estimated length, or a number of bytes indicating the estimated length of a file to be written. Some file systems use this to optimize disk allocation.

:if-does-not-exist

Specifies the action to be taken if the file does not already exist. The following values are allowed:

:error	Signals an error. This is the default if the **:direction** is **:input**, **:probe**, or any of the **:probe**-like modes, or if the **:if-exists** argument is **:overwrite**, **:truncate**, or **:append**.
:create	Creates an empty file with the specified name, and then proceeds as if it had already existed. This is the default if the **:direction** is **:output** and the **:if-exists** argument is anything but **:overwrite**, **:truncate**, or **:append**.
nil	Does not create a file or even a stream. Instead, simply returns **nil** to indicate failure. This is overridden when the value of **:direction** is either **nil** or **:probe** and the value of **:error** is **nil**. In this case, the error object is returned instead of **nil**.

:if-exists

Specifies the action to be taken if the **:direction** is **:output** and a file of the specified name already exists. If the direction is **:input** or **:probe** (or any of the **:probe**-like directions), this argument is ignored.

The following values are allowed:

:error	Signals an error. This is the default when the version component of the filename is not either **:newest** or **:unspecific**.
:new-version	Creates a new file with the same file name but a larger version number. This is the default when the version component of the filename is either **:newest** or **:unspecific**. File systems without version numbers can choose to implement this by effectively treating it as **:supersede**.
:rename	Renames the existing file to some other name, and then creates a new file with the

specified name. On most file systems, this renaming happens at the time of a successful close.

:rename-and-delete	Renames the existing file to some other name and then deletes it (but does not expunge it, on those systems that distinguish deletion from expunging). Then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.
:overwrite	The existing file is used, and output operations on the stream destructively modify the file. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened.
:truncate	The existing file is used, and output operations on the stream destructively modify the file. The file pointer is initially positioned at the beginning of the file; at that time, the file is truncated to length zero, and disk storage occupied by it is freed.
:append	The existing file is used, and output operations on the stream modify the file. The file pointer is initially positioned at the current end of the file.
:supersede	Supersedes the existing file. If possible, the file system does not destroy the old file until the new stream is closed, against the possibility that the stream will be closed in "abort" mode. This differs from **:new-version** in that **:supersede** creates a new file with the same name as the old one, rather than a file name with a higher version number.
nil	Does not create a file or even a stream. Instead, simply returns **nil** to indicate failure. This is overridden when the value of **:direction** is either **nil** or **:probe** and the value of **:error** is **nil**. In this case, the error object is returned instead of **nil**.

:preserve-dates	The default is **nil**. If **t** is specified, the file's reference and modification dates are not updated.

:raw	The value can be **nil** (the default) or **t**, which disables all character set translation in ASCII files. Note that **:raw** is no longer supported. The preferred way to specify character set translation is to use the **:element-type** keyword.
:submit	This is an option to **open** used to get batch jobs. Currently, this is implemented only for VAX/VMS. When the file you are writing is closed, the file is submitted as a batch job by using this option.
:super-image	The value can be **nil** (the default), or **t** which disables the special treatment of Rubout in ASCII files. Normally Rubout is an escape that causes the following character to be interpreted specially, allowing all characters from 0 through 376 to be stored. This applies to PDP-10 file servers only.
:temporary	The default is **nil**. If **t** is specified, the file is marked as temporary, if the file system has that concept.

Functions for Accessing Files

with-open-file *(stream-variable filename . options...)* &body *body...* *Function*

Evaluates the *body* forms with the variable *stream-variable* bound to a stream that reads or writes the file named by the value of *filename*. The *options* forms evaluate to the file-opening options to be used. See the section "File-Opening Options".

When control leaves the body, either normally or abnormally (via **throw**), the file is closed. If a new output file is being written, and control leaves abnormally, the file is aborted and it is as if it were never written. Because it always closes the file, even when an error exit is taken, **with-open-file** is preferred over **open**. Opening a large number of files and forgetting to close them tends to break some remote file servers, ITS's for example.

filename is the name of the file to be opened; it can be a pathname object, a string, or a symbol. Under Genera, it can be anything acceptable to **fs:parse-pathname**. See the section "Naming of Files". The complete rules for parsing pathnames are explained there.

If an error occurs, such as file not found, the user is asked to supply an alternate pathname, unless this is overridden by *options*. At that point, the user can exit or enter the Debugger, if the error was not due to a misspelled pathname.

Under Genera, if you are opening the file to read it with **zl:read**, and you want to bind the package and so forth, see the special functions for handling file attributes.

```
(with-open-file (mystream "myfile" :direction :input :element-type 'string-char)
  (process-data (read mystream)))
```

See the function **fs:read-attribute-list**. See the function **fs:file-attribute-bindings**.

with-open-file-case *(var pathname . options)* &body *clauses* *Function*

Opens a file, binding the input stream to *var*, using the pathname and options given in the arguments. In the following example, it executes the first clause when the file is not found. When the file is found without error, it executes the second clause, which is the real reason for trying to open the file in the first place. See the section "File-Opening Options".

```
(with-open-file-case (x "f:>dla>foo.lisp" ':direction ':input)
  (fs:file-not-found (send x ':report *error-output*))
  (:no-error (stream-copy-until-eof x *standard-output*)))
```

Any errors other than **file-not-found** (for example, access violations or an unresponsive host) cause an error to be signalled normally.

with-open-file-case-if *cond* (*var pathname . options*) &body *clauses* *Function*

Opens a file, binding the input stream to *var*, using *pathname* and *options* given in the arguments. All clauses are evaluated, but the error handling for the body is performed only if the predicate specified by *cond* returns **t**. See the section "File-Opening Options".

Any errors other than **file-not-found** (for example, access violations or an unresponsive host) cause an error to be signalled normally.

with-open-stream (*stream-variable construction-form*) &body *body* *Function*

Like **with-open-file** except that you specify a form whose value is the stream, rather than arguments to **open**. This is used with nonfile streams. See the function **with-open-file**.

```
(with-open-file (filestream "myfile" :direction :output)
  (with-open-stream (my-stream (misc::get-a-stream))
    ...
))
```

See the section "File-Opening Options".

CLOE Note: This is a macro in CLOE.

with-open-stream-case (*var construction-form*) &body *clauses* *Function*

Opens a stream and binds it to *var*, using *construction-form* to create it. It then executes whichever clause is appropriate, given the condition that resulted from the attempt to create the stream. Refer to the example shown for **with-open-file-case**. See the section "File-Opening Options".

with-open-stream-case-if *cond* (*var construction-form*) &body *clauses* *Function*

Opens a stream and binds it to *var*, using *construction-form* to create it. All clauses are evaluated, but the error handling for the body is performed only if the predicate specified by *cond* returns **t**. See the section "File-Opening Options".

with-standard-io-environment &body *body* *Function*

All output in *body* is printed with ***package***, ***readtable***, and other variables bound to consistent values. This is useful when you wish to write some data that you will retrieve later using the function **read**. This is a custom environment that you create, passing all variables and values that are important before *body*.

with-standard-io-environment inhibits the effect of **#.** while reading. This prevents other forms being read and used as trojan horses. This can be inhibited by rebinding **si:*suppress-read-eval*** to **nil**.

with-input-from-string *(stream string* &key *:index (:start* **0**) *:end)* &body *body*

Function

body is executed as an explicit **progn** with the variable *stream* bound to a character input stream that supplies successive characters from the value of the form *string*. **with-input-from-string** returns the results from the last form of the body.

The input stream is automatically closed on exit from the **with-input-from-string** form, no matter whether the exit is normal or abnormal. The stream should be regarded as having dynamic extent. The following keywords can be used:

keyword	**value**
:index	The form after the **:index** keyword should be a place acceptable to **setf**. If the form is exited normally, then the place will have stored into it the index into *string* indicating the first character not read, or the length of the string if all characters were used. The place is not updated as reading progresses, but only at the end of the operation.
:start	An argument indicating the beginning of a substring of *string* to be used. **:start** defaults to 0.
:end	An argument indicating the end of a substring of *string* to be used. **:end** defaults to the length of the string.

Examples:

```
(values (with-input-from-string
          (stream "A long boring string" :index i)
          (read stream)) i) => A and 2

(values (with-input-from-string
          (stream "A long boring string" :index i :start 2)
          (read stream)) i) => LONG and 7

(values (with-input-from-string
          (stream "A long boring string" :index i :start 9 :end 12)
          (read stream)) i) => RIN and 12
```

```
(let ((index 0)
      (new-str (make-array 10 :element-type 'string-char :fill-pointer 0))
      (my-string "Four score and seven years ago our fore-fathers..."))
  (with-input-from-string (instream my-string :index index)
    (loop
      (dotimes (i 10) (vector-push (read-char instream) new-str))
      (when (string= (subseq new-str 0 4) " our")
        (return t))
      (setf (fill-pointer new-str) 0)))
  new-str)

=> " our fore-"
```

zl:with-input-from-string *(var string* &optional *index limit)* &body *body* *Function*

The form:

```
(zl:with-input-from-string (var string)
    body)
```

evaluates the forms in *body* with the variable *var* bound to a stream that reads characters from the string which is the value of the form *string*. The value of the special form is the value of the last form in its body.

The stream is a function that only works inside the **zl:with-input-from-string** special form, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **zl:with-input-from-string** special forms and use both streams since the special-variable bindings associated with the streams conflict. It is done this way to avoid any allocation of memory.

After *string* you can optionally specify two additional "arguments". The first is *index*:

```
(zl:with-input-from-string (var string index)
    body)
```

uses *index* as the starting index into the string, and sets *index* to the index of the first character not read when **zl:with-input-from-string** returns. If the whole string is read, it is set to the length of the string. Since *index* is updated it cannot be a general expression; it must be a variable or a **setf**able reference. The *index* is not updated in the event of an abnormal exit from the body, such as a **throw**. The value of *index* is not updated until **zl:with-input-from-string** returns, so you cannot use its value within the body to see how far the reading has proceeded.

```
(zl:with-input-from-string (var string index limit)
    body)
```

uses the value of the form *limit*, if the value is not **nil**, in place of the length of the string. If you want to specify a *limit* but not an *index*, write **nil** for *index*. Examples:

```
(setq i 0) => 0
(values (zl:with-input-from-string
          (stream "A long boring string" i)
          (read stream)) i) => A and 2

(values (zl:with-input-from-string
          (stream "A long boring string" i)
          (read stream)) i) => LONG and 7

(values (zl:with-input-from-string
          (stream "A long boring string" i 12)
          (read stream)) i) => BORIN and 12
```

with-output-to-string *(stream* &optional *string* &key *:index)* &body *body*

Function

body is executed as an explicit **progn** with the variable *stream* bound to a character output stream that saves characters in *string*. If *string* is not specified, **with-output-to-string** returns the results from the last form of the body as a string.

If *string* is specified, it must be a string with a fill pointer. The output is incrementally appended to the string, as if using **vector-push-extend** if the string is adjustable, and as if using **vector-push** otherwise. In this case, **with-output-to-string** returns the results from the last form of the body.

The output stream is automatically closed on exit from the **with-output-to-string** form, no matter whether the exit is normal or abnormal. The stream should be regarded as having dynamic extent.

The form after the **:index** keyword should be a place acceptable to **setf**. If the form is exited normally, then the place will have stored into it the index into *string* indicating the first character not read, or the length of the string if all characters were used. The place is not updated as reading progresses, but only at the end of the operation.

Examples:

```
(setq string (make-array 2 :element-type 'string-char
                          :fill-pointer t)) => DD
(values (with-output-to-string (stream nil :index i)
          (write-string "a happy day" stream :start 2 :end 7))
         string i) => "happy" and DD and 17

(values (with-output-to-string (stream string :index i)
          (write-string "a happy day" stream :start 2 :end 7))
         string i) => "a happy day" and DD and 22
```

```
(with-output-to-string (outstream)
  (format outstream "~d + ~d = ~d" 4 5 (+ 4 5)))

=> "4 + 5 = 9"

(setq my-string (make-array 10
                            :element-type 'string-char
                            :fill-pointer 5
                            :initial-element #\.
                            :adjustable t))
=> "....."

(with-output-to-string (outstream my-string)
  (format outstream "~d + ~d = ~d" 4 5 (+ 4 5)))

=> NIL

my-string => ".....4 + 5 = 9"
```

zl:with-output-to-string *(var* &optional *string index)* &body *body*

Function

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream that saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

appends its output to the string that is the value of the form *string*. (This is like the **string-nconc** function). The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have an array-leader; element 0 of the array-leader is used as the fill-pointer. If *string* is too small to contain all the output, **zl:adjust-array-size** is used to make it bigger.

If characters with font information are output, *string* must be of type **sys:art-fat-string**. See the section "**sys:art-fat-string** Array Type".

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or **setf**able reference that contains the index of the next character to be stored into. It must be initialized outside the **with-output-to-string** and is updated upon normal exit. The value of *index* is not updated until **with-output-to-string** returns, so you cannot use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if it does have one it is updated.

The stream is a "downward closure" simulated with special variables, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-output-to-string** special forms and use both streams since the special-variable bindings associated with the streams conflict. It is done this way to avoid any allocation of memory. Examples:

```
(setq string (zl:make-array 2 :type 'zl:art-string :fill-pointer 2)) => DD
(setq i 0) => 0
(values (zl:with-output-to-string (stream nil i)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "happy" and DD and 0

(values (zl:with-output-to-string (stream string i)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "a happy day" and "ha" and 5

(values (zl:with-output-to-string (stream string i)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "a happy day" and "ha" and 10
(values (zl:with-output-to-string (stream string)
          (write-string "a happy day" stream :start 2 :end 7))
        string i) => "a happy day" and "hahappy" and 10
```

sys:with-open-file-search (*stream-variable* (*operation defaults auto-retry*) (*type-list-function pathname* **.** *type-list-args*) **.** *open-options*) *body...* *Function*

Performs a **with-open-file**, searching for a file with one of the types in a list of file types. **zl:load** uses this special form when not given a specific file type to search first for a binary file and then for a source file.

The body is evaluated with *stream-variable* bound to a stream that reads or writes the file. *open-options* are alternating keywords and values to be passed to **open**. See the section "File-Opening Options".

type-list-function should be a function whose first argument is *pathname* and whose remaining arguments are *type-list-args*. The function should return two values: a list of file types to be searched, in order of preference, and a base pathname to be merged with the types and *defaults* in searching for the file. *defaults* can be a pathname or a defaults alist; if omitted, the defaults come from **fs:*default-pathname-defaults***. The special form uses **fs:merge-pathname-defaults** for merging.

