

Genera 8.1 Release Notes

Genera 8.1: Introduction and Highlights

Genera 8.1 includes several new features as well as a number of bug fixes and performance improvements. Genera 8.1 is a unified release for all Symbolics platforms, including the new MacIvory model 3.

The new features are as follows:

- Common Lisp Interface Manager (CLIM). CLIM provides a high-level language for defining presentation-based user interfaces that run under Genera and CLOE, on UNIX-based workstations, and on PCs that support Common Lisp. The important features of CLIM are:

- *CLIM provides a portable user interface.*

CLIM fits into an existing host system. With CLIM you can achieve the look and feel of the target host system without implementing it directly, and without using the low-level implementation language of the host system.

- *CLIM is based on a presentation model.*

The CLIM presentation model provides the advantages of having the visual representation of an object linked directly to its semantics. CLIM's presentation model is similar to that used in Dynamic Windows.

- *CLIM supports high-level programming techniques.*

Using CLIM capabilities you can develop a user interface conveniently, including formatted output, graphics, windowing, and commands that are invoked by typing text or clicking a mouse, among other techniques.

- Some features from the X3J13 Common Lisp, including Conditions and the LOOP Facility See the section "X3J13 Common Lisp Features". See the section "Documentation Update: **future-common-lisp:loop**". See the section "Documentation Update: X3J13 Conditions".
- Netbooting for Ivory machines. In Genera 8.1, netbooting Ivory machines works exactly the same as netbooting 3600-family machines. See the section "Netbooting". You should note, however, that you cannot netboot pre-Genera 8.1 worlds from Ivory machines.
- Significant performance improvement of Tables. The external interface is unchanged, but the underlying implementation has been redesigned and reimplemented for speed.

- Optical and SCSI disk drive support on XL machines. On XL machines, Genera 8.1 supports the Storage Dimensions MacinStor Erasable Optical 1000 erasable optical disk drive. You can create a FEP file system on an optical disk, and access the FEP file system in the usual ways. You can use the optical disk to store large files such as image files, world loads, and Staticce databases. For more information,

see the section "SCSI Device Support for XL Machines in Genera 8.1".

- Bundling of some software which was previously sold separately as layered products, including IP-TCP, Staticce Runtime, and several others.

Genera 8.1 bundles (includes without extra cost) some software which was previously sold separately as layered products. Sources and binaries for the following bundled software are now part of the regular Genera 8.1 Sources: IP-TCP, NFS, X-Windows, DMP1, LGP2/3, Delivery DocEx, and Staticce Runtime. See the section "Documentation Update: Staticce Runtime". NFS Supplemental sources are also now part of the regular Genera 8.1 Sources. Concordia Supplemental Sources are now part of the Concordia layered product.

- QIC-11 and 6250 bpi tape support on XL machines
- The X server now conforms to MIT X11 R4 (the X client already conformed to R4).
- Optional distribution on CD-ROM. Customers who receive distribution on CD-ROM will receive C, Fortran, and Pascal (including sources).
- Support for the MacIvory model 3.
- New virus protection software for MacIvories, **Symantec AntiVirus for Macintosh (SAM)**, which provides an Automatic Scan feature to continuously monitor your system as you work. See the section "Symantec AntiVirus for Macintosh".
- There is a new recommended format for boot files for Ivory machines, see the section "Changes to Boot Files for Ivory Machines".

Genera 8.1 includes all the improvements and modifications in Genera 8.0 XL, Genera 8.0 ECO #1, and Genera 8.0 ECO #2, as well as additional improvements. Therefore, this document includes documentation of improvements made in all three previous ECO releases. See the section "Documentation of Genera 8.0 ECO #1". See the section "Documentation of Genera 8.0 ECO #2". See the section "Genera 8.0 XL Documentation Update".

CLIM in Genera 8.1

Overview of CLIM

CLIM, the Common Lisp Interface Manager, is a portable high-level object-oriented user-interface management system. It provides a full spectrum of capabilities for describing the user interface of an application and implementing that user interface on a variety of computers and window systems.

CLIM is portable in several respects. It is written in a standard dialect of Common Lisp and CLOS, and is therefore executable by numerous computers provided by various vendors. Also, user interfaces described by CLIM are independent of any particular window system or output device, so that CLIM applications are readily portable to different hardware and software platforms.

CLIM provides high level facilities for describing a user interface. Complex user interface behaviors such as incremental redisplay, formatting textual and graphical output, context sensitive documentation, direct manipulation, and mixed keyboard-mouse interaction are easily described with a minimum of specialized code.

CLIM is object-oriented in several respects. It is implemented in Common Lisp Object System (CLOS), and uses object-oriented techniques to great advantage internally. More importantly, CLIM provides an object-oriented model for integrating an application with its user interface. This *presentation model* associates user interface behavior directly with application objects, which allows an application to describe its user interface in terms of its own semantics.

CLIM provides the following facilities:

- *Windows* — Convenient facilities for creating, placing, and manipulating windows, including managing margins and scroll bars.
- *Streams* — Stream-oriented, device-independent input and output of arbitrarily mixed text, graphics, and presentations.
- *Graphics* — A rich graphics model which includes a variety of common geometric shapes (such as lines, rectangles, polygons, and ellipses), drawing options (such as line thickness and joint shape), a sophisticated inking model for describing patterns and color, and full affine coordinate transformations (translation, rotation, scaling).
- *Styled text output* — The appearance of textual output (font family, typeface, and size) is specified with an abstract, device independent mechanism called text style.
- *Output recording* — A facility for capturing all output done to a window, which provides the basis for arbitrarily scrollable and redisplayable windows.
- *Presentations* — Presentations associate user interface behavior with application objects, using object-oriented programming techniques. The user interface of an application may be described in terms of its own semantics, using the high level language of presentation types, instead of the lower level language of keystrokes, mouse events, and widgets.

- *Menus and Dialogs* — Many types of menus and dialogs may be automatically generated, using the presentation type facility to describe the desired appearance and behavior.
- *Context-sensitive input* — An application accepts direction from the user by establishing a context in which certain classes of operations and operands are valid, using presentation types. The user interface system uses this context to provide appropriate documentation and feedback to assist the user, and assures the application that user-supplied values are appropriate.
- *Commands* — The user interface operations of an application are described by commands, which operate on presentation types. This uniform mechanism is used for all interaction styles, including direct manipulation, menus, dialogs, keystroke accelerators, and command lines.
- *Formatted output* — High-level macros allow applications to produce neatly formatted tabular and graphical displays with little additional programming.
- *Incremental Redisplay* — Recorded output may be changed and the display incrementally and efficiently updated, without extensive programming.
- *Application frames* — The screen layout and top level behavior of an application are described by application frames.

For detailed user and reference documentation about CLIM, see the section "Common Lisp Interface Manager (CLIM): Release 1.0".

CLIM incorporates refined versions of many features and concepts from Symbolics Dynamic Windows. See the section "Comparing and Contrasting DW and CLIM" for a description of their similarities and differences. There is a facility to assist in the task of converting Dynamic Windows software to CLIM, see the section "Converting from DW to CLIM". Note that conversion of existing Genera software to CLIM is wholly optional, and that Dynamic Windows will continue to be supported.

The CLIM Standard

The Common Lisp Interface Manager was defined and developed by a consortium of cooperating Lisp vendors, including Franz, International Lisp Associates, Lucid, Symbolics, and Xerox PARC. We consider CLIM a *de facto* standard, though it may be proposed for official standardization after the community has gained experience using it.

Symbolics provides versions of CLIM for Symbolics computers running Genera and IBM-compatible personal computers running CLOE. For other platforms, compatible versions of CLIM are available from the vendors listed above among others. Contact your Lisp vendor for availability information, or International Lisp Associates if your vendor has no current plans to offer CLIM.

Symbolics CLIM 1.0, and the CLIM implementations available in 1991 from other vendors, are based on a reference implementation of CLIM called CLIM Version 1. Throughout 1991, the CLIM consortium will be working on a new reference implementation which will incorporate the Silica technology developed at Xerox PARC. Implementations derived from CLIM Version 2 will be compatible with CLIM Version 1, but will provide additional functionality and better performance by integrating directly with popular window toolkits such as OSF/Motif, Open Look, and Microsoft Windows.

CLIM 1.0 Implementation Notes

This section summarizes some known limitations of the Symbolics implementation of CLIM 1.0, particularly with regard to rendering of graphic designs. Note that other implementations of CLIM 1.0 may have different limitations.

- The Genera debugger does not work on CLIM streams. If a CLIM application gets an error of any sort, the debugger will be invoked in a background window, and a notification issued to that effect. You may find it useful to tailor the behavior of these notifications, see the section "Pop-up notifications".
- **clim::make-contrasting-inks** provides up to 8 different colors or patterns of ink.
- **clim::make-contrasting-dash-patterns** provides up to 16 different dash patterns.
- CLIM 1.0 does not support composite designs. Also, CLIM 1.0 provides very limited support for opacity: opacity inks are supported, but are interpreted for rendering as either fully transparent or fully opaque.
- CLIM 1.0 contains limited support for patterned designs. The elements of a patterned design (whether a pattern, stencil, or tile) must be a color or opacity, and not a general design (a shape). Some previous versions of the documentation have referred to more stringent limitations; these limitations are no longer present.
- CLIM 1.0 does not support nonrectangular clipping regions. A clipping region is interpreted as the bounding rectangle of the region supplied.
- Some CLIM 1.0 implementations may not support tilted ellipses (ellipses not aligned with the X or Y axis). The Genera implementation does, but at some performance penalty when using a window system such as X or Macintosh which doesn't support tilted ellipses directly.
- PostScript streams do not yet create Encapsulated PostScript files. Their output can be printed on an Apple Laser Writer.
- For PostScript streams, line styles which specify **:line-unit normal** and a **:line-thickness** other than 1 may not produce the desired effect.

- **filling-output** doesn't work very well in the face of nontextual output (presentations and graphics).
- In Genera, there is no global user interface for changing the foreground and background colors of a CLIM application, they are under program control only. Using the Window Attributes menu to change the drawing and erasing alu functions, or FUNCTION C to change to inverse video, will not affect the foreground and background colors of a CLIM window. (At present, you must refresh the CLIM window to restore proper appearance after such a change.)

Installing CLIM in Genera 8.1

For Genera 8.1, CLIM is distributed as a loadable system on the Genera distribution tapes, which contains loadable binaries for all 3600 and Ivory computers, and all online CLIM documentation. Most CLIM source code is optional, and is available with the Genera supplemental sources. CLIM may be loaded on any Symbolics computer running Genera 8.1, using the following procedure.