If no file is found with any of the types in the list of types, **fs:multiple-file-not-found** is signalled. *operation* is the name of the operation that failed; usually this is the name of the function that contains the **sys:with-open-file-search** form. If *auto-retry* is not **nil** and the condition is not handled, the user is prompted for a new pathname.

open *pathname* &rest *access-path-specific-and-zl-compatible-keywords* &key *(:direction* **:input***) (:element-type* **'character***) :if-exists :if-does-not-exist (:error* **t***)* &allow-other-keys *Function*

Returns a stream that is connected to the specified file. The **open** function only creates streams for *files*; streams for other devices are created by other functions. If an error occurs, such as file not found, the user is asked to supply an alternate pathname, unless this is overridden by *options*.

See the section "File-Opening Options".

When the caller is finished with the stream, it should close the file by using the **:close** operation or the **zl:close** function. The **with-open-file** special form does this automatically, and so is usually preferred. **open** should be used only when the control structure of the program necessitates opening and closing of a file in some way more complex than the simple way provided by **with-open-file**. Any program that uses **open** should set up **unwind-protect** handlers to close its files in the event of an abnormal exit. See the special form **unwind-protect**.

For example:

```
(defun bliss-compile (file)
  (setq file (fs:parse-pathname file))
  (with-open-file (str "comet:usrd$:[mydir]tempfile.com"
                       ':direction ':output
                       ':characters t
                       ':submit t)
    (send str ':line-out
          (format nil "$ BLISS ~A" (send file ':string-for-host)))))
```

Although **open** is a Common Lisp function, the Genera implementation is different from the specification in *Common Lisp*: *the Language* (*CLtL*) in a number of ways:

- *CLtL* defines a fixed set of keywords for **open**: **direction**, **element-type**, **if-exists**, and **if-does-not-exist**. The Genera implementation accepts additional keywords. The CLOE implementation does not.

- *CLtL* says that the default **:element-type** for **open** is **string-char**. In the Genera implementation, the default **:element-type** is **character**. The default in CLOE is **string-char**.

- *CLtL* says that **:element-type** accepts the following types: **string-char**, (**:unsigned-byte** *n*), **unsigned-byte**, (**signed-byte** *n*) **signed-byte**, **character**, **bit**, (**mod** *n*), and **:default**. Specifically:

Element Type	*Status*
string-char	Supported by Genera (but is not the default). CLOE default.

character	Supported by Genera (and is the default). Not supported by CLOE.
unsigned-byte	Not supported by Genera or CLOE.
(**unsigned-byte** 8)	Supported by Genera.
(**unsigned-byte** 16)	Supported by Genera.
(**unsigned-byte** 32)	Supported by Genera for FEP files only.
(unsigned-byte *n*)	Is not supported by Genera or CLOE (except for indicated special cases).
signed-byte	Is not supported by Genera or CLOE.
(signed-byte *n***)**	Is not supported by Genera or CLOE .
bit	Is not supported by Genera.
(mod *n***)**	Is not supported by Genera (due to a bug).
:default	Supported by Genera.

- *CLtL* says that the only valid values of the keyword **:direction** are **:input**, **:output**, **:io**, and **:probe**. The Genera implementation accepts a number of other values for this argument, such as **:in** and **:out**, and device-specific values such as **:block**. The CLOE implementation does not.

The following optional arguments are Symbolics extensions to Common Lisp:

```
&rest access-path-specific-and-zl-compatible-keywords
&allow-other-keys
```

Under CLOE, **open** returns a stream connected to *filename*. The default **:direction** is **:input**, with other possible directions being **:output**, **:io** or **:probe**. The default element type is character; also accepted are other character subtypes, and integer subtypes. The default for both the **:if-exists** and **:if-does-not-exist** parameters is are **:error**. The other possible value for both parameters is **:nil**; thus, **nil** should be returned under the indicated condition. In addition, the value **:create** is accepted for the **:if-does-not-exist** parameter, the values **:new-version**, **:overwrite**, **:append**, **:rename**, **:rename-and-delete**, and **:supersede** are accepted for the **:if-exists** parameter. The **:if-exists** parameter is ignored unless output operations are permitted on the file. Similarly, the **:if-does-not-exist** parameter is ignored unless input operations are permitted on the file.

```
(unwind-protect
  (progn
    (setq myfile (open "myfile" :direction :output :if-exists :rename))
    (format myfile "~A + ~A = ~A~%" 1 2 3))
  (close myfile))
```

In the previous example, **unwind-protect** ensured that the opened file is closed, regardless of the state of the computations involving the file and whatever errors that may occur. It is generally a good idea to wrap any direct calls to **open** in such an **unwind-protect**. An even better idea is to use **with-open-file** where possible.

close *stream* &key *abort* *Function*

stream is closed and no further input or output operations can be performed on it. However, certain inquiry operations can still be performed. It is permissible to close an already closed stream.

If the **:abort** parameter is non-**nil** (the default is **nil**), it indicates an abnormal termination of the use of the stream. Under Genera, attempt is made to clean up any side effects of having created the stream. For example, if the stream performs output to a file that was newly created when the stream was created, then if possible the file is deleted and any previously existing file is not superseded.

The **:abort** keyword argument is ignored by CLOE.

```
(setq file-stream (open "foo" :direction :output))

(format file-stream "hello~%")

(close file-stream)
```

zl:close *stream* &optional *abortp* *Function*

Sends the **:close** message to *stream*.

The *abortp* argument is normally not supplied. If it is **t**, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly created file is deleted, as if it were never opened in the first place. Any previously existing file with the same name remains, undisturbed.

Close File Command

Close File *file-spec keywords*

Closes the specified open files or streams.

file-spec	The pathname of the open file, or the token All. If a pathname is specified, it should be the pathname of an open file. The de-

fault is All. If All is specified, the function **fs:close-all-files** is executed.

keywords	:Mode, :More Processing, :Output Destination, :Query Each
:Mode	{Abort, Normal} The mode in which to perform the close operation. The default is Abort.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Enables you to direct your output. The default is the stream ***standard-output***. Note that redirecting output to a printer can be particularly useful.
:Query Each	{Yes, No} Whether to ask for confirmation before closing each file. The default is Yes.

fs:file-properties *pathname* &optional *(error-p* **t***)* *Function*

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. If *error-p* is **t** (the default) a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned.

fs:change-file-properties *pathname error-p* &rest *properties* *Function*

Some of the properties of a file can be changed, such as its creation date or its author. The properties that can be changed depend on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list**. See the function **fs:directory-list**.

fs:change-file-properties changes one or more properties of a file. *pathname* names the file. The *properties* arguments are alternating keywords and values. If an error occurs and the *error-p* argument is **t**, a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned. If no error occurs, **fs:change-file-properties** returns **t**.

copy-file *from-pathname to-pathname* &key *(:element-type* **':default***)* *(:characters* **':default***)* *:byte-size* *(:copy-creation-date* **t***)* *(:copy-author* **t***)* *:report-stream* *(:create-directories* **':query***)* *Function*

This function copies one file to another. *from-pathname* identifies the source file and must refer to a single file and contain no wild components. *to-pathname* identifies the destination file and can contain wild components, which are eliminated after merging the defaults by means of **:translate-wild-pathname**.

copy-file first attempts to open *from-path*. When that has happened successfully, it parses *to-pathname* and merges it (using **merge-pathnames**) against the *link-opaque truename* of *from-pathname* and a version of **:newest**. The output file specified by *to-pathname* is opened with **:if-exists :supersede**. The processing of *to-pathname* has the following result for version numbers.

Source	*Target*	*Result*
>foo>a.b.newest	>bar>	Retains the version number
>foo>a.b.newest	>bar>x	Makes a new version of >bar>x.b

The defaults for *to-pathname* come from the *link-opaque truename* of *from-pathname*. For systems without links, this is indistinguishable from the truename. Otherwise, the link-opaque truename depends on whether *from-pathname* contains an **:oldest** or **:newest** version. If it does not and if it is fully defaulted, with no wild components, the pathname is its own link-opaque truename. If a pathname *x* contains an **:oldest** or **:newest** version, the link-opaque truename is the pathname of the file or link that corresponds to *x*, with the version number filled in. For example, copying the LMFS file `>a>p1.lisp` to `>b>` results in `>b>p1.lisp`, with the version of `>a>p1.lisp.newest` inherited. This is so whether `>a>p1.lisp.newest` is a real file, a link, or a rename-through link.

By default, **copy-file** copies the creation date and author of the file.

Following is a description of the other options:

:element-type This argument specifies the type of Lisp object transferred by the stream. Anything that can be recognized as being a finite subtype of **character** or **integer** is acceptable. In particular, the following types are recognized:

:characters Possible values:

:default	**copy-file** decides whether this is a binary or character transfer according to the canonical type of *from-pathname*. You do not need to supply this argument for standard file types. For types that are not known canonical types, it opens *from-pathname* in **:default** mode. In that case, the server for the file system containing *from-pathname* makes the character-or-binary decision.
t	Specifies that the transfer must be in character mode.
nil	Specifies that the transfer must be binary mode (in this case, you must supply *byte-size* if using a byte size other than 16).

:byte-size Specifies the byte size with which both files are opened for binary transfers. You must supply **:byte-size** when **:characters** is **nil** and the byte size is other than 16. Otherwise, **copy-file** determines the byte size from the file type for *from-pathname*. When *from-pathname* is a binary file with a known canonical type, it determines the byte size from the **:binary-file-byte-size** property of the type. When the file does not have a known type, it requests the byte size for *from-pathname* from the file server. When the server for the file system containing *from-pathname* cannot supply the byte size, it assumes that the byte size is 16.

:report-stream When **:report-stream** is **nil** (the default), the copying takes place with no messages. Otherwise, the value must be a stream for reporting the start and successful completion of the copying. The completion message contains the truename of *to-pathname*.

:create-directories Determines whether directories should be created, if needed, for the target of the copy. Permissible values are as follows:

t	Try to create the target directory of the copy and all superiors. Report directory creation to ***standard-output***.
nil	Do not try to create directories. If the directory does not exist, handle this condition like any other error.
:query	If the directory does not exist, ask whether or not to create it. This is the default.

zl:copyf *from-path to-path* &key (*characters* **':default**) (*byte-size* **nil**) (*copy-creation-date* **t**) (*copy-author* **t**) (*report-stream* **nil**) (*create-directories* **':query**) *Function*

In your new programs, we recommend that you use the function **copy-file** which is the Symbolics Common Lisp equivalent of the function **zl:copyf**.

Copies one file to another. Copy File (m-X) in the editor uses this function.

from-path and *to-path* are the source and destination pathnames, which can be file specifications. *from-path* must refer to a unique file; it cannot contain any wild components. *to-path* can contain wild components, which are eliminated after merging the defaults by means of **:translate-wild-pathname**. **zl:copyf** first attempts to open *from-path*. When that has happened successfully, it parses *to-path* and merges it (using **fs:merge-pathnames**) against the *link-opaque truename* of *from-path* and version of **:newest**. The output file specified by *to-path* is opened with **:if-exists :supersede**. The processing of *to-path* has the following result for version numbers.

Source	*Target*	*Result*
>foo>a.b.newest	>bar>	Retains the version number
>foo>a.b.newest	>bar>x	Makes a new version of >bar>x.b

The defaults for *to-path* come from the *link-opaque truename* of *from-path*. For systems without links, this is indistinguishable from the truename. Otherwise, the link-opaque truename depends on whether *from-path* contains an **:oldest** or **:newest** version. If it does not and if it is fully defaulted, with no wild components, the pathname is its own link-opaque truename. If a pathname *x* contains an **:oldest** or **:newest** version, the link-opaque truename is the pathname of the file or link that corresponds to *x*, with the version number filled in. For example, copying the LMFS file `>a>p1.lisp` to `>b>` results in `>b>p1.lisp`, with the version of `>a>p1.lisp.newest` inherited. This is so whether `>a>p1.lisp.newest` is a real file, a link, or a rename-through link.

By default, **zl:copyf** copies the creation date and author of the file.

Following is a description of the other options:

:characters

Possible values:

:default	**zl:copyf** decides whether this is a binary or character transfer according to the canonical type of *from-path*. You do not need to supply this argument for standard file types. For types that are not known canonical types, it opens *from-path* in **:default** mode. In that case, the server for the file system containing *from-path* makes the character-or-binary decision.
t	Specifies that the transfer must be in character mode.
nil	Specifies that the transfer must be binary mode (in this case, you must supply *byte-size* if using a byte size other than 16).

:byte-size

Specifies the byte size with which both files are opened for binary transfers. You must supply **:byte-size** when **:characters** is **nil** and the byte size is other than 16. Otherwise, **zl:copyf** determines the byte size from the file type for *from-path*. When *from-path* is a binary file with a known canonical type, it determines the byte size from the **:binary-file-byte-size** property of the type. When the file does not have a known type, it requests the byte size for *from-path* from the file server. When the server for the file system containing *from-path* cannot supply the byte size, it assumes that the byte size is 16.

:report-stream

When **:report-stream** is **nil** (the default), the copying takes place with no messages. Otherwise, the value must be a stream for reporting the start and successful completion of the copying. The completion message contains the truename of *to-path*.

:create-directories	Determines whether directories should be created, if needed, for the target of the copy. Permissible values are as follows:	
	t	Try to create the target directory of the copy and all superiors. Report directory creation to **zl:standard-output**.
	nil	Do not try to create directories. If the directory does not exist, handle this condition like any other error.
	:query	If the directory does not exist, ask whether or not to create it. This is the default.

delete-file *file* *Function*

Deletes the specified file. *file* can be a string, a pathname, or a stream.

delete-file returns an non-**nil** value if successful. An attempt to delete a nonexistent file signals an error. An unsuccessful deletion also signals an error. You cannot specify a **:wild** component.

Under Genera, if *file* is an open stream associated with a LMFS file, the file is deleted immediately, but the stream remains open until closed explicitly.

Under Genera, if *file* is an open stream associated with a file from a non-LMFS file system, then the stream might or might not be closed immediately and the deletion might be immediate or delayed until the stream is explicitly closed, depending on the requirements of the non-LMFS file system.

```
(delete-file "myfile.old")
```

zl:deletef *file* &optional *(error-p* **t***)* *Function*

In your new programs we recommend that you use the function **delete-file** which is the Common Lisp equivalent of the function **zl:deletef**.