1. Confirm that Genera 8.1 is installed.
Show Herald
2. Load the CLIM system.
Load System (a system [default System]) CLIM
3. Load the CLIM online documentation.
Load System (a system [default CLIM]) CLIM-Doc

Compatibility Issues for Genera 8.1 and Prior Releases

Compatibility Issues for MacIvory

The MacIvory Life Support shipped with Genera 8.1 requires Macintosh System Software 6.0.7 (or later) and 32-bit QuickDraw. Note that 32-bit QuickDraw is incompatible with old versions of the Radius Two Page Display's video card, where an "old version" is any TPD card earlier than rev 2.1.

The following information is useful for MacIvory sites that need to use both Genera 7.4 Ivory and Genera 8.0, 8.0.1, 8.0.2, or 8.1.

For a MacIvory to boot and operate successfully, the MacIvory Life Support version must be compatible with the Genera and IFEP versions. Note that MacIvory model 3 requires Genera 8.1 and the version of MacIvory Life Support shipped with Genera 8.1; the hardware cannot run on earlier software.

MacIvory Life Support and Genera and IFEP are completely interoperable within minor releases of Genera 8, including Genera 8.0, 8.0.1, 8.0.2, and 8.1. Thus, all versions of Genera 8 interoperate with all versions of MacIvory Life Support shipped with Genera 8.

However, note that there are incompatibilities between Genera 8 and 7.4. That is, if you are running Genera 7.4, you must use the FEP and MacIvory Life Support versions that were shipped with Genera 7.4. And if you are running Genera 8.0, 8.0.1, 8.0.2, or 8.1, you must use the FEP and MacIvory Life Support versions that were shipped with any of those versions of Genera 8.

Running Two Genera Releases on a Single MacIvory

It is possible, although tricky, to allow two different major releases to coexist on the same MacIvory system. This section describes how to prepare for running two Genera releases (what you need to keep, and how to organize the files), and how to switch from one release to the other.

For the purpose of a specific example, we assume you are running the Genera 7.4 and Genera 8.1 releases.

Preparation for Running Two Genera Releases

For each Genera release, you need to keep the Genera world, the corresponding FEP (kernel and flod files), and all Macintosh software (the three folders: MacIvory Applications, MacIvory System, and MacIvory Development).

For safety's sake, you should keep the floppies or CD-ROM distributed from Symbolics in a safe place, in case any of these files are deleted accidentally.

You should rename the folders so that they clearly represent the software in them:

- The Genera 7.4 versions of the folders should be named: 7.4 MacIvory Applications, 7.4 MacIvory System, and 7.4 MacIvory Development.
- The Genera 8.1 versions of the folders should be named: 8.1 MacIvory Applications, 8.1 MacIvory System, and 8.1 MacIvory Development.

You need to have three sets of HELLO.BOOT and BOOT.BOOT files, one for Genera 7.4, one for Genera 8.1, and one to identify which release to run:

- The 7-4-HELLO.BOOT file should load the FEP version for Genera 7.4. The 7-4-BOOT.BOOT file should load the Genera 7.4 world.
- The 8-1-HELLO.BOOT file should load the FEP version for Genera 8.1. The 8-1-BOOT.BOOT file should load the Genera 8.1 world.
- The HELLO.BOOT file should contain "Hello 8-1" or "Hello 7-4"; this indicates which of the two above HELLO.BOOT files to use. The BOOT.BOOT file should contain "Boot 8-1" or "Boot 7-4"; this indicates which of the two above BOOT.BOOT files to use.

Switching From One Genera Release to the Other

Assume you are running Genera 8.1 and want to switch to running Genera 7.4. Follow these steps:

1. *Before shutting down Genera 8.1*, edit the HELLO.BOOT file to contain "Hello 7-4". Edit the BOOT.BOOT file to contain "Boot 7-4".
2. *Before shutting down Genera 8.1*, you need to switch FEP kernels. When you are running Genera 8.1, you can do this by using the Edit Disk Label command. When switching from Genera 8.1 to Genera 7.4, you would indicate the I311 FEP (which was the 7.4 FEP).

Note, however, when you are running Genera 7.4, the Edit Disk Label command is not defined. Thus, to switch from Genera 7.4 to Genera 8.1, you need to use **si:install-fep-kernel**. For example, to switch to the I325 FEP which is the 8.1 FEP, you would evaluate the following form:

```
(si:install-fep-kernel "I325-kernel")
```

3. Shutdown Lisp, by choosing "Shut Down" from the Ivory pulldown menu. This quits the Genera application.
4. Copy the contents of the 7.4 MacIvory System folder to the System folder. **Be sure to hold down the Option key while dragging to copy, not move the contents of the folder**; you should always keep the contents of the 7.4 MacIvory System folder so you can copy it again later. The system asks if you want to replace the files with the same names; you should answer Yes.
5. Restart the Macintosh.
6. When the Macintosh is running again, open the 7.4 MacIvory Applications folder and run the Genera application. If you are asked whether you want to cold boot, answer Yes. (If you are not asked, then the system cold booted automatically.)

To switch from Genera 7.4 to Genera 8.1, use the same process, but switch "8-1" and "7-4".

Compatibility Issues for Symbolics UX400S

Symbolics UX400S Life Support and FEP/Genera are completely interoperable within minor releases of Genera 8.0. Thus, all versions of Genera 8.0 interoperate with all versions of Life Support shipped with 8.0.

However, note that you cannot mix Genera 8. and 7.4. That is, if you are running Genera 7.4, you must use the FEP and Life Support versions that were shipped with Genera 7.4. And if you are running Genera 8.0, 8.0.1, 8.0.2, or 8.1 you must

use the FEP and Life Support versions that were shipped with any of those versions of Genera 8. (Note that UNIX software shipped with Genera 8.1 is compatible with SunOS 4.1 only.)

Changes to the Lisp Language in Genera 8.1

X3J13 Common Lisp Features

The term "ANSI Common Lisp" has been used in many places in anticipation of a standard, but it is a misnomer. There is not yet such a standard, nor even a draft standard. An ANSI subcommittee, X3J13, is tasked with the creation of such a standard, and Symbolics is participating in that effort.

Since no public draft has been produced yet, and since there is an immediate need for the language which X3J13 has been working on, we have implemented some parts of the emerging standard which seem to be relatively stable. Several parts of that implementation are referred to within this documentation, and we have chosen the term "X3J13 Common Lisp" as a way of referring to those parts in order to emphasize the fact that this conforms only to our best estimate about the current state of the working drafts used by X3J13.

In all cases where "X3J13 Common Lisp" is referred to, please be aware that there might yet be changes to the X3J13 working documents which could necessitate corresponding changes in our implementation. Any mention of ANSI in this document or in the software it describes should not be taken to imply an endorsement by ANSI or any of its affiliated agencies.

The important pieces of X3J13 Common Lisp which Symbolics has implemented include:

- CLOS, which has been available and documented since Genera 8.0
- The Loop Facility
- Conditions

For documentation on the Loop Facility, see the section "Documentation Update: **future-common-lisp:loop**".

For documentation on Conditions,

Changes to CLOS in Genera 8.1

In Genera 8.1, **clos:describe-object** takes two arguments, an *object* and a *stream*, as specified by X3J13, and as documented in the Genera documentation. Previously, **clos:describe-object** took only one argument, the *object*.

Genera 8.1 improves the performance of CLOS programs by precompiling constructors, when possible. CLOS automatically makes a function to handle the specific

case when a compiler optimizer sees **clos:make-instance** with a constant class and constant set of keywords (but not necessarily constant values for the keyword arguments). These functions are compiled automatically the first time they are called, when the world is saved, and after Load System or Load Patches.

In Genera 8.1, **clos:symbol-macrolet** accepts declarations and adds type declarations to its macro expansions.

Genera 8.1 fixes a bug in which the compiler would get an error if it encountered an **:after** method for **clos:initialize-instance** with 45 keyword arguments on an Ivory machine, or 64 keyword arguments on an 3600-family machine.

The commands Show CLOS Generic Function and Show Effective Method now work correctly for encapsulated generic functions, such as occur when **trace** is used.

Changes to the Compiler in Genera 8.1

- The compiler was improved by adding error checking to be sure that the lambda-list keywords **&optional**, **&rest**, **&key**, **&allow-other-keys**, and **&aux** always occur in the correct order. Previously, an incorrect order was silently tolerated, sometimes having mysterious effects.
- In Genera 8.1, the compiler gives a warning if **&whole** appears anywhere other than at the front of the top-level destructuring pattern of a **defmacro**. *Common Lisp: the Language* does not allow this, hence the warning.
- The compiler now checks keyword arguments supplied in a function call against the keyword parameters accepted by the called function. This helps users who check programs for portability by compiling in the Common Lisp Developer and looking for warnings. 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**. The variable **compiler:*inhibit-keyword-argument-warnings*** can be set to **t** to disable this checking, for example if you have a lot of declared arglists that are malformed.
- Compiling a buffer or a form in the editor now produces the same behavior as loading the source of the file or form, except that optimizations are run and code is compiled in the process.
- Inline functions that did a **throw** were not always expanded correctly; this bug has been fixed.
- A bug in the compiler for 3600-family machines was fixed, in which certain uses of **rplaca** and **rplacd** gave spurious results. These cases involved using **rplaca**

and **rplacd** with a locative to a local lexical variable (to give a value to that variable) and then trying to get the value of the variable. For example:

```
(let ((location nil)
      (list nil))
  (setq location (locf list))
  (setf (location-contents location) 'a)
  list)
```

Although this example might seem obscure, similar code which triggers the bug is generated by the **collect** option to **loop**.

- Previously, recompiling or reloading a self-recursive function left the new function indirecting through the old one, thus degrading performance. This bug has been fixed.
- Previously, the compiler and interpreter would have differed in their interpretations of the following forms. Now they agree, and the use of **subst** in **use-of-subst** is not inlined:

```
(proclaim '(inline subst))
(defun subst (x)
  (+ x y))

(defun use-of-subst (a)
  (let ((y 3))
    (subst a)))
```

- The compiler is now able to expand local functions (**flet** and **labels**) inline. The default is not to inline local functions. Of course expanding recursive functions inline is not a good idea.
- The protection against recursive inline functions has been removed. This protection made the expanded code too complicated for **setf** and the compiler to compile properly. User code with recursive inline functions will loop in the compiler rather than working.
- A bug has been fixed in which recompiling **future-common-lisp:defstruct** and macroexpanding a simple structure definition would cause the machine to crash to the FEP.
- The **bin** file dumper can now dump objects with circular definitions.
- Warnings are now issued for duplicate declarations for variables. A new variable, **si:*duplicate-declarations-warnings***, default **t**, controls this behavior. If you are not interested in seeing duplicate declarations warnings, set this variable to **nil**.

Miscellaneous Changes to the Lisp Language in Genera 8.1

- The performance of Table handling has been significantly improved.
- **open** now works on user-defined types such as:

```
(deftype application-db-stream-element-type '(unsigned-byte 8))
(open "foo" :element-type 'application-db-stream-element-type)
```

This allows you to specify file-storage interfaces more abstractly, for example.

- **special** declarations are no longer pervasive. In the following example, the inner binding of **a** would have been special before and will now be lexical.

```
(let ((a 1))
  (declare (special a))
  (let ((a 2))
    (print (list a (symbol-value 'a)))))
```

- The **&key** default values for macros are now evaluated only when they are used. Previously, the **&key** default values were always evaluated, which was a bug.
- Some bugs in **:location** function specs have been fixed, in which infinite print loops could happen, as could problems with block reads of structures containing such functions. Also, the printed representation of certain locatives changed after being used in **:location** function specs.
- The Zetalisp **lambda** variant **subst**, which has not worked in compiled code since Genera 7.0, has been removed for Genera 8.0. A form such as

```
(subst (x) ...)
```

can be replaced with the following form:

```
(lambda (x) ...)
```

- The Lisp printer behaves differently when printing a function which is not current; in other words, a function which is not the same as what you would get if you did (**fdefinition** *'function-name*).

Previously, such a function was printed as:

```
#<DTP-COMPILED-FUNCTION FGH (Superseded) 21000742330>
```

Consider the following example, which shows one of many ways to get a function which is not current:

```
(defun my-interpreted-function (x)
  (print (+ x 2)))

(disassemble 'my-interpreted-function) =>
  0 ENTRY: 1 REQUIRED, 0 OPTIONAL      ;Creating X
  2 START-CALL-INDIRECT-PREFETCH #'PRINT
  4 PUSH FP|2                          ;X
  5 ADD 2
  6 FINISH-CALL-1-RETURN

#<Compiled function MY-INTERPRETED-FUNCTION (Not the current definition) 21000704545>
```

Note that the current definition is still the interpreted one, since **disassemble** compiles the function if it isn't already, but does not install the compiled version. Also note that the printer now says "Not the current definition" instead of "Superseded".

This Lisp printer change also removed the functions **si:print-random-object** and **flavor::print-generic-function**.

- There have been some additions to the *system-type* argument to the Ivory-only macro **sys:system-case**. The currently recognized system-types are:

<i>System-Type</i>	<i>System</i>
Native	A standalone Symbolics computer. Examples are XL400 and XL1200 systems.
MacIvory	A Symbolics processor embedded in a Macintosh modular computer. Examples are MacIvory model 1, MacIvory model 2, and MacIvory model 3.
UX	A Symbolics processor embedded in a Sun-3 or Sun-4 system. Examples are Symbolics UX400S and Symbolics UX1200S.
VME	A Symbolics processor in a VME-based system. This could be a standalone Symbolics computer such as the XL1200, or an embedded processor such as the UX1200S.
Embedded	Any Symbolics processor embedded in a host computer. Examples are any of the MacIvory models or the UX models.

- Nested uses of **sys:system-case** are now optimized.
- The **sys:system-case** clause (**otherwise (error ...)**) is now equivalent to omitting an **otherwise** clause. The one difference is that this clause, in effect, states that you are aware that all other system types should signal an error and you will not be warned about missing required system types at compile time.

- A **sys:system-case** clause of the form (**never** *keylist*) is now accepted and is equivalent to the clause (*keylist* (**error**)). Clauses of this form may be used only if no **otherwise** clause is specified.
- **destructuring-bind** now handles **&optional** defaults correctly when **quote** or **list** forms are used.
- Processes that are about to go into "output hold" wait signal the condition **tv:output-on-deexposed-sheet**. Clients who want to intercept and avoid the output-hold can do so by binding a handler (using **condition-case** or **condition-bind**) and using **sys:proceed** with *proceed-type* **:retry** when the output-hold condition has been corrected. For compatibility, this occurs after the defined output-hold actions have already occurred, so in effect users get control only if they would have been stuck in a **process-wait**.
- The variable **cltl:*features***, used by the Common Lisp Developer, used to contain (:SYMBOLICS 3600 :CLTL :IEEE-FLOATING-POINT) regardless of the machine type, which was a bug. It now correctly contains :IMACH instead of 3600 on Ivory-based machines.
- The **format** directives `~\date\`, `~\time\`, `~\datetime\` and `~\time-interval\` were changed to present dates, times, and time intervals using the appropriate presentation types. Previously, the output of **format** for these directives was not presented using presentation types.
- The presentation type **type-or-string** now accepts a presentation-type keyword **:string-if-quoted**:

:string-if-quoted

Takes a boolean value, which controls whether the following is parsed as a *type* or as a string when explicit quotes are present.

```
((type-or-string type) :string-if-quoted string-if-quoted)
```

The default, if **:string-if-quoted** is not supplied, or is **nil**, is to always try to parse it first as a type and then as a string if that fails. However, if **:string-if-quoted** is **t**, then explicit quoting (with doublequotes) will force the object to be parsed as a string.

Here is an example of **:string-if-quoted**:

```
;;; the following treats "3" as an integer
(accept '(type-or-string integer))
```

```
;;; the following treats "3" as a string
(accept '((type-or-string integer) :string-if-quoted t)
```

- **reduce** now accepts the **:key** keyword which enables you to extract the values to reduce. The **:key** function will be applied exactly once to each element of the se-

quence in the order implied by the reduction order, but not to the value of the **:initial-value** argument, if any.

- Support for the **conditions:abort** restart, part of the compatibility support for the new Common Lisp condition system, was faulty in that pressing `c-ABORT` did not return you to Common Lisp restarts named **conditions:abort**, nor did the debugger offer to take you to such a restart if you were to press `ABORT`. In Genera 8.1, these two problems have been corrected.
- In a prior release, when **clt:truename** was called on an open stream, it coerced that stream to a pathname and then looked for it in the directory — finding an older file or none at all. In this release, it has been fixed to correctly return the name of the file to which the stream is really open.
- `|3600|` is now equivalent to `3600` after the `#+` or `#-` reader macros.

Changes to the User Interface in Genera 8.1

Changes in the Global Command Table in Genera 8.1

The commands in the Global command table have been reorganized in a hierarchical fashion into a number of subtables. The Global command table continues to inherit from these other command tables, so there are no user-visible effect of the change. However, the additional structure imposed has several good effects:

- It enables people to write tools which browse the set of commands in an organized way.
- It allows users to more easily customize the set of operations they want, by allowing them to add/remove not only individual commands, but whole classes of commands.
- It encourages discussion about how command sets are grouped and organized. The current command set is so large and undifferentiated, that it is difficult to take inventory of the whole thing.

In the new scheme, the intent is that no one should ever add commands to "Global" (although no harm will occur if they do). The main reason for not adding something to Global is only to force at least one level of categorization on the tree of commands, breaking it up at least somewhat. When in doubt, the new "Utilities" command table is a better catch-all.

Users and system maintainers who want to add just one or two commands should find an appropriate command table and add it/them there. If there are several related commands in a new domain, it's appropriate to make a separate command table for that purpose.

Users who add custom commands of their own (e.g., in their init file), but for which there is no suitable command table, can add them to "User" avoid using "Global". (This has always been true, but not widely publicized.)

It might be appropriate for a command to appear in more than one command table. Right now, none do, but that is only an artifact of the initial break-up. In the future, certain commands might appear in more than one place.

Global

Communication - Communication with other users.

Conversation - Interactive communication (converse messages)

Mail Reading/Sending - Non-interactive communication (mail messages)

Demonstration - Tools for demonstrating the system to others

Documentation - Tools for finding out about the system

Editing - General commands related to editing

File System - Manipulating the file system

Directory - Commands operating on directories

File - Commands operating on files

FEP - FEP file commands

NFS - NFS file commands

Stattice - Stattice file commands

Printer - User interface to submitting hardcopy requests and checking queue

Programming Tools - Commands related to using the LispM for programming

Debugging - Commands specifically related to debugging programs

Breakpoint - Setting and checking breakpoints

Presentation - Debugging the presentation system

Process - Manipulating processes

Timer Queue - Commands related to Timer Queues (this will probably go away soon)

Tracing - Trace, trace-conditions, etc.

Lisp - Lisp-related commands

CLOS - CLOS-related commands

Evaluation Context - Manipulating evaluation environment (package, base, ...)

Flavors - Flavors-related commands

Inspection - Commands for inspecting the environment

Callers - The 'Who Calls' facility

System Maintenance - SCT-related commands

Tape Administration - Tape tools (distribution and carry)

Session - Commands that only affect the current session

Activities - Manipulating activities

Garbage Collection - Manipulating the GC

Networks - Manipulating the networks

Window - Manipulating windows

Site Administration - Tools for site administrators

Access Control - Access control lists

Mailer - Tools for controlling the mailer

Namespace - Tools for manipulating the namespace

Printer Maintenance - Tools for starting/stopping print spooler, etc.

Spelling - Tools for manipulating the Spell dictionary

- World Building** - Tools for creating and installing worlds
- Utilities** - Interesting things with no other obvious home
- Document Formatting** - Tools for document processing
- Fonts** - Tools for manipulating fonts
- Images** - Tools for manipulating images

Changes to the Command Processor in Genera 8.1

- Genera 8.1 includes a convenient and extensible demonstrations facility. See the section "The Demonstrations Facility".
- The CP command Show Directory has been changed slightly to make certain common cases of large directory listings faster.
- The CP command Show System Plan now longer gets an error if you specify :Machine Types All.
- The CP command Show Object now enables you to specify a double-colon for a symbol name that is an internal symbol. Thus you can now enter colon combinations that are consistent with what the Lisp reader would expect. Within a restricted context (such as CLtL-Only), you can use the triple colon notation to escape.
- The CP command Show Keyboard Layout now displays additional IBM PC keyboard layouts.
- For more user interface uniformity, Clean File and Clean Directory are now available as both CP and Zmacs commands (that is, the Clean Directory CP command, and the (M-X) Clean File Zmacs command have been added).
- The pathnames which appear as output of the Clean File CP command are now mouse sensitive.
- In a previous release, the Boot Machine command would call the Logout command implicitly. In some cases, the Logout command would ask if you were really sure you wanted to Logout. If you answered No, it would suppress the Logout action, but not cancel the rest of the Boot Machine action. The code modularity has been improved so that the behavior is now more predictable.
- The CP command Show FEP Directory previously did not provide a way to list just those files that were *not* one of the defined FEP file types. A new option to the :Type keyword, Other, has been added to do this.
- A bug affecting 3600-family users was fixed, in which the CP command Show FEP Directory, when used to show the FEP directory on the local host, opened access paths to nonexistent FEP units. In Genera 8.1, Show Fep Directory works correctly on 3600-family machines.