Deletes the specified file. *file* can be a pathname or a stream that is open to a file. If *error-p* is **t**, then if an error occurs it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error object is returned; otherwise **t** is returned.

probe-file *file* *Function*

This predicate checks for the existence of a file named *file*. If the file does not exist, it returns **nil**. If the file exists, it returns the truename of the file. This name might be different from *file* because of pathname merging, version numbers, or links. If *file* is an open stream associated with a file, **probe-file** cannot return **nil**, but produces the truename of the associated file. See the function **truename**. For information on the **:probe** value, refer to the discussion of the **:direction** file-opening option, see the section "File-Opening Options".

```
(probe-file "myfile") => #P"/usr/jdoe/myfile.lisp"
```

zl:probef *pathname* *Function*

Returns **nil** if there is no file named *pathname*, or signals an error if anything else goes wrong (such as **sys:host-not-responding**). Otherwise, **zl:probef** returns a pathname that is the truename of the file, which can be different from *pathname* because of file links, version numbers, and so on.

rename-file *file new-name* *Function*

Renames the specified *file* is to *new-name*. *file* can be a string, pathname, or a stream. If *file* is an open stream associated with a file, then both the stream and the file return *new-name* to **truename**.

file can be a pathname, a string, or a stream that is open to a file. The specified file is renamed to *new-name* (a pathname or string). If *error-p* is **t**, when an error occurs it is signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error object is returned; otherwise the three values described below are returned.

file must refer to a unique file; it cannot contain any **:wild** components. *new-name* can contain wild components, which are eliminated after merging the defaults by means of **:translate-wild-pathname**. **rename-file** first attempts to open *file*. When that has happened successfully, it parses *new-name* and merges it (using **fs:merge-pathnames**) against the *link-opaque truename* of *file* and version of **:newest**. This has the following result for version numbers.

Source	*Target*	*Result*
>foo>a.b.newest	>bar>	Retains the version number
>foo>a.b.newest	>bar>x	Makes a new version of >bar>x.b

The defaults for *new-name* come from the link-opaque truename of *file*. For systems without links, this is indistinguishable from the truename. Otherwise, the link-opaque truename depends on whether *file* contains an **:oldest** or **:newest** version. If it does not and if it is fully defaulted, with no wild components, the pathname is its own link-opaque truename. If a pathname x contains an **:oldest** or **:newest** version, the link-opaque truename is the pathname of the file or link that corresponds to x, with the version number filled in. For example, renaming the LMFS file `>a>p1.lisp` to `>b>` results in `>b>p1.lisp`, with the version of `>a>p1.lisp.newest` inherited. This is so whether `>a>p1.lisp.newest` is a real file, a link, or a rename-through link.

rename-file returns three values:

1. The pathname produced by merging and defaulting *new-name*. This is the attempted result of the renaming, produced by performing a **merge-pathnames** operation using *file* for the defaults.

2. The pathname of the object that was actually renamed. This might not be the same as *file*. For example, *file* might have an **:oldest** or **:newest** version, or LMFS rename-through links might be involved. This pathname never has an **:oldest** or **:newest** version.

3. The actual pathname that resulted from the renaming. This might not be the same as *new-name*. For example, *new-name* might have an **:oldest** or **:newest** version, or LMFS create-through links might be involved.

The **:rename** message to streams and pathnames returns the second and third of these values.

Examples:

This example is as simple as possible. Using LMFS, on host `johnny`, with no links involved:

```
(rename-file "johnny:>a>foo.lisp" "bar") =>
#<LMFS-PATHNAME "johnny:>a>bar.lisp">
#<LMFS-PATHNAME "johnny:>a>foo.lisp.17">
#<LMFS-PATHNAME "johnny:>a>bar.lisp.1">
```

This example is as complex as possible. Using LMFS, on host `eddie`, with links

```
>abel>moe.lisp.4 => >baker>larry.lisp (rename-through) (latest)
>baker>larry.lisp.4 =>
  >charlie>sam.lisp.19 (not rename- or create-through) (latest)
>david>jerry.lisp.5 => >earl>ted.lisp (create-through) (latest)

(rename-file "eddie:>abel>moe.lisp.4" "eddie:>david>jerry") =>
#<LMFS-PATHNAME "eddie:>david>jerry.lisp">
#<LMFS-PATHNAME "eddie:>baker>larry.lisp.4">
#<LMFS-PATHNAME "eddie:>earl>ted.lisp.1">
```

An unsuccessful renaming signals an error.

zl:renamef *file new-name* &optional *(error-p* **t***)* *Function*

In your new programs we recommend that you use the function **rename-file** which is the Common Lisp equivalent of the function **zl:renamef**.

Renames one file. The Rename File (`m-X`) command in the editor uses this function.

undelete-file *pathname* &optional (*error-p* **t**) *Function*

Undeletes the specified file. *file* can be a pathname or a stream that 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 object is returned; otherwise **t** is returned. **undelete-file** is like **zl:deletef** except that it undeletes the file instead of deleting it. **undelete-file** is meaningful only for files in file systems that support undeletion, such as TOPS-20 and the Lisp Machine File System.

zl:undeletef *file* &optional *(error-p* **t***)* *Function*

In you new programs, we recommend using function **undelete-file**, which is the Symbolics Common Lisp equivalent of the function **zl:undeletef**.

Undeletes the specified file. *file* can be a pathname or a stream that 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 object is returned; otherwise **t** is returned. **zl:undeletef** is like **zl:deletef** except that it undeletes the file instead of deleting it. **zl:undeletef** is meaningful only for files in file systems that support undeletion, such as TOPS-20 and the Lisp Machine File System.

zl:viewf *file* &optional (*output-stream* **zl:standard-output**) *Function*

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 **zl:viewf** is analogous with **zl:deletef**, **zl:renamef**, and so on. Note: **zl: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).

fs:close-all-files *Function*

Closes all open files. This is useful when a program has run wild opening files and not closing them. It closes all the files in **:abort** mode, which means that files open for output will be deleted. Using this function is dangerous, because you might close files out from under various programs such as Zmacs and Zmail; only use it if you have to and if you feel that you know what you're doing.

fs:*remember-passwords* *Variable*

If not **nil**, causes the first password for each file access path to be remembered. This suppresses prompting for passwords on subsequent attempts by the same user to use that access path. The default value is **nil**.

Note that if you set this variable in an init file, your first login password, typed before the init file is loaded, is not remembered.

Caution: Remembered passwords are accessible. Even after you log out the remembered password for each access path is accessible. If password security is important, leave this variable set to **nil**.

Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as **defun** and **defvar** that define the functions and variables of the program.

Loading a compiled (or BIN) file is similar, except that the file does not contain text but rather predigested expressions created by the compiler that can be loaded more quickly.

These functions are for loading single files. There is a system for keeping track of programs that consist of more than one file: See the section "System Construction Tool".

load *filename* &key *(:verbose* ***load-verbose***) *:print (:if-does-not-exist* **:error**) *:package :default-package (:set-default-pathname* ***load-set-default-pathname***) *Function*

Loads the file specified by *filename* into the Lisp environment. The file can be either a Lisp source file or a binary file. If *filename* specifies the type, it is used; otherwise, **load** looks first for a binary file, then for a Lisp file. If *filename* is a string, it is passed with ***load-pathname-defaults*** as the defaults. See the function **fs:parse-pathname**.

If true, which is the default, the **:verbose** argument causes *load* to print a message indicating what file is being loaded into what package.

:print is not implemented.

:if-does-not-exist specifies the action to be taken if the file does not already exist. The following values are allowed: **:error**, **:create**, **:reprompt**, and **nil**. **:reprompt** reprompts instead of signalling. For information on the other three values, refer to the discussion of **:if-does-not-exist** file-opening option, see the section "File-Opening Options".

:package takes the argument *package*. It binds *package* to ***package***, overriding any package specified in the file attribute list. **:package** is a Symbolics extension to Common Lisp.

:default-package specifies the package to be used if the file's attribute list does not specify a package.

:set-default-pathname controls whether this file's pathname is recorded in ***load-pathname-defaults***. The default value is **t**.

zl:load *pathname* &optional *pkg nonexistent-ok-flag dont-set-default-p no-msg-p* *Function*

Loads the file named by *pathname* into the Lisp environment. The file can be either a Lisp source file or a binary file. If the *pathname* specifies the type, it is used; otherwise, **zl:load** looks first for a binary file, then for a Lisp file. Normally, the file is read into its "home" package, but *pkg* can be supplied to specify the package. *pkg* can be either a package or the name of a package as a string or a symbol. If *pkg* is not specified, **zl:load** prints a message saying what package the file is being loaded into.

nonexistent-ok controls the action of **zl:load** if none of the files is found. If it is **nil** (the default), you are prompted for a new file unless the corresponding condition (**fs:multiple-file-not-found**) is handled. If it is not **nil**, it is the returned value if the file is not found. Other reasons for not finding the file, such as the host being down or the directory not existing, are signalled as different errors. For example, **zl:load** fails when the host is down even when you specified the *nonexistent-ok* argument.

pathname can be anything acceptable to **fs:parse-pathname**. See the section "Naming of Files". *pathname* is defaulted from **fs:load-pathname-defaults**, which is the set of defaults used by **zl:load** and similar functions. See the variable **fs:load-pathname-defaults**. Normally **zl:load** updates the pathname defaults from *pathname*, but if *dont-set-default* is specified this is suppressed.

If an ITS *pathname* contains an FN1 but no FN2, **zl:load** first looks for the file with an FN2 of BIN, then it looks for an FN2 of >. For non-ITS file systems, this generalizes to: if *pathname* specifies a type and/or a version, **zl:load** loads that file. Otherwise it first looks for a binary file, then a Lisp file, in both cases looking for the newest version.

If the value of *no-msg-p* is **t** (it defaults to **nil**), then **zl:load** does not print out the message that it usually prints (that is, the message that tells you that a certain file is being loaded into a certain package).

load-verbose *Variable*

Provides the default value for the **:verbose** argument to **load**. Its initial value is **t**.

```
(load "myfile")
NIL

(let ((*load-verbose* t))
  (load "myfile"))
;;; Loading "/usr/me/myfile.lisp" into package USER
NIL
```

load-pathname-defaults *Variable*

The defaults alist for the **load** function.

load-set-default-pathname *Variable*

Controls whether the **load** and **zl:load** functions change ***load-pathname-defaults*** to reflect the file loaded. The default value is *t*.

zl:readfile *pathname* &optional *pkg no-msg-p* *Function*

zl:readfile is the version of **zl:load** for text files. It reads and evaluates each expression in the file. As with **zl:load**, *pkg* can specify what package to read the file into. Unless *no-msg-p* is **t**, a message is printed indicating what file is being read into what package. The defaulting of *pathname* is the same as in **zl:load**.

File Attribute Lists

Any text file can contain an *attribute list* that specifies several attributes of the file. The functions that load files, the compiler, and the editor look at this attribute list. File attribute lists are especially useful in program source files, that is, a file that is intended to be loaded (or compiled and then loaded).

If the first nonblank line in the file contains the three characters "-*-", some text, and "-*-" again, the text is recognized as the file's attribute list. Each attribute consists of the attribute name, a colon, and the attribute value. If there is more than one attribute they are separated by semicolons. An example of such an attribute list is:

```
; -*- Mode:Lisp; Syntax:Zetalisp; Package:User; Base:10 -*-
```

The semicolon makes this line look like a comment rather than a Lisp expression. This example defines four attributes: mode, syntax, package, and base.

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.

An attribute name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. An attribute value can be such a name, or a decimal number, or several such items separated by commas. Spaces can be used freely to separate tokens. Upper and lowercase letters are not distinguished. There is no quoting convention for special characters such as colons and semicolons. File attribute lists are different from Lisp property lists; attribute lists correspond to the text inside a file, while file properties are characteristics of the file itself, such as the creation date.

The file attribute list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret.

Symbolics Common Lisp has a parser for file attribute lists that creates some Lisp data structure that corresponds to the file attribute list. When a file attribute list is read in and given to the parser (the **fs:read-attribute-list** function), it is converted into Lisp objects as follows: Attribute names are interpreted as Lisp symbols, and interned on the keyword package. Numbers are interpreted as Lisp fixnums, and are read in decimal. If an attribute value contains any commas, then the commas separate several expressions that are formed into a list.

When a file is edited, loaded, or compiled, its file attribute list is read in and the attributes are stored on the attribute list of the generic pathname for that file, where they can be retrieved with the **:get** and **:plist** messages. See the section "Generic Pathnames". So, to examine the attributes of a file, you usually use messages to a pathname object that represents the generic pathname of a file. Note that there other attributes there, too. The function **fs:read-attribute-list** reads the file attribute list of a file and sets up the attributes on the generic pathname; editing, loading, or compiling a file calls this function, but you can call it yourself if you want to examine the attributes of an arbitrary file.

If the attribute list text contains no colons, it is an old EMACS format, containing only the value of the **Mode** attribute.

The following are some of the attribute names allowed and what they mean.

Mode	The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are Lisp and Text.
Package	The name of the package into which the file is to be loaded. See the section "The Need for Packages". For more information about the format and semantics of the Package attribute, see the section "Set Package".
Base	The number base in which the file is written. This affects both **zl:ibase** and **zl:base**, since it is confusing to have different input and output bases. The most common values are 8 and 10. If a file has no Base attribute, the value of the Syntax attribute affects the default of Base. See the Syntax attribute below.
Syntax	The syntax of the programs contained in the file can be either Zetalisp or Common-Lisp. If a file has no Syntax attribute, the value of the Base attribute affects the default of Syntax. • If there is a Base attribute, but no Syntax attribute, the syntax is assumed to be Zetalisp. • If there is a Syntax: Common-Lisp attribute, and no Base attribute, the base is assumed to be 10. • If there is neither a Base nor a Syntax attribute, Base is assumed to be the default base (10) and the syntax is assumed to be Zetalisp. Furthermore, a warning is issued (upon beginning an editing session on the file) to the effect that there is neither a Syntax nor a Base attribute. You should edit your program accordingly.
Lowercase	If the attribute value is not **nil**, the file is written in lowercase letters and the editor does not translate to uppercase. (The editor does not translate to uppercase by default unless the user selects "Electric Shift Lock" mode.)
Fonts	The attribute value is a list of font names, separated by commas. The editor uses this for files that are written in more than one font.
Backspace	If the attribute value is not **nil**, the file can contain backspaces that cause characters to overprint on each other. The default is to disallow overprinting and display backspaces the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.

Patch-File	If the attribute value is not **nil**, the file is a "patch file". When it is loaded, the system does not complain about function redefinitions. Furthermore, the remembered source file names for functions defined in this file are changed to this file, but are left as whatever file the function came from originally. In a patch file, the **defvar** special-form turns into **zl:defconst**; thus patch files always reinitialize variables.

You are free to define additional file attributes of your own. However, you should choose names that are different from all the names above, and from any names likely to be defined by anybody else's programs, to avoid accidental name conflicts.