- The meaning of the `:Detailed` argument to the Show Herald CP command has changed slightly. Previously, the possible values were Yes and No. In Genera 8.1, the possible values are Yes, Normal, and No. The new value Normal means what No used to mean (to display the systems which are usually of interest to users). In Genera 8.1, No shows only machine information and release level, and no information about system versions. The default if this argument is not supplied has been changed from No to Normal, so that the effect of using Show Herald with no keyword arguments is the same as it used to be.
- More parts of the Show Herald output are mouse-sensitive, such as the world name, memory usage, machine and site names; consult the mouse documentation line.
- The CP command Show Herald has been made faster.

The `:Name` argument to the CP command Show Processes has been expanded to enable you to specify an existing process name, a process name substring, or a combination of the two. If you specify substrings only, the processes named by those substrings are considered for viewing (subject to the other options). If you specify process names only, only those processes are considered. If you specify a mix, then processes named exactly and processes named by substrings are both considered. When specifying substrings, it is recommended that you use string quotes to avoid unintentional completion.

- The Show Directory CP command has a new keyword, `:Excluding`:

`:Excluding` `{file1, file2 ...}` Exclude files that do match the indicated wildcard filenames from the directory listing. Logical pathnames and pathnames having versions specifiers of "oldest" and "newest" are not permitted as excluded *files* because the exclusion test will be done against the truename of the physical pathname, and such pathnames would never match.

- The Show Notifications CP command also has an `:Excluding` keyword. It accepts a string or a sequence of strings and does not show notifications that contain that string or strings.
- A new keyword, `:More Processing`, has been added to *all* commands which use `:Output Destination`. The purpose of the new keyword is to provide control over ****More**** processing for commands that do substantial amounts of non-interactive output. Since the criterion for when to include this keyword is very similar to the criterion for when to include the `:Output Destination` keyword, we chose to make both of these options be controlled by the **`:provide-output-destination-keyword`** option to **`cp:define-command`**, rather than making a new option which would need to be supplied in nearly all cases where **`:provide-output-destination-keyword nil`** is now provided.

: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 Command Processor Commands in Genera 8.1

There is a new Command Processor command for remote debugging:

Monitor Screen Command

Monitor Screen *host*

Allows viewing the actual contents of some other lisp machine screen. It is useful for observing the state of file servers from afar, and for helping debug someone else's machine when it is not practical to get to the machine itself.

host The name of a cooperating host. **Note:** The person whose screen is being monitored must explicitly turn on the server to allow someone else to see their screen.

Monitoring is turned off by default. The person whose screen is being monitored must explicitly turn on the server to allow someone else to see their screen. They can choose to disallow remote viewing, allow remote viewing with a warning notification, or allow remote viewing with no warning. They may also choose to make the operation of the server visible in the wholine, or to make it invisible. (The latter is useful for watching file servers, when it is important to see something in the wholine area such as open files, or some other progress indication that appears there. See also the "Show Open Files Command".)

If the machine is not logged in, you can always monitor its screen. If the machine is logged in, however, monitoring is disallowed by default. You enable monitoring with the function **chaos:enable-monitor-screen-server**.

chaos:enable-monitor-screen-server &key (:server-enabled :no) (:wholine-appearance :visible)
Function

Allows someone else to monitor your screen using the Monitor Screen CP command. Calling this function without any arguments disables monitoring your screen (the default). Note that, if your screen is already being monitored, the current monitoring session will *not* be terminated.

:server-enabled One of **:yes**, **:no**, or **:notify**. The default is **:no**. **:no** means no one can monitor your screen. **:yes** means anyone can monitor your screen, and you will not see a notification that they are

doing so. **:notify** means that anyone can monitor your screen, but you will see a notification first when they start.

:wholine-appearance

One of **:visible** or **:invisible**. The default is **:visible**. When **:server-enabled** is **:yes** or **:notify**, **:visible** means that the wholine area will display the fact that the server is operating if someone is monitoring your screen. **:invisible** means that the wholine area will not note whether anyone is monitoring your screen.

Note that the server side of this capability has existed for many releases, though no easily-accessible user side existed. Even then, the default for the server was to disallow access.

Site administrators, and others who build worlds, should *not* enable this facility in their world-building scripts, since allowing arbitrary monitoring of others' screens is generally considered a serious invasion of privacy. Users may wish to ensure that the worlds they run do not have this turned on by default.

This is a Chaos-only protocol, meaning that only Chaosnet users can monitor screens. This means that sites which communicate with the outside world via TCP/IP, rather than Chaos (that is essentially all sites) do not have to worry about users elsewhere on the Internet monitoring any screens at their site, regardless of whether individual users enable monitoring.

Many of these capabilities are not necessary if the machine(s) of interest run some remote window system, such as the X Window System.

Screen monitoring only works if both machines are *not* embedded machines (for example, they are *not* UX-family or MacIvories). Both machines must have screens which are only one bit deep, that is, color or gray scale screens cannot be used.

To monitor a screen, you do something like the following:

```
Monitor Screen (of cooperating host [default CARDINAL]) NUTHATCH
```

To stop monitoring a screen, type any character.

There is a new Command Processor command for getting information about memory errors:

Show Memory Error Corrections Command

Note: This command is implemented only on Ivory machines.

Show Memory Error Corrections *keywords*

With no keyword arguments, displays the number of memory errors that have occurred.

keywords: **:Detailed**, **:More Processing**, **:Newest**, **:Output Destination**

- :Detailed {Yes, No} List the logged error correction codes. 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").
- :Newest {*a number*} The number of items to be output. The default is 20.
- :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***.

With the :Detailed keyword, displays the newest 20 error corrections, for example:

```
There have been 38 memory error corrections.
The newest 20 error corrections are listed.
The most recent memory error correction is listed first.
```

<i>Memory Address</i>	<i>Check Bit</i>	<i>ECC Syndrome</i>	<i>Disposition</i>
2040256	11	145	Soft (corrected) error, physical access
...	Soft (corrected) error(s), unlogged
1532420	11	145	Soft (corrected) error, virtual access
...	Soft (corrected) error(s), unlogged
5646204	11	145	Soft (corrected) error, virtual access
.	.	.	.
.	.	.	.
.	.	.	.

For each error listed, it also lists the kind of error and its disposition. The kind of error is either Uncorrectable error or Soft (corrected) error.

(The former appears only if you successfully used Continue in the FEP when the uncorrectable error occurred).

The disposition is one of:

repaired by scrubber

meaning there was an error that could be fixed by rewriting the data.

stuck (*call field service*)

meaning there is an error that cannot be fixed by rewriting, but if it is a soft error, the correct data is being returned. However, it cannot be fixed to give no errors; hence you probably have a stuck bit in your memory.

ignored (corrected data written to disk)	meaning there was an error that was not worth fixing (the good data went to the disk already).
virtual access	meaning an error that has not (yet) been fixed in virtual memory by the error scrubber.
physical access	meaning an error that has not (yet) been fixed in physical memory by the error scrubber.
unknown access	meaning either a virtual or a physical access error, but we cannot figure out which. Most likely this is due to an error that occurred in the FEP.

There is a new Command Processor command for showing the table of contents of a documentation topic:

Show Table of Contents Command

Show Table of Contents *topic keywords*

Shows the table of contents of the documentation *topic*.

<i>topic</i>	A topic which is documented and accessible in Document Examiner.
<i>keywords</i>	:Crossreferences, :Depth, :More Processing, :Output Destination, :Sources
:Crossreferences	{Yes, No} Whether to display crossreferenced topics. The default is No.
:Depth	{ <i>integer</i> } How many levels deep to display. The default is to show all levels.
: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* .
:Sources	{Yes, No} Whether to display associated source files. The default is No. Note that in a world with a compressed documentation database, the source file information is not available.

There is a new Command Processor command for showing the structure of a command table:

Show Command Table Command

Show Command Table *command-table keywords*

Shows the structure of a command table.

command-table

keywords :Locally, :More Processing, :Multiple Columns, :Output Destination, :Show Commands

:Locally {Yes, No} Whether to suppress recursive display of information about inherited command tables. The default is No, and 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").

:Multiple Columns {Yes, No, *integer*} Whether to use multiple columns in the display. The default is No, and the mentioned default is Yes. An integer number of columns is also permitted. This is especially useful for command tables such as Global which inherit from a large number of other command tables.

: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.

:Show Commands {Yes, No} Whether to show the commands in the command table. The default is Yes. If No is specified, then only command table names are shown, and but not the individual commands.

The name of the command table is presented with descriptive information presented indented beneath it. Names shown in bold are command tables from which the command table inherits. Names shown in italics are commands in that command table. For example:

Show Command Table (named) News

News

Monitor AP Newswire

Sports

Show Scores

Record Bet

Weather

Show Weather Report

Show Wind Chill Table

This means that the command table News has one command of its own, Monitor AP Newswire, and inherits from two other command tables, Sports and Weather, each of which has two commands of its own.

T

New FUNCTION Key to Control End of Page Action

New FUNCTION key assignments FUNCTION E and FUNCTION c-E have been added to control what happens at the end of a screen. The possibilities are as follows:

FUNCTION E Sets the global default end of page mode. With no argument, it prompts with a menu for a new end of page mode. If an argument is supplied, it selects one of these possible modes:

<i>Argument</i>	<i>Value and Meaning</i>
1	Scroll. Scroll text on the viewport upward to show the next line.
2	Truncate. Continue with output beyond the end of page, but do not automatically scroll up.
3	Wrap. Continue with output from the top of the viewport, without moving text already displayed on the viewport.

FUNCTION c-E Sets the end of page mode for the selected window. With no argument, it prompts with a menu for a new end of page mode. If an argument is supplied, it selects one of these possible modes:

<i>Argument</i>	<i>Value and Meaning</i>
0	Default. When an end of page occurs, use whatever action is globally selected at that time. The global action is controlled by FUNCTION E or Set Screen Options.
1	Scroll. Scroll text on the viewport upward to show the next line.

- | | |
|---|-----------------------------------------------------------------------------------------------------------------|
| 2 | Truncate. Continue with output beyond the end of page, but do not automatically scroll up. |
| 3 | Wrap. Continue with output from the top of the viewport, without moving text already displayed on the viewport. |

If the window does not support an end of page mode, FUNCTION c-E just beeps.

Miscellaneous Improvements to the User Interface in Genera 8.1

- Genera 8.1 now protects users at mixed-architecture sites from copying a .ilod file to a 3600-family machine, or copying a .load file to an Ivory machine. The Copy World command now warns you if you try to copy the wrong type of world file for your machine.
- In Genera 8.0 (and before), if you used Show Directory to show a directory on a host which had no namespace information (for example, by naming the host using the "CHAOS|nnnnn" notation) and then waved the mouse over a subdirectory name which appeared in the resulting output, you were sometimes thrown into the debugger. In Genera 8.1, mouse motion over a directory name on a host of unknown type will no longer cause entry to the debugger.
- Genera 8.1 fixes a bug in the menu interface to reordering sequences (used in Symbolics Concordia's Reorder Records command, for example) which caused the menu to pop up on the wrong console, for example, when multiple X consoles were in use to the same machine.
- The "Quantum" field displayed by the Monitor System Status CP command was not really useful, and has been removed to conserve space in the overall presentation of process information. This change has the beneficial side-effect of reducing the likelihood of line wraparound problems on remote terminals.
- The output of **step** has been improved. At deep levels of recursion, the indentation of the **step** output is reset to column 0, so the output will be more readable to the user, instead of running into the right margin of the screen. The variable **si:*step-indentation-restart-fraction*** controls when the indentation is set back to 0; its value is a non-zero fraction of the screen size after which the stepper should go back to column 0 for its indentation, or **nil** to prevent the stepper from ever resetting to column 0.
- Some bugs have been fixed in FSM Search that prevented it from finding some matches it should have found for complex patterns involving <or>, and that caused it to signal an array out of bounds error when the search pattern was

too complicated. FSM Search is accessible through Extended String Search and Tags Search in Zmacs, Find String in Zmail, and numerous similar commands that prompt "(Extended search characters)" and is used internally in the Conversion Tools.

- A bug has been fixed in which the CP commands Show Timer Queue and Delete Timer Queue Entry and the underlying (**si:print-timer-queue**) get an error if nothing has ever added an old-style timer queue entry (because **si:*timer-queue*** is unbound in that case). The two CP commands remain in the system, but are normally inaccessible from the CP, because they are not in any command table. These commands are a remnant of the previous timer queue implementation, which has been replaced.
- The method for computing the presentation subtype relationship between two **cp:command-name** and two **cp:command** presentation types used to ignore the **:command-table** argument but no longer does. A presentation type (CP:COMMAND-NAME :COMMAND-TABLE *a*) is now correctly considered a presentation subtype of a presentation type (CP:COMMAND-NAME :COMMAND-TABLE *b*) if either *a* and *b* are the same command table or *b* inherits from *a*. (Note also that if the **:command-table** argument is unsupplied, it defaults to **cp:*command-table***, so the same rule can be applied to the atomic cases of these presentation type specifiers.)
- The presentation type **token-or-type** now interactively describes its tokens as "special tokens" rather than "tokens".
- There is a variable to control the message explaining the ellipsis in Help displays:

dw:*display-ellipsis-help*

Variable

Controls the presentation of a help message explaining the meaning of a notation such as "Foo ... (7)" to indicate 7 possibilities beginning with "Foo". By default, the value is **t** which means that this help message continues to be presented every time such a notation is used until the first time such an item is clicked on, at which point the value becomes **nil** and the message is no longer presented. The value may be set to **nil** before this in order to suppress the message at an earlier point (in an init file, for example) or it may be set to **:always** in order to keep it from ever becoming **nil**, regardless of whether such an item is clicked on or not.

- A bug in **dw:accept-variable-values** which caused a default not to be provided when the variable's current value was **nil** has been fixed. A number of programs which use this utility will see an improvement due to this fix.
- The initial window and the Show Herald command now display the size of physical memory and swapping space in megawords rather than kilowords.

- Genera 8.1 improves the CP command Help. The Help command now has new options and also correctly compresses command tables.

Help command Displays a list of Command Processor commands.

keyboard Specifies the *keyboard* help category.

commands Specifies the *commands* help category.

keywords :Format, :Command Table

:Format {Brief, Full, Detailed} The level of detail to show in the list. The default is Brief. This is the same as pressing the HELP key.

Brief means that only unique command names are shown in full; for the rest, the verb is shown with the number of commands that start with that verb.

Full shows all commands in abbreviated form.

Detailed displays the entire list of available commands.

:Command Table The command table to show.

- Genera 8.1 improves the appearance of the pop-up debugger menu. The new variable **dbg:*menu-proceed-style*** controls the presentation style used for the pop-up menu:

:simple Shows only the proceed options Resume, Abort, Suspend, c-M.

:novice Shows the same as **:simple**, but also shows debugger special commands (for example, when there's a non-existent directory, the option to create the directory will appear.)

:expert Shows all proceed options, restarts, and special commands. This is the default.

:simple-icon Like **:simple**, but uses a more simple presentation format.

:novice-icon Like **:novice**, but uses a more simple presentation format.

:expert-icon	Like :expert , but uses a simpler presentation format.
:classic	For those who want to continue to use the existing presentation format.

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.

<i>name</i>	The name of a demonstration.
<i>keywords</i>	: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** 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**.

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))

```

Changes to User Interface Programming in Genera 8.1

The **sequence** presentation type has a new presentation argument:

:element-default *object*

Specifies that *object* is to be used as the default when doing a recursive **accept** of the first element of the sequence. This is different than specifying the default for the sequence, since you might want the sequence default to be empty, and yet you might want to specify that if the user decides to type an element, that element should be parsed against a particular default. For example:

```

(accept '((sequence pathname)) :default '())
Enter pathnames of files: tennis
;completes to "ACME:tennis"
=> (#P"ACME:tennis")
((SEQUENCE-ENUMERATED FS:LMFS-PATHNAME))

(accept '((sequence pathname)
         :element-default #P"S:>Joe>bowling.text")
         :default '())
Enter pathnames of files: golf
;completes to "ACME:>Joe>golf.text.newest"
=> (#P"ACME:>Joe>golf.text.newest")
((SEQUENCE-ENUMERATED FS:LMFS-PATHNAME))

```

The **sequence-enumerated** presentation type has a new presentation argument:

:element-defaults '(*object1 object2 ...*)

Specifies an enumerated sequence of *objects* to be used one by one as the default for each recursive **accept** involved in accepting elements of the enumerated sequence. This is different than specifying the default for the sequence, since you might want the sequence default to be empty, and yet you might want to specify that *if* the user decides to type an element, that element should be parsed against a particular default. For example:

```
(accept '((sequence-enumerated pathname pathname)
         :element-defaults (#P"ACME:>JDoe>file.text"
                           #P"ACME:>Fred>file.text"))
        :provide-default nil)
```

```
Enter the pathnames of two files: a, b
;completes to "ACME:>JDoe>a.text.newest, ACME:>Fred>b.text.newest"
=> (#P"ACME:>JDoe>a.text.newest" #P"ACME:>Fred>b.text.newest")
((SEQUENCE-ENUMERATED FS:LMFS-PATHNAME FS:LMFS-PATHNAME))
```

Changes to Networks in Genera 8.1

Remote Login in Genera 8.1

- **neti:enable-serial-terminal** now takes two additional keywords, **:status-line-p** and **:status-line-update-frequency**. These allow you to specify that the remote terminal has a status line at startup time. The argument to **:status-line-update-frequency** is in sixtieths of a second, default 300 (5 seconds).
- Genera 8.1 fixes some bugs in the status line for remote terminals. Previously the status line did not handle incremental redisplay well, occasionally left blots on the status line, and displayed strange-looking progress notes.
- When you login to a Symbolics host remotely, the display has been changed from the full herald to a brief herald containing the information as in this example:

```
Symbolics System, MAMA-CASS|FEP0:>Genera-8-1.load.1
3640 Processor, 2048K words Physical memory, 29396K words Swapping space.
Genera 8.1
Symbolics Junco
```

You can use the Show Herald CP command if you need the full herald.

Set Remote Terminal Options Command

Set Remote Terminal Options

Prompts for the options to set up a remote terminal. On a serial terminal, keyword arguments to **neti:enable-serial-terminal** are used to determine reasonable defaults.

Set Remote Terminal Options is only available from a remote terminal.

Changes to NFS in Genera 8.1

- Genera 8.1 fixes a bug in NFS related to listing files in a directory. Because NFS has no way to list just one file in a directory, you have to list the whole directory each time. When saving a file in ZMACS, NFS would read the entire directory to find the newest version of the file, save the file and then read the entire directory again to see if excess versions had to be deleted. A caching scheme makes this operation faster.
- The NFS LMFS longname problem has been fixed. NFS was using only the directory-entry portion of a potentially much longer LMFS filename, and therefore would get file-not-found errors when it should not have.
- Some NFS servers (not Sun and Ultrix) object to the root directory mounting string of "/" that is provided by **fap-nfs-mount-string-for-host**, and prefer "/" instead. The NFS Client has been changed accordingly, giving it broader compatibility.

Miscellaneous Improvements to Networks in Genera 8.1

- Previously, the Network RPC system, which implements RPC services over TCP and UDP agents, plus a fair amount of substrate for using RPC to UNIX hosts, was bundled with the NFS product. Genera 8.1 makes these useful RPC capabilities more widely available by making Network RPC a component of RPC. Network RPC is autoloaded when IP-TCP is loaded. Network RPC is no longer a stand-alone system.
- Genera 8.1 implements the Van Jacobson TCP performance modifications, including slow-start and congestion avoidance. This improves the performance on slow links.
- Directory handling has been made more robust, for TCP-FTP to a UNIX host. The directory output now correctly handles pathnames and link targets with spaces in them, and pathnames with the token "->" in them
- Genera 8.1 improves the reliability of IP gateways after warm-booting.
- IP routing initialization reliability is improved. IP packets are no longer accepted until routing initialization is complete.
- Genera 8.1 fixes a bug in which if you were not using the main console (for example, you were connected via an X Window) and you used the Namespace Editor commands Edit Namespace Object or Create Namespace Object, the namespace editor would expose on the main console and not the remote console.
- Genera 8.1 includes an improvement to UDP. Previously, each UDP broadcast would start a new server process because UDP doesn't allow "connections" to be made for broadcasts. Now UDP accepts "broadcast connections" instead of making multiple processes.

- IP-TCP performance has been improved on Ivory-based machines (those with Rev4 Ivory chips or higher), by using Ivory subprimitives for checksumming.
- The serial protocol used on embedded machines (MacIvories and UX machines) now implements send-break.
- Performance of the Ethernet interface on embedded machines has been improved.

Changes to Garbage Collection in Genera 8.1

New Function: `si:ephemeral-gc-flip-area`

si:ephemeral-gc-flip-area *area* &key *:all-levels* (*:mode* **si:*ephemeral-migration-mode***) *Function*

Immediately performs an ephemeral garbage collection of the specified area.

The **:all-levels** keyword controls which levels to collect. **t** means all levels of the area should be collected. **nil** means just the first level. **nil** is the default.

The **:mode** keyword allows the user to specify the migration mode. Possible values are **:normal**, **:dynamic**, **:keep**, **:collect**, and **:extra**. The default is the current dynamic value of **si:*ephemeral-migration-mode***.

This function is rarely needed by most users.

Miscellaneous Changes to Garbage Collection in Genera 8.1

Genera 8.1 fixes a bug in using a form such as the following to increase the number of ephemeral levels:

```
(make-area :name 'sys:working-storage-area :n-levels 4)
```

Prior to this bug-fix, the newly created levels never had any objects created in them.

When gc flips are inhibited, the notification you get complaining that things are bogged down did not tell you what process is responsible. In Genera 8.1, the notification does tell you which process is slowing things down.

A bug in the ephemeral garbage collector on Ivory machines has been fixed. This bug could cause an Ivory machine to halt with the error "pointer to unallocated level".

Changes to Conversion Tools in Genera 8.1

New Conversion Tools in Genera 8.1

Genera 8.1 offers two new conversion tools:

- 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".

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.

Miscellaneous Changes to Conversion Tools in Genera 8.1

In Genera 8.0, the conversion tools were not converting **sys:print-self** methods to **clos:print-object** methods. They only converted **:print-self** methods. In Genera 8.1, **sys:print-self** methods are treated the same as **:print-self** methods and are converted properly.

Changes to the FEP in Genera 8.1

For information about other changes to the FEP, see the section "Changes to the FEP in Genera 8.0.2".

Change to FEP-Tape Activity in Genera 8.1

Previously, when you used FEP-Tape to restore kernel and flods to the FEP file system of an Ivory machine, it did not install the kernel. In Genera 8.1, the Read Tape command of the FEP-Tape activity automatically installs any newly restored installable files (FEP kernel, color sync file) at the earliest time that is prudent.

Note that the other two methods of installing new flods in a FEP file system (Copy Flod Files and restoration from the IFEP) already installed the FEP kernel

automatically, so this change makes the FEP-Tape activity consistent with those methods.

New FEP Feature: Using a Serial Terminal to Communicate with the FEP

This new FEP feature was made available in Genera 8.0.2 and is therefore also available in Genera 8.1. However, this feature only applies to Ivory machines, and excludes Ivory embedded machines (i.e. XL400, XL1200).

Using a Serial Terminal to Communicate with the FEP

You can use a serial terminal to communicate with the FEP. One case in which this can be particularly useful is in troubleshooting an XL1200 single-monitor color station. For example, if the color monitor cannot come up, you can connect a serial terminal to the serial port on the color console unit, and use that terminal to give FEP commands such as Show Disk Label, or Set FEP Options.

In addition to connecting the serial terminal physically, you need to give the Set Console FEP command to tell the FEP to use the serial console. You might need to also use the Set Monitor Type FEP command to tell the FEP whether the serial console is ASCII (a dumb terminal) or X3.64 (such as a VT100).

You need to set the serial terminal's parameters for 9600 baud, 8 bits, and no parity.

Keep in mind that serial terminals don't have all the special keys of the Symbolics keyboard. If you need to transmit special Symbolics characters, such as Meta or Super characters, you need to understand how serial terminal keys are mapped to the Symbolics keys.

Mapping of Serial Terminal Keys to Symbolics Keys

The keyboard of a serial terminal does not have the same set of keys as does the Symbolics keyboard. For example, such a keyboard typically lacks a Meta key, Super key, Hyper key, and Symbol key. These keyboards do, however, have a Control key, however, most such keyboards can handle only Control-A, Control-Z, and a few other characters, most of which are reserved for escapes.

Accessing the Symbolics Character Set

The following characters may be used to access the Symbolics character set:

```
c-^ = Toggles the Control bit    c-] = Toggles the Super bit
ESC = Toggles the Meta bit      c-\ = Toggles the Hyper bit
c-@ = Toggles the Shift bit
```

For example, to enter `c-m-C`, you need to set both the Control bit and the Meta bit, by entering `c-^` and `ESC`; you can then press `C` to enter `c-m-C`.

Similarly, you might need to enter `c-sh-C`. The serial keyboard has both a Control key and a Shift key, but you cannot press them both at once to enter `c-sh-C`. You can enter `c-^` to set the Control bit, then press the Shift key while typing C. Or, you can enter `c-@` to set the Shift bit, and then press the Control key while typing C.

Entering Special Symbolics Keys

The character `c-_` (that is, the Control key and the underscore `_` key) is used as a prefix to enter special characters as follows:

H = <Help>	L = <Line>
E = <End>	P = <Page>
A = <Abort>	F = <Refresh>
S = <Suspend>	B = <Back-Space>
R = <Resume>	N = <Network>
C = <Complete>	1 = <Square>
I = <Clear-Input>	2 = <Circle>
X = <Escape>	3 = <Triangle>

For example, if you press `c-_` followed by H (that is, two keystrokes) on the keyboard of a serial terminal, you get the effect of the HELP key on a Symbolics keyboard.

Entering Symbol Characters

`c-_ _` is the prefix for Symbol characters. (That is, Control Underscore followed by Underscore, two keystrokes.)

For example, you can enter `c-_ _ P` (that is, three keystrokes) to get the effect of SYMBOL-P.

`c-_ ?` displays the `c-_` dispatch table.

Saving Previous FEP Kernels and FLOD Files

Some users like to "houseclean" their FEP-related files, and delete all but the most recent version. This is a dangerous habit, and we recommend against it. Backup versions of FEP files are necessary in some debugging situations. Since FEP files (both the kernel and flod files) do not require much disk space, we recommend that you save the previous versions of these files.

Ivory users in particular should save the previous version of the FEP kernel and flod files. Note that on Ivory machines, there is no FEP in PROM as there is on 3600-family machines, so your options are very limited if you depend on a single copy on disk and it goes bad.

The system attempts to prevent you from deleting the previous kernel, but it does not keep you from deleting the previous flods. The kernel is not much use without its flod files.

Test Memory FEP Commands Now Available on Ivory

Four FEP commands that in previous releases were 3600-family commands only, now work on Ivory machines:

Test All FEP Command

Test All

Runs all FEP tests on a 3600-family machine. On an Ivory machine, it runs only Test Main Memory. For information about each FEP test:

See the section "Test Main Memory FEP Command".

See the section "Test Simple Main Memory FEP Command".

See the section "Test A Memory FEP Command".

See the section "Test Disks FEP Command".

You can load the Test All FEP command by scanning the overlay (flod) file `*-tests.flod`.

Test Location FEP Command

Test location on a 3600-family machine:

Test Location

Tests a single location in main memory.

On an Ivory machine:

Test Location *address keywords*

address {*octal-number*} The location to test, in octal.

keywords :Base, :Hard ECC Error Action, :Passes, :Test Pattern

:Base {2, 8, 10, 16} The base in which to display the errors. The default is 8.

:Hard ECC Error Action {Halt, Report, Clear} The action to take when a hard ECC error is encountered. The default is halt, so that the error can be fixed. Report means just report the error. Clear means clear the location. Clearing the location can damage a running world. This action should be used with caution.

:Passes {*integer*} The number of times to run each test pattern over the location.

:Test Pattern {Quick, Checkerboard, Walking-bit, All} The pattern to use to test. The default is Checkerboard. All uses each pattern in turn.

You can load the Test Location FEP command by scanning the overlay (fload) file `*-tests.fload`.

Test Main Memory FEP Command

Test main memory on a 3600-family machine:

Test Main Memory

Tests some locations in main memory. It runs more slowly (and less thoroughly) than Test Simple Main Memory.

See the section "Test Simple Main Memory FEP Command".

On an Ivory machine:

Test Main Memory *keywords*

<i>keywords</i>	:Base, :Hard ECC Error Action, :Passes, :Test Pattern
:Base	{2, 8, 10, 16} The base in which to display the errors. The default is 8.
:Hard ECC Error Action	{Halt, Report, Clear} The action to take when a hard ECC error is encountered. The default is halt, so that the error can be fixed. Report means just report the error. Clear means clear the location. Clearing the location can damage a running world. This action should be used with caution.
:Passes	{ <i>integer</i> } The number of times to run each test pattern over the memory.
:Start	{ <i>location</i> } The address at which to start the test. There is a lower limit of 1011015 (octal) to protect the FEP itself, which on an Ivory machine resides in main memory.
:Test Pattern	{Quick, Checkerboard, Walking-bit} The pattern to use to test. The default is Checkerboard.

You can load the Test Main Memory FEP command by scanning the overlay (fload) file `*-tests.fload`.

Test Simple Main Memory FEP Command

Test simple main memory on 3600-family and Ivory machines:

Test Simple Main Memory

Runs a memory test. It runs faster (and more thoroughly) than Test Main Memory.

See the section "Test Main Memory FEP Command".

Load the Test Simple Main Memory FEP command by scanning the overlay (fload) file *-tests.flod.

Changes to Boot Files for Ivory Machines

For Genera 8.1, we recommend a change in how boot files for Ivory machines are set up.

```
Hello Innn
Hello Local (or hostname)
```

Innn and *Local* represent two .boot files. Their contents should be as follows:

Hello *Innn* Boot File

The *Innn*.boot file (where *nnn* is the IFep version number, which is 328 for Genera 8.3) should contain the commands to scan the flod files and initialize things.

```
Scan I328-lisp.flod
Scan I328-loaders.flod
Scan I328-info.flod
Scan I328-debug.flod
Initialize Hardware Tables
```

Hello *Local* Boot File

The *local*.boot file should contain those commands that set up this specific machine, declaring paging files, setting the network address, and any other boot options.