The function **fs:pathname-attribute-list** is generally the most useful function for obtaining a file's attributes.

fs:pathname-attribute-list *pathname* *Function*

Returns the attribute list for a file designated by *pathname*.

fs:read-attribute-list *pathname stream* &key *:dont-reset-stream* *Function*

Parses file attribute lists from *stream* and updates *pathname* to have that attribute list. The value of this function is the attribute list read from the stream (not the updated attibute list of the pathname).

The *pathname* argument can be a pathname object (*not* a string or namelist, but an actual pathname), or an empty list, or a locative to a property list to be updated. If a pathname is given, it is usually a generic pathname. For more information about generic pathnames: See the section "Generic Pathnames".

stream should be a stream that has been opened and is pointing to the beginning of the file whose file attribute list is to be parsed. The function reads from the stream until it gets the file attribute list, parses it, and puts the corresponding attributes onto the attribute list of *pathname*. The stream is set back to the beginning of the file by using the **:set-pointer** file stream operation unless *:dont-reset-stream* is set to **t**. See the message **:set-pointer**.

The obsolete name of this function is **fs:file-read-property-list**.

Programs in Symbolics Common Lisp generally react to the presence of attributes on a file's file attribute list by examining the attribute list in the generic pathname's property list. However, file attributes can also cause special variables to be bound whenever Lisp expressions are being read from the file—when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base attributes work. You can also deal with attributes this way, by using **fs:file-attribute-bindings**:

fs:file-attribute-bindings *pathname* *Function*

Examines the property list of *pathname* and finds all those property names that have file-attribute bindings. Its obsolete name is **fs:file-property-bindings**.

Each such pathname-property name specifies a set of variables to bind and a set of values to which to bind them. This function returns two values: a list of all the variables, and a list of all the corresponding values. Usually you call this function on a generic pathname whose attribute list has been parsed with **fs:read-attribute-list**. Then you use the two returned values as the first two subforms to a **progv** special form. Inside the body of the **progv** the specified bindings will be in effect.

Usually, *pathname* is a generic pathname. It can also be a locative, in which case it is interpreted to be the property list itself.

Of the standard names, the following ones have file-attribute bindings, with the following effects:

- **zl:package** binds the variable **zl:package** to the package. See the variable **zl:package**.

- **zl:base** binds the variables **zl:base** and **zl:ibase** to the value. See the variable **zl:base**. See the variable **zl:ibase**.

- **fs:patch-file** binds **fs:this-is-a-patch-file** to the value.

Any properties whose names do not have file-attribute bindings are ignored completely.

You can also add your own pathname-property names that affect bindings. If an indicator symbol has a file-attribute binding, the value of that property is a function that is called when a file with a file attribute of that name is going to be read from. The function is given three arguments: the file pathname, the attribute name, and the attribute value. It must return two values: a list of variables to be bound and a list of values to bind them to. Both these lists must be freshly consed (using **list** or **ncons**). The function for the **zl:base** keyword could have been defined by:

```
(defun (:base file-attribute-bindings) (file ignore bse)
  (if (not (and (typep bse 'fixnum)
                (> bse 1)
                (< bse 37.)))
      (ferror nil "File ~A has an illegal -*- Base:~s -*-"
                  file bse))
  (values (list 'base 'ibase) (list bse bse)))
```

Finally, the function **sys:dump-forms-to-file** offers, among other things, the option of manipulating the attribute list of a binary file. See the section "Putting Data in Compiled Code Files".

For example, the following form converts a Lisp file to a binary file, without compiling. The attribute list is obtained from the input stream and cached in the generic pathname. The function **fs:file-attribute-bindings** obtains the list of variables to bind from the generic pathname; these bindings are necessary to ensure that the file is read in the right base, syntax, and package. The **progv** actually accomplishes the binding of the variables.

```
(defun binify-file-internal (input-file output-file)
  (setq input-file (fs:parse-pathname input-file))
  (with-open-file (input input-file :direction :input :characters t)
    (let* ((generic-pathname (send input-file :generic-pathname))
           (attribute-list (fs:read-attribute-list generic-pathname input)))
      (multiple-value-bind (variables-list values-list)
          (fs:file-attribute-bindings generic-pathname)
        (progv variables-list values-list
          (loop with eof-val = (ncons 'eof)
                for form = (read input eof-val)
                while (neq form eof-val)
                collect form into forms
                finally
                  (sys:dump-forms-to-file output-file forms
                                          attribute-list)))))))
```

Accessing Directories

To understand the functions in this section, you need to understand how files are named. See the section "Naming of Files".

Functions for Accessing Directories

fs:directory-list *filename* &rest *options* *Function*

Finds all the files that match *pathname* and returns a freshly consed list with one element for each file and an entry with nil as its car that refers to the file system. Note that this file system entry is usually the first entry. *options* are a list of keywords, with no values, that modify the operation. Each element in the returned list is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file.

The matching is done using both host-independent and host-dependent conventions. Any component of *pathname* that is **:wild** matches anything; all files that match the remaining components of *pathname* are listed regardless of their values for the wild component. In addition, there is host-dependent matching. Typically, this uses the asterisk character (*) as a wild-card character. A pathname component that consists of just a * matches any value of that component (the same as **:wild**). *, appearing in a pathname component that contains other characters, matches any character (on ITS) or any string of characters (on TOPS-20, LMFS, UNIX, and Multics) in the starred positions and requires the specified characters otherwise. Other hosts follow similar but not necessarily identical conventions.

The *options* are keywords that modify the operation. These keywords *do not* take values. The following options are currently defined:

:noerror	If a file-system error (for example, no such directory) occurs during the operation, an error is normally signalled and the user is asked to supply a new pathname. However, if **:noerror** is specified and an error occurs, an error object describing the error is returned as the result of **fs:directory-list**. This is identical to the **:noerror** option to **open**.
:deleted	This is for file servers with soft deletion, such as TOPS-20, LMFS, and FEP. It specifies that deleted (but not yet expunged) files are to be included in the directory listing. Normally, they are not included.
:no-extra-info	This results in only enough information for listing the directory as in Dired.
:sorted	This causes the directory to be sorted so that at least multiple versions of a file are consecutive in increasing version number.

The properties that might appear in the list of property lists returned by **fs:directory-list** are host-dependent to some extent. The following properties are defined for most file servers.

:length-in-bytes	The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file and binary bytes for a binary file).
:byte-size	The number of bits in a byte.
:length-in-blocks	The length of the file in terms of the file system's unit of storage allocation.
:block-size	The number of bits in a block.
:creation-date	The date the file was created, as a universal time. This does not necessarily mean the time that the file itself was created, but rather, the time that the data in it were created. This property corresponds to the concept of "modification date" on many systems. See the section "Dates and Times".
:modification-date	The most recent time at which this file was modified, expressed in Universal Time. This is the same as the creation date if the file has been opened for appending. Operations such as renaming and property changing update this property, but do not update creation date. The dumper, for instance, is driven off this property. See the section "Dates and Times".
:directory	A boolean. If **t**, the object in question is a directory, as opposed to a file or a link. This property can only be returned as **t** in a hierarchical file system.

:auto-expunge-interval	For directories, the time interval between automatic expungings of this directory. If, on a file system that supports this feature (such as TOPS-20 or LMFS), a directory is never automatically expunged, the value of the property will be **nil**. The time interval, when supplied, is expressed as a positive integer, in seconds.
:last-expunge-time	For directories, the date that the directory was last expunged. It is **nil** if the directory has never been expunged.
:reference-date	The most recent date that the file was used, as a universal time.
:author	The name of the person who created the data in the file, as a string.
:account	A string. Highly system-dependent in format.
:deleted	A boolean. **t** for a "deleted" file, in file systems supporting "soft deletion".
:dont-delete	A boolean. If it is **t**, an error results if an attempt is made to delete the file.
:dont-dump	A boolean. Suppresses backup dumping.
:dont-reap	A boolean. A flag used by directory maintenance tools.
:dumped	A boolean. **t** if and only if the file has been dumped to backup tape.
:generation-retention-count	A number that specifies how many versions of a file should be saved.
:link-to	A string. This is the target pathname of a link, as a string.
:offline	A boolean. **t** if the file has been moved to archival storage.
:physical-volume	A string. The volume on which the file is mounted.
:protection	A string. What protections have been set for the file.
:reader	A string. The last person to have read the file.
:settable-properties	A list of the properties that may be changed for the file using **fs:change-file-properties**.
:temporary	A boolean. **t** if the file is temporary.

Compare Directories Command

Compare Directories *pathname1 pathname2 keywords*

Compares the two specified directories. This command compares only filenames, not the contents of the files in the directories. If the directories contain the same information, you are notified that there are no differences in the two directories. If there are differences, two lists are printed. The first list contains the names of all the files in the first directory that are not in the second directory. The second list contains the names of all the files in the second directory that are not in the first directory.

pathname1	The pathname of the first directory to be used in the comparison. The default is the usual pathname default.
pathname2	The pathname of the second directory to be used in the comparison. The default is the usual pathname default.
keywords	:Ignore Versions, :More Processing, :Output Destination
:Ignore Versions	{Yes or No} The default is No. If Yes, then consider files with the same name and type to be the same even if they have different version numbers.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "`FUNCTION M`").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

fs:multiple-file-plists *filenames* &rest *options* *Function*

Returns the property list for each file in *filenames*. For example:

```
(fs:multiple-file-plists
  (list "sys: doc; str; str1.sar" "sys: sys2; table.lisp")) =>

((#P"SYS:SYS2;TABLE.LISP.NEWEST" :TRUENAME
#P"Q:>rel-7>sys>sys2>table.lisp.43" :LENGTH 97047 :AUTHOR "Moon"
:BYTE-SIZE NIL :CREATION-DATE 2729141698)
(#P"SYS:DOC;STR;STR1.SAR.NEWEST" :TRUENAME
#P"Q:>rel-7>sys>doc>str>str1.sar.45" :LENGTH 15625 :AUTHOR "nancy"
:BYTE-SIZE NIL :CREATION-DATE 2728753545))
```

The properties that might appear in the list of property lists returned by **fs:multiple-file-plists** are host-dependent to some extent. The following properties are defined for most file servers:

:truename	Returns the pathname of the file actually open on this stream. This can be different from what **:pathname** returns because of file links, logical devices, mapping of "newest" version to a particular version number, and so on.
:length	The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file and binary bytes for a binary file).
:author	The name of the person who created the data in the file, as a string.
:byte-size	The number of bits in a byte.
:creation-date	The date the file was created, as a universal time. This does not necessarily mean the time that the file itself was created, but rather, the time that the data in it were created. This property corresponds to the concept of "modification date" on many systems. See the section "Dates and Times".

Note that &rest *options* are passed along to the function doing the work.

fs:change-file-properties *pathname error-p* &rest *properties* *Function*

Some of the properties of a file can be changed, such as its creation date or its author. The properties that can be changed depend on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list**. See the function **fs:directory-list**.

fs:change-file-properties changes one or more properties of a file. *pathname* names the file. The *properties* arguments are alternating keywords and values. If an error occurs and the *error-p* argument is **t**, a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned. If no error occurs, **fs:change-file-properties** returns **t**.

fs:file-properties *pathname* &optional *(error-p* **t***)* *Function*

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. If *error-p* is **t** (the default) a Lisp error is signalled. If *error-p* is **nil** and an error occurs, the error object is returned.

fs:complete-pathname *defaults string type version* &rest *options* *Function*

string is a partially specified file name. (Presumably it was typed in by a user and terminated with the COMPLETE or END to request completion.) **fs:complete-pathname** looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded in a host-dependent fashion.

string is completed relative to a default pathname constructed from *defaults*, the host (if any) specified by *string*, *type*, and *version*, using the function **fs:default-pathname**. See the function **fs:default-pathname**. If *string* does not contain a colon, the host comes from *defaults*; otherwise the host name precedes the first colon in *string*.

options are keywords (without following values) that control how the completion will be performed. The following option keywords are allowed. Their meanings are explained more fully below.

:deleted	Look for files that have been deleted but not yet expunged. The default is to ignore such files.
:read or **:in**	The file is going to be read. This is the default. The name **:in** is obsolete and should not be used in new programs.
:write or **:print** or **:out**	The file is going to be written (that is, a new version is going to be created). The names **:print** and **:out** are obsolete and should not be used in new programs.
:old	Look only for files that already exist. This is the default. **:old** is not meaningful when **:write** is specified.
:new-ok	Allow either a file that already exists, or a file that does not yet exist. **:new-ok** is not meaningful when **:write** is specified. The **:new-ok** option is no longer used by any system software, because users found its effects (in the Zmacs command Find File (`c-X c-F`)) to be too confusing. It remains available, but programmers should consider this experience when deciding whether to use it.

The first value returned is always a string containing a file name; either the original string, or a new, more specific string. The second value returned indicates the status of the completion. It is non-**nil** if it was completely successful. The following values are possible:

:old The string completed to the name of a file that exists.

:new The string completed to the name of a file that could be created.

nil The operation failed for one of the following reasons:

- The file is on a file system that does not support completion. The original string is returned unchanged.
- There is no possible completion. The original string is returned unchanged.
- There is more than one possible completion. The string is completed up to the first point of ambiguity.

- A directory name was completed. Completion was not successful because additional components to the right of this directory remain to be specified. The string is completed through the directory name and the delimiter that follows it.

Although completion is a host-dependent operation, the following guidelines are generally followed:

When a pathname component is left completely unspecified by *string*, it is generally taken from the default pathname. However, the name and type are defaulted in a special way described below and the version is not defaulted at all; it remains unspecified.

When a pathname component is specified by *string*, it can be recognized as an abbreviation and completed by replacing it with the expansion of the abbreviation. This usually occurs only in the rightmost specified component of *string*. All files that exist in a certain portion of the file system and match this component are considered. The portion of the file system is determined by the specified, defaulted, or completed components to the left of this component. A file's component *x* matches a specified component *y* if *x* consists of the characters in *y* followed by zero or more additional characters; in other words, *y* is a left substring of *x*. If no matching files are found, completion fails. If all matching files have the same component *x*, it is the completion. If there is more than one possible completion, that is, more than one distinct value of *x*, there is an ambiguity and completion fails unless one of the possible values of *x* is equal to *y*.

If completion of a component succeeds, the system attempts to complete any additional components to the right. If completion of a component fails, additional components to the right are not completed.

A blank component is generally treated the same as a missing component; for example, if the host is a LMFS, completion of the strings "foo" and "foo." deals with the type component in the same way. The strings are not completed identically; completion of "foo" attempts to complete the name component, but completion of "foo." leaves the name component alone since it is not the rightmost.