```
Declare Paging Files FEP0:>Paging-1.page
Declare More Paging Files FEP0:>Paging-2.page,Paging-3.page
Set Boot Options :Network Address Chaos|52525 :IDS Enable
```

Here is the recommended sequence of commands for a Symbolics Ivory-based machine that is cold booting a world from the local disk:

```
Load World
Start
```

This loads the most recent world on your local disk, which is usually the one you want to boot.

For a netbooted machine, the recommended sequence is:

```
Netboot inc-site-genera-8-3
Start
```

If you update to Genera 8.1 using Disk Restore, these boot files get created for you.

Note: Copy World and Copy Flod Files do not yet understand this form for boot files. Therefore, if you use Copy World or Copy Flod Files, do not accept the option to have them update your boot files. Update your boot files by hand.

Miscellaneous Changes to the Fep in Genera 8.1

- There was a bug in the NFEP overlays shipped with Genera 8.0.1 which caused bus errors when the Clear Machine FEP command was issued on some machines. Regrettably, the bug is extremely elusive and could not be diagnosed in time for Genera 8.1. In order to correct this, we are reverting to the NFEP overlay versions which were shipped with Genera 8.0, which did not exhibit this problem. To keep NFEP overlay installation simple, we have increased the version numbers on the Genera 8.0 overlays before releasing them for Genera 8.1. Other than the changed version numbers, the Genera 8.1 NFEP overlays are identical to the Genera 8.0 overlays.
- A bug was fixed that affects 3600-family users, in which the Show Fep Directory CP command, when used to show the FEP directory on the local host, opened access paths to nonexistent FEP units. In Genera 8.1, Show Fep Directory works correctly on 3600-family machines.
- The CP command Show Fep Directory previously did not provide a way to list just those files that were not one of the defined FEP file types. A new option to the **:Type** keyword `Other` has been added to do this.
- The Netboot Fep command is now offered on Ivory machines.

Changes to Streams in Genera 8.1

- Genera 8.1 fixes a bug in which serial streams did not signal end of file if asked for input after they were closed. This bug manifested itself as remote terminal input processes hanging around doing operations on closed streams after the terminal had been disabled.
- Another serial bug was fixed, in which a machine could hang when closing embedded serial channels.
- Genera 8.1 fixes a bug in pipe streams; when you tried to close both sides of a pipe stream cleanly, the second one closed got an error. This bug could cause the X server to crash.

Changes to the Window System in Genera 8.1

- Genera 8.1 fixes a bug in multiple line drawing and multiple rectangle drawing on 3600-family machines. This bug was introduced in Genera 8.0.1, so it affected only those users who received and installed either Genera 8.0.1 or 8.0.2.
- In Genera 8.1, it is now easier to mark text in a Dynamic Window that is bigger than the size of the screen. The marking and yanking menu (which you can get to by clicking Right in a Dynamic Window) has a new command **Extend Marked Text**, which extends the nearest piece of marked text to include all the text between it and the cursor.
- A new variable, **tv:*who-line-function-hook***, allows users to insert their own information into the status line. This variable can be set to a symbol with a function definition to display things in the progress note area of the status line.

Changes to Zmail in Genera 8.1

- Genera 8.1 fixes the interaction between the Logout CP command and expunge queries in Zmail. Specifically, if your Zmail was set to query before expunging mail files, and you gave the Logout CP command, answered "Yes" to a query to expunge and save a Zmail mail file, but then answered "No" to the query to confirm expunging the mail file, Zmail tried to abort the command but blew up instead.

This fix changes the semantics of answering "No" to the query to confirm expunging a mail file. In the past, a negative response would abort the command being executed. Now, a negative response simply aborts the attempt to expunge.

For example, if you had five mail files and type "S" to save them all in Zmail: In the past, answering "No" when asked to expunge the third mail file would abort the save command and the third, fourth, and fifth files wouldn't be saved. In Genera 8.1, answering "No" will abort only the expunge operation for the third file. Zmail will proceed to save the third file without expunging it and will then expunge and save the fourth and fifth files.

- The L command, which adds keywords to messages from the keyboard, has been extended to allow you to click on a message in the summary window and add that message's keywords to the current message.
- The variable **zwei:*default-reply-to-list*** defines a new profile option which is the initial contents of outgoing mail's Reply-To field.
- Several message reference formats generated by popular Unix mailers are now recognized by Zmail. Zmail uses the message references which appear in In-Reply-To and Message-ID header fields to link messages together into conversations.

- An improved subject search algorithm provides better subject matching.
- You can use the new Zmail command Set Key (m-k) to customize the user interface by interactively establishing a binding from a Zmail command name to a keystroke.
- There is a new command, Add Message References, to make re-assembling conversations easier:

Add Message References (m-k)

Add references to the current message. Prompts for a reference or references. Terminate by pressing END. Normally, typing a message reference is too cumbersome, however, so you can click Left on a message in the summary area to use its Message-ID immediately, or you can click Middle on a message in the summary area to have its Message-ID inserted into the minibuffer. Using Mouse Middle is especially useful if you mean to add several message references at once because you do not have to reissue this command several times — you can just accumulate all of the references in the minibuffer at once and then finally press END.

If you wish to place this command on a key, you can do

```
(zwei:set-comtab zwei:*zmail-comtab*
'(#\Super-R zwei:com-add-message-references))
```

- Genera 8.1 offers improved Unix RMAIL compatibility; per-message labels are now handled without leading spaces.
- A bug that could cause Zmail background processing to get stuck probing for new mail has been fixed.
- A bug which caused SELECT M to hang with the SELECT Key process in "Buffer Full" state when creating a new Zmail frame has been fixed.
- There is a new command, Merge Keywords In Conversation (m-k) to add any keywords on any message in a conversation to all the messages in the conversation.

Zmail Commands to Create or Receive ECOs

There are several Zmail commands for handling ECOs. The intent of these commands is to make it possible to distribute patches through electronic mail. The assumption is that you write a distribution to disk, and then using the Mail ECO, encode that distribution in an ascii representation and insert it in a mail buffer. If you have a very large distribution to send, and are afraid that there may be prob-

lems with encoding, you can split the process into two stages by encoding first with Make Encoded ECO File.

In actual fact, you can encode any kind of file as an ECO file, but non-distribution files cannot be loaded with Decode ECO.

Decode ECO (m-X) Zmail Command

Decode ECO (m-X)

Turns an ascii encoded ECO message back into a temporary file and then offers to load that file. You get an error if the file is not a distribution file.

Mail ECO (m-X) Zmail Command

Mail ECO (m-X)

Sends an ECO as a mail message. It prompts for a file containing an encoded ECO. If the file is not yet encoded for mailing, Mail ECO offers to encode it (prompting for a pathname to use to hold the encoded ECO). The encoded ECO is then placed in a message buffer. You can add any comments and terminate the message with END as for any other mail message.

Make Encoded ECO File (m-X) Zmail Command

Make Encoded ECO File (m-X)

Encodes a file for distributing as an ECO. It prompts for a source file and an output pathname to hold the encoded result.

There is also a CP command, Show ECOs, that allows you to see your ECO level:

Show ECOs Command

Show ECOs *keywords*

Shows any ECOs (Engineering Change Orders) loaded into your world.

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***.

Changes to the Mailer in Genera 8.1

- In Genera 8.1, the SMTP protocol is preferred over the older CHAOS-MAIL protocol.
- In Genera 8.1, the mailer has greater compliance with the Host Requirements RFC 1123, specifically section 5.2.14, which states that mailers should use four-digit years in date fields and also use numeric timezone offsets instead of the timezone names listed in RFC 822.
- Genera 8.1 fixes a bug in the mailer, which occurred when a series of Lisp machines exchanged mail using the SMTP protocol, causing several problems with the Return-Path field of the delivered message. First, the original workstation, which normally doesn't support mail service, was listed in the Return-Path. As a result, if the mail couldn't be delivered and a rejection notice was mailed back, it would sit in a mailer's queue as that mailer tried in vain to deliver it to the workstation. Second, if the mail passed among several mailers at a site, each mailer would add its name to the Return-Path. Consequently, the Return-Path could quickly exceed 255 characters in length which would cause it to be rejected by some Unix mailers.
- According to section 5.3.1.1 of the Host Requirements RFC, RFC 1123, the minimum interval before giving up on delivering a message should be 4-5 days. The Symbolics mailer previously defaulted to 3 days, but in Genera 8.1 the limit has been increased to one week. The following two mailer options that control the timeout are now documented:

mailer:unresolvable-host-interval

Variable

The minimum interval before the mailer gives up on delivering a message when mail cannot be delivered to a given host because domain information on it cannot be resolved. The default is one week.

mailer:undeliverable-host-interval

Variable

The minimum interval before the mailer gives up on delivering a message when mail cannot be delivered to a given host. The default is one week.

- If two or more domain names were resolved to the same host, the Mailer would try to add any message queued for those domains to the host's queue once for each original domain. This behavior resulted in notifications of the form:

```
DATE TIME from Mailer-Server@HOST: + Background: Attempt to add
MESSAGE to the queue for SOME-HOST but MESSAGE is already queued
for SOME-HOST.
```

This bug has been fixed.

- Previously, if you had mail queued for an unresolved domain and you cold boot the mailer, the mailer could fail to boot if, when restoring its work files, the mailer could now definitely determine that the previously unresolved domain does not exist. This bug has been fixed.

Documentation Updates Related to the Mailer

Note that changes to the **mailer:forwarding-table-hosts** variable don't take effect until the following form is evaluated:

```
(mailer:clear-parsed-forwarding-table-hosts)
```

If you put the above form into the Options.lisp file after the form which sets **mailer:forwarding-table-hosts**, then whenever the Mailer reads Options.lisp (e.g., because you clicked on "Update Options" in the Mailer log window), the new list of hosts will take effect.

The following mailer options are now documented:

mailer:logs-directory

Variable

The pathname of the directory where the Mailer will store its log files. The default is #P"LOCAL:>Mail>Dynamic>".

mailer:probe-interval

Variable

The minimum time to wait before retrying a host that was down when delivering outgoing mail. If this value is 10 minutes, the first probe will occur after 10 minutes, the second 20 minutes after the first, the third 30 minutes after the second, etc. The default is 10 minutes.

mailer:mailbox-deletion-threshold

Variable

If the number of entries in the mailbox table decreases by more than the threshold amount specified by this option, the mailer will refuse to perform any background updates on the theory that a major editing mistake has taken place. This option is interpreted as follows:

nil Don't check for shrinkage.

- $0 < \textit{threshold} < 1$ Refuse if more than this percentage of entries would be deleted.
- $\textit{threshold} \geq 10$ Refuse if more than this number of entries would be deleted.

The default is 100.

Changes to Zmacs in Genera 8.1

- The Zmacs command "Set Export For Buffer" is now available as a $m-x$ command when in C Mode. Previously, it was only available as a $c-m-x$ command. This command used to prompt for a setting in the minibuffer. It now uses a **y-or-n-p** style to just prompt for a single character.
- For command name consistency with the CP, the Zmacs command Set File Properties ($m-x$) has been added as a synonym for Change File Properties ($m-x$). (The old name is retained temporarily for compatibility, but might go away in a future release.)
- For more user interface uniformity, Clean File and Clean Directory are now available as both CP and Zmacs commands (that is, the Clean Directory CP command, and the ($m-x$) Clean File Zmacs command have been added).
- The first line of a buffer listing resulting from List Buffers, which says "Buffers in Zmacs", is mouse sensitive. You can click on this line to execute Edit Buffers on the same set of buffers. This is especially useful when you have done $c-U$ $c-X$ $c-B$ and then realize you wanted to do $c-U$ $c-X$ $c-sh-B$ instead.
- A new option, T , has been added to the set of choices available when you press HELP while in Zmacs. Pressing T will attempt to autoload and run the Zmacs tutorial from SYS:EXAMPLES:TEACH-ZMACS.LISP. The tutorial itself is not new, only its accessibility from the HELP key is new.
- Select All Buffers As Tag Table ($m-x$) now permits a numeric argument. If supplied, it prompts for a string and only buffers whose name contains that string are considered.
- In previous releases, if you used $m-.$ in Zmacs to edit the definition of a function specification that was defined as a readable instance variable of a component flavor of **tv:window**, you were asked if you really wanted to edit **flavor:read-instance-variable**. In Genera 8.1, you are first asked if you want to edit the flavor which declared that readable instance variable. For example,