If *string* does not specify a name, then the name of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, completion fails.

- With the **:write** option, the default name is always used when *string* does not specify a name, regardless of what files exist.

- With the **:new-ok** option, if any files with the default name exist in the specified, defaulted, or completed directory, the default name is used. If no such files exist, but all files in the directory have the same name, that name is used instead. Otherwise, the default name is used.

The special treatment of the case where all files in the directory have the same name is not very useful and is not implemented by all file systems.

If *string* does not specify a type, then the type of the default pathname is *preferred* but is not necessarily used. The exact meaning of this depends on *options*:

- With the default options, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, completion fails.

- With the **:write** option, the default type is always used when *string* does not specify a type, regardless of what files exist.

- With the **:new-ok** option, if a file with the specified, defaulted, or completed name and the default type exists, the default type is used. If no such file exists, but one or more files with that name and some other type do exist and all such files have the same type, that type is used instead. Otherwise, the default type is used.

In file systems such as LMFS and UNIX that require a trailing delimiter (> or /) to distinguish a directory component from a name component, the system heuristically decides whether the rightmost component was meant to be a directory or a name, and inserts the directory delimiter if necessary.

If *string* contains a relative directory specification for a host with a hierarchical file system, it is assumed to be relative to the directory in the default pathname and is expanded into an absolute directory specification.

The host and device components generally are not completed; they must be fully specified if they are specified at all. This might change in the future.

If *string* does not specify a version, the returned string does not specify a version either. This differs from file name completion in TOPS-20; TOPS-20 completes an implied version of "newest" to a specific number. This is possible in TOPS-20 because completing a file name also attaches a "handle" to a file. In Genera, the version number of the newest file might change between the time the file name is completed and the time the actual file operation (open, rename, or delete) is performed.

A pathname component must satisfy the following rules in order to appear in a successful completion:

- The host, device, and directory must actually exist.

- The name must be the name of an existing file in the specified directory, unless **:write** or **:new-ok** is included in *options*.

- The type must be the type of an existing file with the specified name in the specified directory, unless **:write** or **:new-ok** is included in *options*.

- A pathname component always completes successfully if it is **:wild**.

When the rules are not satisfied by a component taken from the default pathname, completion fails and that component remains unspecified in the resulting string. When the rules are not satisfied by a component taken from *string*, completion fails and that part of *string* remains unchanged (other components of *string* can still be expanded).

zl:listf *path* &optional *(output-stream standard-output)* *Function*

Display an abbreviated directory listing. The default for name, type, and version of *path* is **:wild**.

```
(listf "f:>maria>mit-220")
```

The format of the listing varies with the operating system.

Conversion Tools

About the Conversion Tools

The Conversion Tools are a series of special-purpose Zmacs commands that save you time and effort in editing large pieces of code in ways that can be done semi-automatically. This is particularly useful when converting from one Lisp dialect to another.

The available Conversion Tools are:

- Flavors to CLOS, for converting object-oriented programs from using Flavors to using the Common Lisp Object System (CLOS). See the section "Flavors to CLOS Conversion".

- Zetalisp to Common Lisp, for converting Zetalisp programs to Symbolics Common Lisp. See the section "Zetalisp to Common Lisp Conversion".

- Symbolics Common Lisp to portable Common Lisp, for assistance in converting Symbolics Common Lisp programs to more portable programs that will run in a variety of Common Lisp implementations, including Genera and Cloe as well as other vendors' Common Lisps. See the section "Symbolics Common Lisp to Portable Common Lisp Conversion".

- Package Conversion, for moving a program to a different package. See the section "Package Conversion".

- Tools you write yourself, to do any source-to-source conversions you may require. See the section "Creating Your Own Conversion Set".

Though the Conversion Tools are simple, the conversion task is not. For this reason you must use the Conversion Tools with knowledge and care. For example, when converting Zetalisp to Common Lisp, you should be sufficiently comfortable with both Zetalisp and Common Lisp to be aware of function equivalences and to understand the possible effects of a conversion (such as a changed order of argument evaluation for functions whose calling sequence is different in Common Lisp).

It is also very desirable that you be familiar with the details and the intent of the code you are converting. As a trivial example, in order to get the correct conversion of a Zetalisp division operation, you must know what numeric data types the operation is intended for, whether or not truncated division is needed, and so on; this knowledge will let you select quickly among the three Common Lisp alternatives offered you by the Conversion Tools.

Obviously, you also need familiarity with the basic workings of the Zmacs editor.

Bear in mind that the conversion commands are intended as an *aid* to conversion, not as a fully automatic conversion tool. Used properly, they will save you time and effort, but you must monitor the results carefully after each step and be aware that you might have to do some manual work after conversion: for instance, comments might not end up exactly in the right place in the rearranged program, indentation might change, converted functions might need some additions to the code, and so on.

Getting Started with Conversion

Loading the Conversion Tools

The Conversion Tools reside on the source tape you received with Genera. Once you have restored the distribution tape as described in your *Software Installation Guide*, type this from the Lisp Listener:

```
Load System Conversion-Tools
```

When the system files are loaded, you are ready to use the Conversion Tools.

For your convenience, the distribution tape also includes two sample programs you can use to try out the Conversion Tools: SYS:CONVERSION-TOOLS;CONVERSION-TEST-PROGRAM.LISP (the main example), and SYS:CONVERSION-TOOLS;CONVERSION-OCTAL-TEST-PROGRAM.LISP (for radix conversion).

Preparing Your Files

All phases of the Conversion Tools run over files that have been read into the Zmacs editor. If you have a large number of files you can save yourself some work by treating them as a unit and having them read in automatically; you can do this in one of two ways: for system files, that is, files that you have grouped via **defsystem**, execute `m-X` Select System As Tag Table before starting. For other collections of files, select these files as a Tag Table by executing `m-X` Select Some Files As Tag Table. All files in the Tag Table are read into editor buffers when you issue the first conversion command. For more on working with Tag Tables, see the section "Tag Tables and Search Domains in Zmacs".

As a precaution, especially if your file server does not maintain multiple file versions, we suggest you copy your source files to another directory. You can then run the Conversion Tools over the original files in their home directory.

Running the Conversion Routines

All the conversion commands use the Zmacs extended command syntax and are, therefore, prefixed by m-X. Command completion is provided as usual. Generally the commands can be applied to Region, Buffer, or Tag Table. To convert the base from octal, for example, you could type any one of these three commands:

```
m-X Convert Base of Region
m-X Convert Base of Buffer
m-X Convert Base of Tag Table
```

For commands affecting a region, you would, of course, mark the region before issuing the conversion command. Remember that only the region you select is converted while the rest of your program is left unchanged. If you don't mark a region, it defaults to the definition containing the cursor.

The Tag Table version is the most powerful. It applies the conversion to all of the files specified by the current Tag Table. Use one of these commands first to select a current Tag Table:

```
m-X Select All Buffers As Tag Table
m-X Select Some Files As Tag Table
m-X Select System As Tag Table
m-X Select Some Buffers As Tag Table
```

Certain conversion commands apply only to buffers meeting certain crtieria, such as buffers that are in a certain package. Whenever you use one of these commands with a Tag Table, the command finds and display a list of all the candidate buffers and asks you how to proceed: Y(es) operates on all files, N(o) on none, S(elective) on only some of these files (specified by you). As each file is converted, the system displays its pathname. The message "Done." appears after all files have been converted.

Interaction with the Conversion Tools is flexible; when the system queries you on specific changes, it always offers you the option of editing your program manually and then resuming the current conversion. (You do this by typing c-R).

If an unwanted conversion occurs, you can undo it with the Zmacs undo facility (press c-sh-U, or mark a region and press m-sh-U). For more information, see the section "Zwei Undo Facility".

As already stated, each conversion phase should produce a program that is fully compatible with the source program. It is a good idea to test your program after each conversion phase and to save your buffers after you are satisfied.

Since there is no reliable convention for structured comments, the tools ignore them, just as the Lisp compiler would (this is not always how the editor acts). If you have lots of commented-out code that you want to convert too, you can uncomment it before converting, and recomment it when done. For instance, if you use the `#||` **...** `||#` convention, you can cause the comment delimiters to be ignored by using `m-X Multiple Query Replace` to replace `#||` with `#|`**{begin}**`|#` and `||#` with `#|`**{end}**`|#`, which are unlikely to occur already in your program. When the conversion is complete, you can replace the comment delimiters as easily as you removed them.

When the Conversion Tools encounter an error they usually proceed to the next definition and restart from there, after printing the error message. You should go back and check the place where the error occurred to see what happened.

Getting Help

As mentioned, command completion is available for the Conversion Tools in the usual fashion. See the section "Zmacs Command Completion".

Additionally, there is a *Help facility* that you invoke by pressing the HELP key; the following general kinds of help are available:

- Finding commands: if you press HELP after typing part of a command, the system displays a list of commands that can be used to complete it. For instance, if you invoke HELP after typing Convert at the minibuffer the following displays:

```
You are typing at a mini-buffer that acts like an input editor.
You are being asked to enter an extended command.

These are the possible extended commands starting with "conv":
  Convert Base Of Buffer          Convert Lisp Syntax Of Buffer
  Convert Base Of Region          Convert Lisp Syntax Of Region
  Convert Base Of Tag Table       Convert Lisp Syntax Of Tag Table
  Convert Functions Of Buffer     Convert Package Of Buffer
  Convert Functions Of Region     Convert Package Of Region
  Convert Functions Of Tag Table  Convert Package Of Tag Table
```

- Finding buffers: when the system verifies the name of the buffer to be converted and you don't want the default (current) buffer, HELP offers to display all your editor buffers; select the buffer you want from this display either by clicking on it with the mouse, or by typing the buffer name into the minibuffer.

- Options menu: when the system queries you for specific object conversions, HELP gets you a menu of possible responses. The convention is:

Action	*Character or Key*
Do it	Y, SPACE
Skip it	N, RUBOUT

Do it every time	P
Manual Edit	c-R
Do it, then Allow Edit	, (comma)
Redisplay screen	c-L, REFRESH
Do it, leave a comment	; (semicolon)

If multiple conversions are offered, you can also press the number of the conversion (1 for the first, 2 for the second, etc.) to do it. If you press comma it does the first conversion then allows editing. Period does the second conversion and then allows editing. Slash does the third conversion and then allows editing. If there are more than three possibilities, only the first three can be done with an edit.

If you edit your program manually during conversion, press END to signal completion of your edit and return to the current conversion step.

The HELP and SCROLL keys are active during this query, along with the standard scrolling commands such as c-V and s-R.

Flavors to CLOS Conversion

Commands:

```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

To translate a Flavors program to CLOS, issue one of the above commands. When prompted "Conversion to use", select Flavors to CLOS. This semiautomatic conversion will do most of the work required to convert a program from Flavors to CLOS. Additional manual effort will be required if the program uses Flavors features that do not have any direct counterpart in CLOS.

Each form in the program that can be converted to CLOS results in an interactive query showing the old form in context in the editor buffer and one or more suggested replacement functions. You can then enter one of a variety of single-character commands. Press HELP for a list of options. For further information, see the section "Getting Help with Conversion".

This is not a conversion from Symbolics Common Lisp to portable Common Lisp. The Flavors to CLOS conversion converts only Flavors functions and macros; it does not touch the rest of the program.

If your Flavors program is in Zetalisp, it is recommended, but not required, that you first convert it to Common Lisp before converting it to CLOS. See the section "Zetalisp to Common Lisp Conversion".

The general idea is to convert a flavor to a class with the same name, to convert messages to generic functions, and to convert Flavors generic function and method definitions to CLOS generic function and method definitions with the same generic function name. Flavors names and syntax are replaced with CLOS names and syntax wherever a correspondence exists. Flavors features that do not exist in CLOS are left in the program for you to convert by hand.

CLOS uses symbols that are not accessible in packages such as **cl-user** and **scl**. Unless you first move your Flavors program into a package that uses **clos** or **future-common-lisp**, the CLOS symbols will be inserted into your program with package prefixes. Thus **defmethod** will be converted to **clos:defmethod**. If you later move your program to a package or an implementation where CLOS is directly accessible, you can use m-X Query Replace to remove the package prefixes. For information about moving your Flavors program to another package, see the section "Package Conversion".

The conversion tool extracts information about your program, such as what method-combination type a generic function uses or what instance variables and **defun-in-flavor** functions are defined for a flavor, by looking in three places:

1. Forms that have already been converted during the same conversion operation.

2. Forms that have been read into editor buffers.

3. Flavor and generic function definitions that have been loaded into the world.

Consequently, you will get better results if the entire program is read into the editor or loaded into the world before converting any of it.

The new version of the program can't be used at the same time as the old version, because it uses the same names with different meanings (a class is different from a flavor, and a CLOS generic function is different from a Flavors generic function). However, if you move the new version into a new package before converting it you can avoid this problem. For information about moving your Flavors program to another package, see the section "Package Conversion".

CLOS does not have any equivalent of Flavors' functions whose scope is limited to methods for a particular flavor. Therefore **defun-in-flavor** is converted to **clos:defmethod** (with the addition of an argument, since **self** is no longer conveyed automatically) and **defmacro-in-flavor** is converted to ordinary **defmacro**. These conversions can result in name conflicts, if the same name had been used in the scope of two different flavors. These conflicts will probably show up as method lambda-list congruency errors and must be resolved manually.

Options for **defflavor** or **defgeneric** that do not have counterparts in CLOS are converted into an **:unconverted-flavor-options** or **:unconverted-defgeneric-options** option to remind you to convert these options manually. CLOS does not accept **:unconverted-flavor-options** or **:unconverted-defgeneric-options**, so if you compile the program without completing the manual conversion, an error will be reported.

Some **defflavor** options, such as **:required-flavors** or **:abstract-flavor**, can simply be deleted without harming the operation of a working program. Other options, like **:mixture** or **:special-instance-variables**, have no simple conversion to CLOS. If you use them you might have to restructure your program. The following options have fairly straightforward conversions that are a little too complex to be done automatically:

:constructor	Use **defun** to define a function that calls **clos:make-instance**.
:default-handler	Use **clos:defmethod** to define a default method for each generic function that needs default handling.
:init-keywords	Use **&key** in a **clos:initialize-instance** method.

Symbolics CLOS has an extension to allow slots to be located with **locf**. If you use **:locatable-instance-variables** in Flavors, it is converted to the Symbolics CLOS **:locator** slot-option. This will not work in other CLOS implementations.

Some Flavors efficiency features with no counterpart in CLOS are simply discarded. For example, **defwhopper-subst** is simply converted into an **:around** method, and **defsubst-in-flavor** is treated the same as **defun-in-flavor**.

Messages are converted to generic functions. This applies both to receivers (**defmethod**, **:gettable-instance-variables**, and so on.) and to senders (**send**, **lexpr-send**). Messages sent with **funcall** or **apply** are not recognized as messages and are not converted. The first time a particular message is encountered during a conversion operation, you will be prompted for the name of the replacement generic function. You can press RETURN to accept the offered default. You can press SPACE to complete to "None", which means to leave this message unconverted. You can later convert the message by hand, or leave it as a message if your program runs partially in CLOS and partially in Flavors. A generic function name that is already in use as a special form, macro, or nongeneric function will not be accepted.

The following Flavors constructs are not automatically converted. They are left in the program and must be converted by hand. Some of these do not have any simple conversion to CLOS; if you use them, you might have to restructure your program. Others are used infrequently enough that automatic conversion did not seem worthwhile.

compile-flavor-methods
define-method-combination
defwrapper
:fasd-form
get-handler-for
lexpr-send-if-handles
operation-handled-p
recompile-flavor
send-if-handles
:unclaimed-message
:which-operations

In addition, none of the documented Flavors constructs in the **flavor**, **system**, and **zl** packages is converted. These are generally too internal to have exact correspondences in CLOS.

Mixed use of Flavors and CLOS is not supported at present. That is, a class cannot inherit from a flavor, a flavor cannot inherit from a class, CLOS generic functions cannot be used with Flavors methods, and Flavors generic functions cannot be used with CLOS methods nor with CLOS instances. The only supported mixed use is that CLOS generic functions can be used with Flavors instances and CLOS methods can be specialized to a flavor as if it was a class. Therefore if you convert a program to CLOS, you must convert all uses of a given flavor to use a class instead, and all Flavors methods for a given generic function to be CLOS methods instead.

Symbolics Common Lisp to Portable Common Lisp Conversion

The Symbolics Common Lisp to portable Common Lisp conversion tool assists in converting Symbolics Common Lisp programs to more portable programs that will run in a variety of Common Lisp implementations, including Genera and Cloe as well as other vendors' Common Lisps.

In general, the strategy is to convert to "CLtL" Common Lisp (as defined by the book *Common Lisp: the Language*, first edition, by Guy L. Steele, Jr.) when that is possible and otherwise leave things unconverted. However, if the program is in a package from the Common-Lisp, CLtL, or CLtL-Only universe, the resulting prefix is **future-common-lisp:**.

A possible alternative strategy, which we did not adopt, would have been to take advantage of extensions present in particular Common Lisp implementations such as Symbolics Cloe, creating a conversion tool targeted to one particular Common Lisp implementation rather than to the common subset of most implementations. You can convert the remaining portions of your program to use extensions provided by particular implementations, if you so choose, after performing the automatic conversions to the common subset; perhaps inserting **#+/#-** conditionalization.

The program output by the Symbolics Common Lisp to portable Common Lisp conversion tool should be run in the Common Lisp Developer to help verify the correctness of the conversion, before porting it to another implementation.

See the section "Developing Portable Common Lisp Programs".

The Symbolics Common Lisp to portable Common Lisp conversion tool converts a subset of Symbolics Common Lisp constructs to portable Common Lisp. Constructs that can be locally converted into portable constructs are converted. Constructs that have no portable equivalent, or that would require nonlocal changes to the program, are not converted. Examples of this include locatives and array leaders.

Constructs such as **who-calls** that are only intended to be called interactively, not incorporated into programs, are not converted.

Certain large facilities such as Flavors and Dynamic Windows are not converted by this tool (to CLOS and CLIM, respectively), because there are separate tools just for them.

Several facilities from the future X3J13 Common Lisp are widely available now and thus are assumed to be present in the target implementation. These include **loop**, **defpackage**, **destructuring-bind**, the Condition system, and the **dynamic-extent** declaration.

A number of Symbolics Common Lisp facilities such as resources, initializations, SCT, and many others have no counterpart in X3J13 Common Lisp, so no conversion is attempted. You should convert uses of these facilities by hand or obtain an implementation of the facility in each target environment of interest.

Some extensions to standard constructs, such as the **:area** and **:displaced-conformally** arguments to **make-array**, are discarded during the conversion; this might not produce the desired behavior. In general the program output by the conversion tool will require some testing and additional manual changes before the conversion process is complete. Compiling the program in the Common Lisp Developer and checking the compiler warnings is the first step in this process. The second step is to run the Symbolics Common Lisp version (in regular Genera) and the converted version (in the Common Lisp Developer) simultaneously and compare their behavior for a set of test cases.

To run the Symbolics Common Lisp to portable Common Lisp conversion tool, issue one of the following commands and specify "Symbolics Common Lisp to portable Common Lisp" in response to the "Conversion to use" query. Completion and help are available when entering the name of a conversion, so you do not need to type the entire name.

Commands:

```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

When converting to the Common Lisp Developer (the CLtL or CLtL-Only package universe), superfluous package prefixes such as **cl:** will frequently be left in the program. This occurs because the Common Lisp Developer uses alternative versions of many Common Lisp symbols that disable Symbolics extensions or perform extra error checking. These package prefixes should be removed using one of the commands listed above, specifying "Common Lisp to Common Lisp Developer" in response to the "Conversion to use" query. This removes the prefixes only from symbols that are essentially equivalent; using `m-X` Replace String would be dangerous as it might also remove package prefixes that indicate constructs that have not yet been converted.

Package Conversion

Commands:

```
m-X Convert Package of Region
m-X Convert Package of Buffer
m-X Convert Package of Tag Table
```

The Package Conversion tool modifies a program so that it can be read in a different package but still get the same symbols. This tool is typically used as one step in converting a program from one Lisp dialect to another. Once the program has been moved to a package that inherits the symbols of the target dialect, all symbols inherited from the original dialect are tagged with package prefixes and can easily be found so that they can be converted to constructs of the new dialect.

For example, one step of converting from Zetalisp to Common Lisp is to move the program from a package that uses Zetalisp to a package that uses Common Lisp. At each place where a symbol inherited from Zetalisp is referenced, if the same symbol does not occur in Common Lisp a **zl:** package prefix is inserted in front of the symbol. This affects inherited symbols only. Symbols that are directly present in the old package are replaced by symbols with the same names directly present in the new package.

No symbol substitution is done; the file is just changed to read the same global symbols into the new package. For instance, **memq** is converted to read **zl:memq**, not to the corresponding Common Lisp function name, **member** (that translation occurs later).

In some cases there can be name conflicts between local symbols of the program and defined symbols of the target dialect. This occurs most often when converting from Zetalisp to Common Lisp. If name conflicts are possible, you should check for them before converting the package. See the section "Step Three: Name Conflict Resolution".

Procedure for Package Conversion

The Conversion Tools ask you for a package to replace each of the packages in the program. When converting a region or a buffer, this is just one package, the buffer's package. When converting a Tag Table, this is the set of packages of all the files in the Tag Table. You have three choices:

- Choose a package that is already defined.
- Create a new package. This is the default.
- Don't convert this package, just keep on using it.

If you convert a package more than once in the same session, after the first time the default is to do the same conversion that you did last time, but all three choices are still available.

When you choose to create a new package, you must specify its name and the package it uses (inherits from). This is normally done by specifying a package universe (such as Zetalisp, Common-Lisp, or CLtL-Only) and letting the name and the used package default according to the selected package universe. For example, when converting from Zetalisp to Common-Lisp, the original package might be named **foo** and inherit from **global**. If you specify the Common-Lisp package universe for the new package, it would be named **common-lisp-foo** and would inherit from **symbolics-common-lisp** or **common-lisp** (your choice).

If you have complex package declarations, you might prefer to create a new package with a different name by editing your **defpackage** form, before doing the package conversion. Then specify that new package when the Conversion Tools ask what your old package becomes.

If you don't care about keeping your old package name, you can simply convert to a new package name and you're done. If instead you want to keep your old package name and redefine it as a new package, you must follow a more complex procedure so that the old and new packages can coexist in the same world temporarily during the conversion process. The procedure is as follows:

1. Suppose you are converting from Zetalisp to Symbolics Common Lisp and you have an old package, called **my-package**, that uses Zetalisp. During conversion you replace this with a new package called **common-lisp-my-package**, which uses Symbolics Common Lisp. This new package name will only be used temporarily during the conversion process. The Conversion Tools convert all your symbols from **my-package** to **common-lisp-my-package**.

2. If the Conversion Tools changed the file attribute lines at the front of your files to refer to **common-lisp-my-package**, change them back to **my-package**.

3. Now save your files.

4. Edit the **:use** option of the **defpackage** form for **my-package** to specify **symbolics-common-lisp**. Save the file containing this definition.

5. Reboot into a fresh world, without loading your system. Evaluate your **defpackage** form to create a new version of **my-package** that uses Symbolics Common Lisp instead of Zetalisp.

6. Now read the source files of your system into the editor and use `m-X` Query Replace to convert any occurrences of **common-lisp-my-package** to **my-package**. (There will probably not be any unless your system is written in multiple packages.)

7. Recompile the system, and everything in this world should work.

Package Prefix Conversion of a Buffer

The output buffer produced by the Name Conflict Resolution step is our input buffer for this next step. Type:

```
m-X Convert Package of Buffer
```

After verifying the file name as usual, the conversion command asks for the names of the new packages to use, and shows the list of symbols to bypass. To change any of these values, click Middle on them. Confirm the values as they stand by pressing `END`.

The Conversion Tools then proceed with the package conversion. Note the large number of package prefixes (for example, **zl:**) that now appear in your program.

Package Prefix Conversion of a Region

Command: m-X Convert Package of Region

This works analogously to the Buffer version of the command, except that it operates only on the region you have marked.

Package Prefix Conversion of a Tag Table

Command: m-X Convert Package of Tag Table

Converting files in a Tag Table works in similar fashion to buffer conversion: first you see a list of all the packages used by files in the Tag Table and you select those you want to convert and what package to convert them to. Next you are asked about bypassing the conversion of symbols that are more often used as variables than as functions. Finally, you are asked to confirm the list of files to be converted (all files that are in the Tag Table and in a package that is being converted).

Next you are asked whether the file attribute lists should be set to the new package, and from that point on, the package conversion proceeds automatically until done.

Zetalisp to Common Lisp Conversion

The Zetalisp to Common Lisp Conversion Tools convert programs from Symbolics' Zetalisp dialect to Symbolics' extended version of Common Lisp. The kinds of changes needed to convert your code from the Zetalisp to the Common Lisp package structure and syntax are summarized in the six-step breakdown of the Zetalisp portion of the Conversion Tools system as follows:

1. Syntax Conversion
2. Radix Conversion (optional)
3. Name Conflict Resolution
4. Package Prefix Conversion

4a. CL Prefix Removal (optional)

5. Function Conversion

5a. Remaining ZL Prefix Removal (optional)

6. Structure Conversion

These six steps are executed entirely via Zmacs commands that run over files read into the editor buffer. The conversion commands are simple and easy to use; some are largely automated; others depend on your input for correct results. A Help facility, invoked by pressing the HELP key, is also available.

Each of the six steps produces a program that is complete, can be recompiled, and that functions equivalently to the source program. Each step is dependent on its predecessor and must, therefore, occur in the enumerated order, including the three optional steps, 2, 4a and 5a, if you choose to execute them.

Note

The Zetalisp to Common Lisp Conversion Tools do not necessarily produce portable Common Lisp code. Using the Common Lisp package is helpful, since after conversion Symbolics Common Lisp symbols will be identifiable by their **scl:** prefixes; but the Zetalisp to Common Lisp Conversion Tools do not especially flag conversions that might involve the use of functions that have Symbolics Common Lisp extensions to Common Lisp.

The Conversion Tools specifically don't do the following function conversions:

- Zetalisp functions for which there is no Common Lisp analog
- Zetalisp functions used inside a macro
- Complex Zetalisp functions, or those requiring rearrangement of the code in their vicinity

Unconverted functions will, of course, remain prefixed by **zl:** as a result of the Package Prefix conversion; they'll be easy to find either visually or with Zmacs Search commands.

We now present a step-by-step description of the Zetalisp to Common Lisp Conversion Tools, listing the Zmacs commands used to perform each step.

Step One: Syntax Conversion

Commands:
```
m-X Convert Lisp Syntax of Region
m-X Convert Lisp Syntax of Buffer
m-X Convert Lisp Syntax of Tag Table
```

Zetalisp uses "/" as the syntax-quoting (escape) character. Common Lisp uses "\" for that purpose. Since the syntax is reliable in both cases, the conversion is automated.

The syntax conversion done in this first step results in the following changes:

ZL character	*Becomes CL character*
//	/
#/	#\
#\	#\
/	\
\	\\

#^	**#\control-**
#Q	**#+lispm**
#M	**#+Maclisp**
#N	**#+NIL**

(The last four character changes are listed only for the sake of completeness; they are unlikely to appear in any files.)

The buffer syntax conversion tool changes the syntax in the file attribute line to Common Lisp. The tag table syntax conversion tools asks whether to change the syntax in the file attribute line; normally you answer yes.

Syntax Conversion of a Buffer

Type:

```
m-X Convert Lisp Syntax of Buffer
```

The system verifies the name of the input buffer, as usual for file and buffer related commands. (Use HELP to see a display of all your buffer names, if you don't want to convert the current buffer.)

The syntax conversion then proceeds automatically. The file attribute line is changed to Common Lisp from Zetalisp, and all uses of the Zetalisp quoting character "/" have changed to the Common Lisp quoting character "\".

Syntax Conversion of a Region

Command: `m-X Convert Lisp Syntax of Region`

To apply the syntax conversion to a region, first mark the region, then issue the conversion command. Only the region you marked is converted, and the rest of your code remains unchanged.

Syntax Conversion of a Tag Table

Command: `m-X` Convert Lisp Syntax of Tag Table

You can use the Tag Table option of the conversion command if you have previously issued either `m-X` Select System As Tag Table or `m-X` Select Some Files As Tag Table. When you type the conversion command, the system first reads in all the files in the Tag Table that are in Zetalisp. It then displays a list, "Files to be converted" and asks whether to convert all, none, or some of these files. If you opt for conversion, the system asks whether it should set the file attribute lists also. The syntax conversion then proceeds automatically; the system displays the name of each file as it is converted, ending with the message "Done."

Be sure to test your program after the syntax conversion, and to save the buffer before proceeding to the next step.

Step Two: Radix Conversion (Optional)

If any of your files have a base of 8 (octal), you may wish to convert them to base 10 (decimal); if so, you must do it now. This step is completely optional, since Common Lisp programs can have nonstandard radices.

Commands:

```
m-X Convert Base of Region
m-X Convert Base of Buffer
m-X Convert Base of Tag Table
```

After verifying the buffer (or files, in the case of Tag Tables) to convert, the conversion commands give you the option of having large octal numbers (numbers larger than 10 octal) converted to decimal automatically or selectively. If you opt for selective conversion, you are queried separately for each "large" number in your program. The range of responses is:

Action	*Character or Key*
Do it	Y, SPACE
Skip it	N, RUBOUT
Manual Edit	c-R
Do it, then Allow Edit	, (comma)
Redisplay screen	c-L, REFRESH

If you edit your program manually during conversion, press END to signal completion of your edit and return to the current conversion step.

Step Three: Name Conflict Resolution

Commands:

```
m-X Find Conflicting Symbols in Buffer
m-X Find Conflicting Symbols in Tag Table
```

In your Zetalisp program you may have defined some functions or variables whose names will conflict with the names of Common Lisp functions after conversion. For instance, you might have defined a function named **search** that conflicts with the Common Lisp function **search**.

Before your program can be moved to a new package, all function and variable names that conflict with the names of functions or variables inherited by the new package must be changed. This happens in two stages. First the Conversion Tools generate a buffer that lists all conflicting symbol names. You then edit this buffer, typing a new name after each of these symbols. During this edit you can also delete lines whose symbols are not truly conflicting, such as symbols used only for naming local variables. Then use m-X Multiple Query Replace command to substitute the new name for the old one in the source program.

Name Conflict Resolution of a Buffer

The output buffer produced by the Syntax Conversion step is our input buffer for this next step. Type:

```
m-X Find Conflicting Symbols in Buffer
```

After verifying the filename, the system asks how the buffer's package will be converted. See the section "Step Four: Package Prefix Conversion".

Select your choice. Now the system creates a buffer:

```
[Creating ZMACS Buffer "Conflicting symbols".]
```

and places you in it. In our example, the Conflicting Symbols buffer might contain only the name

```
SEARCH
```

Now edit the buffer to supply a new name. The new name must appear on the same line as the old name, immediately following it. After editing, our sample buffer of conflicting symbols looks like this:

```
SEARCH  MY-SEARCH
```

Move back to your input buffer and type:

```
m-1 m-X Multiple Query Replace from Buffer
```

(m-1 specifies that only whole-word occurrences of the symbol are picked for substitution.) The Query Replace command works in the usual fashion, letting you confirm or bypass the substitution in each individual case.

Name Conflict Resolution of a Tag Table

Command: m-X Find Conflicting Symbols in Tag Table

If you are working with a Tag Table, the conversion command asks how the packages used by files in the Tag Table will be converted. It then creates the buffer of conflicting symbols, which you edit as described earlier. Now type:

```
m-1 m-X Tags Multiple Query Replace from Buffer
```

Zmacs displays each occurrence of a symbol to be replaced, and you respond with an instruction to replace or skip, as appropriate, then press c-. to continue. See the section "Performing Operations with Tag Tables". The system notifies you when it has finished all substitutions.

Step Four: Package Prefix Conversion

Commands:
```
m-X Convert Package of Region
m-X Convert Package of Buffer
m-X Convert Package of Tag Table
```

This step modifies a program so that it can be read in a Common Lisp package instead of a Zetalisp package, but still get the same symbols. Each time a symbol inherited from Zetalisp is referenced, if the same symbol does not occur in Common Lisp a **zl:** package prefix is inserted in front of the symbol. No symbol substitution is done; the file is just changed to read the same global symbols into the new package. For instance, **memq** is converted to read **zl:memq**, not to the corresponding Common Lisp function name, **member** (that translation occurs later).

For a description of this step, see the section "Package Conversion".

A number of Zetalisp global symbols are much more often used as the names of local variables than as functions; examples are **zl:args** and **zl:array**. The Conversion Tools display a list of such symbols and offer you the option of suppressing their conversion. This will later save you the trouble of removing the unnecessary **zl:** prefixes generated by the conversion. You can edit the list, adding or removing symbols as required.

Step Four-a: CL Prefix Removal (Optional)

If your program was already using functions in the Common Lisp package by means of an explicit **cl:** package prefix, you may want to remove these prefixes now. Use the standard Zmacs Search and Replace commands for this purpose.

Step Five: Function Conversion

Commands:
```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

Many Zetalisp functions and variables readily translate into their corresponding Common Lisp functions. Three such types of translation are performed in this step.

The simplest translation is the direct substitution of one symbol for another; you can opt to have the system do these without querying you about each case. Other translations may involve some changes such as the addition of a keyword, the changing of an expression to a list, a changed order of arguments, and so on. In such cases the conversion command displays the proposed change, and asks you to confirm it. The most complicated renamings involve cases where there may be more than one option for translating a given Zetalisp function. Here, the conversion command displays the available options, along with some explanation about each, and asks you to select the most appropriate among them. This is where maximum familiarity with the code and the conversion set comes into play.

Before doing the function conversion, the system also prompts you for a conversion set; typically this is Zetalisp to Common Lisp. It can also be any of several other predefined conversion sets, such as Flavors to CLOS, or a conversion set that you have defined yourself.

As already stated, Zetalisp functions without a Common Lisp analogue, Zetalisp symbols not appearing their usual syntactic context, and complex Zetalisp functions are not translated in this step. However, all such untranslated functions remain prefixed by **zl:** as a result of Package Prefix Conversion, so you can find them and deal with them yourself later.

The substituting mechanism is careful not to lose comments; they may not, however, end up exactly in the right place in the rearranged program. Indentation might also have changed. You should look the program over as the conversion progresses.

Function Conversion of a Buffer

The output buffer produced by the Package Prefix Conversion step is our input buffer for this next step. Type:

```
m-X Convert Functions of Buffer
```

After verifying the file name as usual, the Conversion Tools prompt for a conversion set; the options can be displayed by pressing HELP.

The **defstruct** option is for Structure conversion, the final conversion step. Here, select Zetalisp to Common Lisp as the conversion set.

Next indicate whether or not you want to be queried for straightforward renamings. Since these are simply direct symbol substitutions, the favored approach is to let the system do them automatically.

After doing the simple conversions, the system uses a typeout window to query you on all other substitutions. Before each query, the system displays the affected line(s) in bold along with the immediate context so you can identify the code. For a table of your possible actions at this point, see the section "Getting Help with Conversion".

Function Conversion of a Region

Command: m-X Convert Functions of Region

This works analogously to the Buffer version of the command, except that it operates only on the region you have marked.

Function Conversion of a Tag Table

Command: m-X Convert Functions of Tag Table

The sequence of events is as for the Buffer version: the conversion command prompts you for a conversion set, asks if you want to be queried for simple renamings, and displays proposed changes or options, requesting a response. It displays each file name as it finishes its conversion; its last message is "Now no more sets of possibilities."

Step Five-a: Remaining ZL Prefix Removal

Some Zetalisp functions may have been too complicated to convert automatically because the change requires rearranging the code in the vicinity of the function.

Structure definitions and constructors are still unconverted, and should be left with their **zl:** prefixes until the Structure Conversion step.

Some Zetalisp functions might remain, however, because a global symbol is really being used in an innocent way and is not suppressed explicitly during the initial package conversion. Such **zl:** prefixes can be removed altogether, since the new local symbols serve better. Use the standard Zmacs Search and Replace commands to do this.

Step Six: Structure Conversion

Commands:

```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

To convert structures, you use the same command as for Function Conversion, except that this time you specify **defstruct** as the conversion set instead of Zetalisp to Common Lisp. As before, you have the option of doing straightforward renamings without a query. Keep in mind that since the system being converted is not guaranteed to be loaded, you will be asked about all forms beginning with **make-** that have an even number of arguments and no keywords other than **:make-array**. This may include a number of legitimate function calls.

For the most part, the **defstruct** macro is compatible with its **zl:defstruct** counterpart. However, some of the options accepted by both have a slightly different behavior when given to **cl:defstruct** than when given to **zl:defstruct**. In some cases, default behavior when no options are given also differs. There are, as well, differences in the format of the constructors generated by each **defstruct**; these are discussed in more detail below. For these reasons, we need a special conversion phase just to convert structure definitions and constructors. This phase deals with all **defstruct** forms, and all forms beginning with **make-** that appear to be structure constructor macros. You will probably need to do some editing of the conversion results, as we explain later.

In Zetalisp, you give the structure component names to the constructor macro as arguments for initializing these components, or slots. Common Lisp constructor functions take instead keyword arguments with the same name as the structure components. Further, Common Lisp constructor functions don't accept keyword arguments to the **:make-array** keyword. This Zetalisp form of a constructor macro, for example, is not acceptable to Common Lisp:

```
:make-array (:length 20)
```

The Conversion Tools let you get around this restriction by offering a Symbolics Common Lisp extension to **defstruct** that adds the option **:constructor-make-array-keywords** to a **cl:defstruct** macro. This option is followed by a list of all the **:make-array** arguments destined for use by the constructor function; these arguments then become top-level arguments to the constructor instead of appearing as keyword arguments to the **:make-array** keyword argument in the constructor. Only arguments included in the **:constructor-make-array-keywords** option to **defstruct** can be used in the constructor function.

If you opt for this approach, and if during conversion the system finds that the **zl:** constructor contains **:make-array** keyword arguments that were not explicitly specified in the **zl:defstruct :make-array** option, it gives you an explicit message, telling you to add the missing keywords. The message is also recorded in the compiler warnings database to let you do your editing all at once after the conversion.

Warning: Currently, the constructor function always expands to a call to **zl:make-array**. This works in most cases, but could create problems if your Zetalisp structure definition specifies a type for the new structure (or any other keyword argument to **zl:make-array** that differs from a **make-array** keyword).

Structure Conversion of a Buffer

All previous phases of Common Lisp conversion must have already been executed on the buffer before performing Structure Conversion.

This section uses examples in the file

SYS:CONVERSION-TOOLS;CONVERSION-STRUCTURES-TEST-PROGRAM.LISP

Type:

```
m-X Convert Functions of Buffer
```

After verifying the filename to convert as usual, the conversion command asks for a conversion set to use; this time, type **defstruct** . As for function conversion, you are asked if the conversion should do straightforward renamings without query; answer Yes. Straightforward renamings include converting **zl:defstruct** to **defstruct**, changing the names of structure components in constructor functions to keywords, and changes involving different default behavior in Zetalisp and Common Lisp **defstruct**, as in the use of **:conc-name** in our sample.

After querying you for each conversion (press HELP for the full list of possible responses), the system produces the new code. Major changes are as follows:

All **zl:defstruct** macros have become Common Lisp **defstruct**.

In all the converted constructor functions, the names of the structure components have become keyword arguments.

Example differences in zl: and cl: defstruct default behavior

The treatment of the **:conc-name** option to **defstruct** points to differences in the default behavior of the Zetalisp and Common Lisp forms. The **zl:defstruct** for the structure **my-structure** contains a **:conc-name** option without an argument. This form of the option is the default for **defstruct**. Therefore, **:conc-name** becomes redundant and is removed during the conversion. In contrast, the **zl:defstruct** for the structure **header-structure** in the example does not contain a **:conc-name** option, because we wanted to use the default version of the **zl:defstruct**, which is the **:conc-name nil** option. Since this is not the default behavior for **defstruct**, the conversion adds an explicit **:conc-name nil** option.

Treatment of :make-array keywords

The **zl:defstruct** for the structure **header-structure** in our example specifies that the structure should be implemented as an array, and uses a **:make-array** option to define the array length.

The Zetalisp constructor macro, **make-header-structure**, also uses the **:make-array** keyword, followed by two keyword arguments to it, namely **:length** and **:displaced-to**. To allow use of the latter two in the CL constructor function, which does not accept keyword arguments to **:make-array**, the conversion did the following:

It replaced the **:make-array** option in the **zl:defstruct** with the Symbolics Common Lisp option **:constructor-make-array-keywords**, following it with the former **:length** argument to **:make-array** which now serves as an argument rather than a keyword.

It also replaced the **:make-array** option in the Zetalisp constructor macro by the top-level keyword arguments, **:length** and **:displaced-to**.

While converting the constructor macro, the system finds that the constructor **:displaced-to** argument is missing from the argument list of the **:constructor-make-array-keywords** option in the **defstruct**. This is because it was not specified in the **zl:defstruct**, appearing only in the constructor. The system therefore displays a message, telling you to add this argument to the list of arguments following the **:constructor-make-array-keywords**, in the **defstruct**, so that the constructor function can subsequently use it. To deal with all such messages together at the end of the conversion, execute m-X Edit Compiler Warnings, which displays the Compiler Warnings database.

If your zl:defstruct used the :type option:

Suppose your original code specifies that the structure be implemented as a string array, as follows:

```
(zl:defstruct (header-structure :array-leader
                                (:make-array (:length 20
                                              :type 'zl:art-string)))
  slot-1)

(defun make-one ()
  (make-header-structure slot-1 'one :make-array (:length 40)))
```

This is then converted to the following Common Lisp form:

```
(defstruct (header-structure (:conc-name nil) :array-leader
                             (:constructor-make-array-keywords
                               (length 20) (element-type 'string-char)))
  slot-1)

(defun make-one ()
  (make-header-structure :slot-1 'one :length 40))
```

Since the constructor function currently expands to a call to **zl:make-array**, the form **(zl:make-array 20 ... :element-type 'string-char)** would result. This is not legal, since Zetalisp recognizes only **type** as an array type specifier.

This problem will be fixed in a future release. Currently, you can get around it by editing the converted code of the **defstruct** to restore the original Zetalisp type declaration:

```
(defstruct (header-structure (:conc-name nil) :array-leader
                             (:constructor-make-array-keywords (length 20)
                                                               (type 'zl:art-string)))
  slot-1)
```

Creating Your Own Conversion Set

It's easy to create your own conversion set, which can be invoked by specifying its name to one of the `m-X Convert Functions of ...` commands. You don't have to know anything about the internals of the editor; all you have to do is define translations from forms to forms, very much in the style of **defmacro**. The system takes care of parsing the text in the editor buffer into a form, calling your conversion, querying the user about the conversion, and writing the new form back into the editor buffer, maintaining comments and indentation.

The general method of operation of function conversion is to search the buffer for a string that looks like it matches a conversion in the conversion set. When such a string is found, the innermost pair of balanced parentheses containing it marks the form to be converted. The system reads this form with **read** and looks for one or more matching conversions. It calls each conversion. A conversion can return a replacement form, or can return **nil** if it is not really applicable. The argument template you specify when you define a conversion compiles into code that returns **nil** if the actual form doesn't match the template. The body of your conversion can make additional checks and return **nil** if they are not satisfied. If no applicable conversion is found the system goes back to searching the buffer, without saying anything.

Once the set of matching and applicable conversions has been determined and their replacement forms have been collected, the system displays the context and queries the user whether to make the conversion, and which one to make if there are multiple possibilities. If the user replies affirmatively, the system determines the differences between the old form and the new form, and edits the buffer so that it will read as the new form. This editing process is done in a way that preserves comments and most indentation.

Besides function conversions, a conversion set can also define conversions for funargs and for bare symbols. A function conversion applies to a symbol as the first element of a list. A funarg conversion applies to **#'***symbol* or **(function** *symbol***)**. A symbol conversion applies to a symbol regardless of context. Funarg and symbol conversions are simpler since there are no arguments and hence no template matching and conditional processing.

To define a conversion set, start by using the macro **conversion-tools:define-conversion-set**. This defines macros to define function, funarg, and symbol conversions for that conversion set. Once you have defined the conversions, you can use the `m-X Convert Functions of Region` command to apply the conversion set to test cases and see if it works as desired.

To see a fairly complex example of defining a conversion set, see the file `SYS:CONVERSION-TOOLS;ZETALISP-CONVERSIONS.LISP`.

conversion-tools:define-conversion-set *set-name symbol-macro function-macro* &key *:funarg-macro :message-macro :send-functions :pretty-name :search-strings :default-conversions* *Function*

Defines a conversion set named *set-name*, which can be run by specifying its name to one of the `m-X Convert Functions of ...` commands.

If *symbol-macro* is non-**nil**, it is the name of a macro that defines a symbol conversion for this conversion set. A symbol conversion applies to a symbol regardless of context.

If *function-macro* is non-**nil**, it is the name of a macro that defines a function conversion for this conversion set. A function conversion applies to a symbol as the first element of a list.

If *funarg-macro* is non-**nil**, it is the name of a macro that defines a funarg conversion for this conversion set. A funarg conversion applies to **#'***symbol* or **(function** *symbol***)**.

If *message-macro* is non-**nil**, it is the name of a macro that defines a message conversion for this conversion set. A message conversion applies to any list whose first element is one of the symbols in *send-functions* (which defaults to just **send**) and whose third element is constant and a symbol.

You can use *send-functions* to make the *message-macro* apply to **funcall** as well as **send** if you like.

If *pretty-name* is specified, it is a string that names the conversion set. The `m-X Convert Functions of ...` commands accept this string. If *pretty-name* is not specified, it defaults based on *set-name*.

If *search-strings* is specified, it is a list of strings to search for to find relevant symbols in the editor buffer. If *search-strings* is not specified, the search uses all of the symbols for which a conversion has been established. Sometimes speed can be improved by specifying *search-strings*, for example if all relevant symbols will have a certain package prefix.

If *default-conversions* is specified, it is a list of symbols that name function conversions (see the **:name** option to the *function-macro*). These conversions are applied to all function calls, after any conversions that are specifically for the function.

The arguments to the *function-macro* are *name template* &key *:form :name :modification-depth :documentation :documentation-level :documentation-length*. As an abbreviation, the *function-macro* can be called with simply *name template form* if none of the other options are required. The arguments are

name	the symbol that names the function. Calls to this function are subject to the conversion being defined. This works for macros and special forms as well as true functions.
template	a **defmacro**-style argument list that matches the function call being converted. The variables bound by *template* are available in *:form*; their values are subforms of the function call being converted. If the function call being converted does not match the template, for example the number of arguments is different, this conversion is quietly ignored.
:form	a form that evaluates to the replacement form, or to **nil** if this conversion is not really applicable. Typically this is a backquote expression. If you need more than one form, you must enclose them in **progn**; the *function-macro* does not have a body. Within this form, the variable **conversion-tools:function-name** is bound to the name of the function in the function call being converted. This can be useful in connection with *default-conversions*.
:name	a symbol that names the conversion. This defaults based on *name* and the name of the conversion set, so it only has to be specified when there is more than one conversion defined for the same function.
:modification-depth	the depth down from the original form in which lists have some elements changed. It is usually 1 except for things like selectq → case, where a t might turn into an otherwise, thus a modification depth of 2 is required because there are 2 levels of parenthesis around the otherwise. *:Modification-depth* helps in the editing of the buffer to install the replacement form. *:Modification-depth* defaults to 1.
:documentation	a string that briefly describes what this conversion does.
:documentation-level	controls ***print-level*** while printing the replacement form, when querying the user. This defaults to a smart default.

:documentation-length	controls ***print-length*** while printing the replacement form, when querying the user. This defaults to a smart default.

The arguments to the *symbol-macro* are *old-symbol* and *new-symbol*. Wherever the *old-symbol* is encountered, it is replaced by the *new-symbol*. If a function or funarg conversion is also defined for *old-symbol*, and the context where the symbol appears is appropriate, the function or funarg conversion is used instead of the symbol conversion.

The arguments to the *funarg-macro* are *name* &key *:new-function :documentation :documentation-length :documentation-level :modification-depth :name*. As an abbreviation, the *funarg-macro* can be called with simply *name new-function* if none of the other options are required. The arguments are

name	the symbol that names the function. References to this function as a constant (using #' or **function**) are subject to the conversion being defined.
:new-function	the symbol that is to be used instead.
:documentation	a string that briefly describes what this conversion does.
:documentation-level	controls ***print-level*** while printing the replacement form, when querying the user. This defaults to a smart default.
:documentation-length	controls ***print-length*** while printing the replacement form, when querying the user. This defaults to a smart default.
:modification-depth	the depth down from the original form in which lists have some elements changed. It is usually 1 except for things like selectq → case, where a t might turn into an otherwise, thus a modification depth of 2 is required because there are 2 levels of parenthesis around the otherwise. *:Modification-depth* helps in the editing of the buffer to install the replacement form. *:Modification-depth* defaults to 1.
:name	a symbol that names the conversion. This defaults based on *name* and the name of the conversion set, so it only has to be specified when there is more than one conversion defined for the same function.

The arguments to the *message-macro* are *message template* &key *:flavor :form :name :modification-depth :documentation :documentation-level :documentation-length*. As an abbreviation, the *message-macro* can be called with simply *message template form* if none of the other options are required. The arguments are

message	the symbol that names the message to be converted.
template	a **defmacro**-style argument list that matches the message send being converted. The first argument matches the object receiving the message; the remaining arguments match the argu-

ments to the message. The variables bound by *template* are available in *:form*; their values are subforms of the message send being converted. If the message send being converted does not match the template, for example the number of arguments is different, this conversion is quietly ignored.

:flavor — a symbol that names the type of object receiving the message, or a list **(or** *type type...***)** that names several types that could receive the message. If *:flavor* is specified, then the conversion only applies if the object receiving the message appears to be of the required type; this is assumed if the object is the value of a variable with the same name as the type, otherwise the user is queried.

:form, *:name*, *:modification-depth*, *:documentation*, *:documentation-level*, *:documentation-length*
these arguments are the same as for the *function-macro*.

The Demonstrations Facility

The demonstrations facility is an extensible facility for using and writing programs that demonstrate or highlight interesting aspects of a system-defined or user-defined program or application.

Using the Demonstrations Facility

To use the demonstration facility, use these CP commands:

Show Demonstration Names Command

Show Demonstration Names

Shows the names of demonstrations which are available in the demonstrations facility.

The output is mouse sensitive. Consult the mouse documentation line for information on the available options.

The output is partitioned into "loaded" and "unloaded". The demonstration definition resides in the file system and is not loaded until it is first referred to. Loading a demonstration loads only its definition, not its supporting code, and is therefore always fast. When a demonstration is first run, it is initialized, which may take longer.

Run Demonstration Command

Run Demonstration *name keyword*

Runs a demonstration. If the demonstration has not been initialized, it will be initialized automatically before it is first run.

name	The name of a demonstration.
keywords	:More Processing, :Output Destination, :Show Instructions, :Specify Options
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Show Instructions	{Yes, No, Ask} Specifies whether to show instructions. The default is Yes.
:Specify Options	{Yes, No} Specifies whether to interactively prompt for option values (if any) to control the demonstration.

Initialize Demonstration Command

Initialize Demonstration *name keyword*

Does any pre-loading specified in the demonstration definition, such as loading any required systems and running the initializer function. This command is not functionally necessary (because when a demonstration is first run, it is initialized automatically, if it has not been initialized yet), but it can save time in situations when you want to have the first run of a demonstration be fast.

name	The name of a demonstration.
keywords	:Force
:Force	{Yes, No} Specifies whether to force initialization even if it has already been done. (Even when this is Yes, required systems are not re-loaded—only the initializer action is re-run).

Show Demonstration Summary Command

Show Demonstration Summary *name keyword*

Shows a brief summary of what the demonstration does.

name	The name of a demonstration.
keywords	:More Processing, :Output Destination
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Show Demonstration Instructions Command

Show Demonstration Instructions *name keyword*

Shows the essential instructions on how to use the demonstration.

name	The name of a demonstration.
keywords	:More Processing, :Output Destination
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Show Demonstration Legal Notice Command

Show Demonstration Legal Notice *name keyword*

Shows any copyright notices related to the demonstration.

name	The name of a demonstration.
keywords	:More Processing, :Output Destination

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Show Demonstration Background Information Command

Show Demonstration Background Information *name keyword*

Displays background information about the demo which might help you understand where the demo came from or what it is trying to show, but which is not essential to actually running the demo. For example, if the demonstration is the Life game, this Show Demonstration Background Information might give information about who invented the Life game, the rules of the game, and other interesting information.

name	The name of a demonstration.
keywords	:More Processing, :Output Destination

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Extending the Demonstrations Facility

The package **demonstration**, whose short name is **demo**, supports the demonstrations facility.

This section documents a variable and a function that are central to the demonstrations facility, and then describes how to install a new demo.

demo:*demonstration-pathnames* *Variable*

A list of the (possibly wildcarded) pathnames which are searched by the demonstrations facility (Show Demonstration Names, Run Demonstration, and so on.)

The default is `("SYS:SITE;*.DEMO.NEWEST")`. This means that demonstration definitions can be installed by creating a file in SYS:SITE;*demonstration-name*.DEMO with the appropriate contents. However, since the files in this directory might change when new system versions are installed, maintainers of site systems might wish to create an alternate directory for the purpose of holding site-specific demonstration definitions. If this is done, an appropriate wildcard should be added to this list by the site system. For example:

```
(pushnew "acme:>corporate-demos>*.demo.newest"
         demonstration:*demonstration-pathnames*
         :test #'string-equal)
```

demo:define-demonstration *name options (*&key *:pretty-name :required-systems :restrictions :initializer :instructions :background-information :legal-notice :initialize) summary* &body *forms* *Function*

Defines a demonstration called *name*.

options are formal parameters to the *forms* in the body. Each *option* has the form: (*var init type key1 val1 key2 val2...*) where *var* names a variable, *init* is *var*'s initial value, *type* is var's type (must be a valid type argument for **accept**), and the remaining *keys* and *vals* are other arguments to **accept**.

The keyword arguments to **demo:define-demonstration** are described below. All keyword values are evaluated normally (so some values may require quoting).

:pretty-name	A string that must be **string-equal** to the given name, but can be used to change the case to something prettier. The default is **(string-capitalize** *name***)**.
:required-systems	A list of systems which must be loaded when this system is initialized. The default is **'()**.
:restrictions	A list of restriction keywords for this demonstration. Possible keywords include:

:large-screen-only
Does not work with a small screen.

:local-screen-only
Does not work with a remote screen.

:local-terminal-only
Does not work with a remote terminal. The default is **'()**.

:initializer A function which is run once (the first time the demonstration is about to be run, or the first time Initialize System is used) to initialize the demonstration.

:instructions A string containing instructions for using the demonstration, or a function of one argument (a stream) which will display the instructions.

:background-information
Like **:instructions**, but contains additional interesting information not essential to actually running the demonstration.

:legal-notice Like **:instructions**, but contains any legal information associated with the demonstration.

:initialize A boolean value indicating whether to initialize the demonstration as soon as it is loaded. The default is **nil**.

For some important information about the order of loading the demonstration definition and the demonstration's system, See the section "Installing a Demonstration".

summary is a short string which describes the action of the demonstration for menu purposes; this is used by the Show Demonstrations command.

forms are the forms to be executed when the demonstration is actually run. They may refer to variables named in *options*.

Installing a Demonstration

To install a demonstration, create a file in one of the files on the list **demo:*demonstration-pathnames***. For example, the default value of this variable is "SYS:SITE;*.DEMO.NEWEST"; you might create a definition file named "SYS:SITE;MY-TOYS.DEMO".

The file can contain any Lisp forms, but somewhere they must include either directly (in the text) or indirectly (by using **load**) a definition for a **demo:define-demonstration** form for a demonstration whose name is **string-equal** to the file's name. For example, in the file "SYS:SITE;MY-TOYS.DEMO" we would expect to find something like:

```
;-*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(DEMO:DEFINE-DEMONSTRATION MY-TOYS ((SOME-PARAMETER 64. '(INTEGER 0 100)))
     (:REQUIRED-SYSTEMS '("MY-TOYS-SUPPORT")
      :RESTRICTIONS '(:LOCAL-TERMINAL-ONLY)
      :INSTRUCTIONS "Sit back, relax, and enjoy the ride."
      :LEGAL-NOTICE "Copyright (c) 1990 J. Doe.  All Rights Reserved.")
     "Show off my skills as a toymaker."
   (MY-TOYS-DEMO-DRIVER SOME-PARAMETER))
```

Note that this file might be loaded before MY-TOYS-SUPPORT is loaded, so you must be careful to either define functions it needs (such as MY-TOYS-DEMO-DRIVER, in the above example) in a package that will exist (such as **user**), or else you should be careful to not assume that the package exists. For example:

```
;-*- Mode: LISP; Syntax: Common-Lisp; Package: USER; Base: 10 -*-

(DEMO:DEFINE-DEMONSTRATION MY-TOYS ((SOME-PARAMETER 64. '(INTEGER 0 100)))
     (:REQUIRED-SYSTEMS '("MY-TOYS-SUPPORT")
      :RESTRICTIONS '(:LOCAL-TERMINAL-ONLY)
      :INSTRUCTIONS "Sit back, relax, and enjoy the ride."
      :LEGAL-NOTICE "Copyright (c) 1990 J. Doe.  All Rights Reserved.")
     "Show off my skills as a toymaker."
   (FUNCALL (INTERN "MY-TOYS-DEMO-DRIVER" "TOYS-INTERNAL") SOME-PARAMETER))
```