```
m- . (FLAVOR:METHOD :BORDERS TV:WINDOW)
```

asks:

```
Do you mean instance variable TV:BORDERS of flavor TV:BORDERS-MIXIN? (Y or N)
```

- FEP command files are now automatically edited in a new "FEP-Command" mode.

FEP-Command Mode

Sets up the buffer for editing FEP command files. This mode is automatically entered when you read in a FEP .boot file. Use `C-M-X` FEP Command Mode to enter this mode manually. Comment delimiters are "(" and ")". `TAB` is Indent Relative. All tabbing is done by inserting spaces, not tabs.

- If List Callers (`M-X`) or any of the other Zmacs commands in the same family (Edit Callers, Multiple List Callers, List Callers in Package, List Callers in System to name a few) is invoked while the cursor is in a method definition, the default is now the name of the generic function rather than the name of the method.
- When a private patch buffer is selected in Zmacs, the site system is no longer offered as a default system upon which to operate when any of these commands was issued: Recompile Patch (`M-X`), Reload Patch (`M-X`), Resume Patch (`M-X`), or Revoke Patch (`M-X`). Instead, the default system offered is now "Private" in these cases.
- Start Private Patch (`M-X`) 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.

Changes to Utilities in Genera 8.1

SCSI Device Support for XL Machines in Genera 8.1

Installing a SCSI Device

This section describes how to physically connect a SCSI device to the XL machine, how to configure the device with an appropriate SCSI address.

Connecting the Device

1. Find an appropriate cable. You need a cable that has a standard SCSI connector (a 50-pin D connector) on each end. Note that you cannot use the cable provided by Storage Dimensions with the MacinStor SCSI disk, because one end of that cable does not have a standard SCSI connector.

2. Connect the SCSI cable to the SCSI connector in the back of the optical disk drive. Connect the other end of the SCSI cable to the SCSI connector in the back of the XL machine.
3. Be sure that the SCSI bus is terminated properly. Whatever SCSI device is at the end of the SCSI bus must have a SCSI terminator on it. If you need more information about how to connect SCSI devices, see the section "Attaching a SCSI Device".

Choosing the SCSI Address for the Device

1. Choose an unused SCSI address for the device.

Each SCSI device on the SCSI bus must have a unique SCSI address between 0 and 7 (inclusive). SCSI address 0 is used by the Ivory processor board, so that is unavailable.

If you have other SCSI devices attached, you need to find out what SCSI addresses they use. The Show Machine Configuration command gives the SCSI addresses of any devices attached to your XL machine.

2. When you have chosen a SCSI address, configure the device by setting the SCSI address according to the device manufacturer's instructions.

Using SCSI Optical Disk Drives

On XL machines, Genera supports the Storage Dimensions MacinStor Erasable Optical 1000 disk drive. You can create a FEP file system on an optical disk, and access the FEP file system in the usual ways. You can use the optical disk to store large files such as image files, world loads, and Static databases. (You cannot boot locally off a world load stored on an optical disk, but you can store the world load there for archival purposes and for netbooting from other machines.)

Cartridges formatted for use on XL machines are not compatible with MacIvory and vice-versa. On MacIvory, the MacinStor is treated like any other disk drive. MacIvory cartridges contain a standard Macintosh file system, and within that file system you may create one or more Ivory partitions. When using a MacinStor on a MacIvory, the procedures and commands described in the following sections do not apply. See the section "Using the MacIvory Control Panel".

Normal Use of an Optical Disk

The initial installation instructions are described in the section "Installing a SCSI Device". Once the disk drive is installed, and an optical disk has been initialized with a FEP filesystem, the normal use is as follows:

1. Insert the optical disk into the drive.

2. Mount the optical disk using the Mount Optical Disk CP command.
3. Use the optical disk by giving Genera commands in the same way that you would access any FEP filesystem. You can write files to the FEP filesystem and read them back. You can do Show FEP Directory on the FEP filesystem. (Note the restrictions below on using the optical disk for paging, or with LMFS, or with worlds.)
4. Dismount the optical disk using the Dismount Optical Disk CP command.
5. Eject the optical disk from the drive by pressing the button on the front of the drive.

Guidelines for Use of Optical Disks

It is important to understand that you cannot dismount the optical disk while any application is accessing it (for example, when there is an open stream to the optical disk). Also, you cannot eject the optical disk until it has been dismounted. The software prevents you from doing these things, in order to protect the integrity of the data on the FEP filesystem on the optical disk. This has the following implications:

Using the optical disk for paging is not recommended.

Once you have added a paging file that is stored on an optical disk, you cannot dismount (or eject) the disk until you next cold-boot Genera. It might be useful in an emergency to set up a paging file on the optical disk and use it long enough to save your files, knowing that you will cold boot soon anyway.

Putting a LMFS on an optical disk is not recommended.

Once you have activated a LMFS that is stored on an optical disk (which happens the first time you try to access any file from that LMFS), you cannot dismount (or eject) the disk until you next cold-boot Genera. This is because LMFS has no concept of a dismountable disk, and the integrity of the data must be protected by preventing the disk from being dismounted while LMFS is active. It might be useful in some emergency situations to create a single-partition LMFS on an optical disk and use it briefly, knowing that you will cold boot soon anyway. On the other hand, you can store any file in the FEP filesystem of the optical disk, so there seems little point in creating a LMFS on the optical disk.

Do not write protect erasable optical disks that you use as FEP filesystems. The software for accessing the FEP from Lisp cannot read from write-protected disks.

There is an additional restriction:

Using the optical disk for booting a world is not possible.

The optical disk drive is not recognized by the FEP until Genera is up and the Mount Optical Disk command has been given. Therefore, you

cannot boot from a world stored on an optical disk. You can, however, store worlds on an optical disk for archival purposes or to use the optical disk host as a Netboot server.

Setting up a Blank Optical Cartridge

1. When preparing to use a blank optical cartridge, insert it into the drive. Note that each cartridge has two sides which must be prepared separately. Only one side is accessible at any time. In Genera, the first thing to do is give the Write Partition Map CP command, and give the appropriate SCSI address as an argument. There are keyword arguments as well, but the defaults are usually appropriate.

Write Partition Map *SCSI-address*

Note that this command will erase any data on the disk, and create a new partition map on it.

2. Mount the optical disk by using the Mount Optical Disk CP command.

Mount Optical Disk *SCSI-address*

This command makes the optical disk accessible to Genera.

3. Create a FEP filesystem by giving the Create Initial FEP Filesystem command.

Create Initial FEP Filesystem *FEP-unit-number*

Note: Create Initial FEP Filesystem *differs* from the other commands used here in that it takes a FEP unit number, *not* a SCSI address.

The convention is that the FEP unit number is 7 greater than the SCSI address. For example, if you set up the optical disk drive at SCSI address 4, the FEP unit is FEP 11. (Note that if you have multiple I/O boards, and the optical disk drive is attached to the second I/O board, then the FEP unit is 23 plus the SCSI address.)

Now the optical disk has a FEP Filesystem on it, which you can access from Genera. Once the FEP filesystem has been created, you refer to it by the FEP unit number, not the SCSI address.

4. Show the FEP directory, as a test, by using the Genera CP Command:

Command: Show FEP Directory

You should see the new FEP unit appear along with the other FEP units.

Commands for Using Optical Disks

Write Partition Map Command

Write Partition Map *SCSI-address keywords*

Writes a new partition map on an optical disk. This command erases any data or partitions already on the disk.

SCSI-address The SCSI address of the optical disk drive.
keywords :Partition Map Size, :Apple Driver Size, :Apple Hfs Size,
 :FEPFS Size

The *keywords* enable you to control the size of four partitions. The values are in number of blocks, or Rest. If you supply keywords, three of the partitions should be specified as integers, and one should be Rest (indicating that the remaining space should be used for that partition).

:Partition Map Size The default is 40 blocks.
 :Apple Driver Size The default is 30 blocks.
 :Apple Hfs Size The default is 0 blocks.
 :FEPFS Size The default is Rest.

Create Initial FEP Filesystem Command

Create Initial FEP Filesystem *FEP-unit*

Creates an initial FEP filesystem on the given *FEP-unit*. This command is used for initializing an optical disk or a SCSI disk.

FEP-unit The number of the FEP unit.

The convention for mapping SCSI addresses to FEP unit number is that the FEP unit number is 7 greater than the SCSI address. For example, if you set up the SCSI disk drive at SCSI address 4, the FEP unit is FEP 11. (Note that if you have multiple I/O boards, and the SCSI disk drive is attached to the second I/O board, then the FEP unit is 23 plus the SCSI address.)

Mount Optical Disk Command

Mount Optical Disk *SCSI-address*

Mounts the optical disk indicated by *SCSI-address*. Once the disk is mounted, if it has a FEP filesystem on it, you can access it by using its FEP unit number. Otherwise, you should use the Create Initial FEP Filesystem command.

SCSI-address The SCSI address of the optical disk drive.

The convention for mapping SCSI addresses to FEP unit number is that the FEP unit number is 7 greater than the SCSI address. For example, if you set up the optical disk drive at SCSI address 4, the FEP unit is FEP 11. (Note that if you have multiple I/O boards, and the optical disk drive is attached to the second I/O board, then the FEP unit is 23 plus the SCSI address.)

Dismount Optical Disk Command

Dismount Optical Disk *SCSI-address keywords*

Dismounts the optical disk indicated by *SCSI-address*. Once the disk is dismounted, you cannot access it again until you mount it.

SCSI-address The SCSI address of the optical disk drive.

keywords :Eject

:Eject {Yes, No} Whether to eject the optical disk after it is dismounted. The default is No. The mentioned default is Yes.

Using Magnetic SCSI Disk Drives

On XL machines, Genera 8.1 supports the Storage Dimensions MacinStor SCSI disk drive series. Currently, only the MacinStor 320 and the MacinStor 650 have been qualified for use with XL machines. However, other SCSI disk models may also work properly.

Setting up a Blank SCSI Magnetic Disk

Note that new SCSI disks cannot be initialized from the FEP. You need an XL running Lisp to set up a new disk.

1. Connect the SCSI disk to the XL and power up the disk. Use the Format SCSI Disk command to format the disk. Provide the appropriate SCSI address as an argument.

```
Format SCSI Disk SCSI-address :Sector Size 1280
```

Note that this command will erase any data on the disk.

2. Warm boot Lisp so Genera will recognize the formatted drive.
3. Create a FEP filesystem by giving the Create Initial FEP Filesystem command. Provide the appropriate FEP unit as an argument.

```
Create Initial FEP Filesystem FEP-unit
```

The convention for mapping SCSI addresses to FEP unit numbers is that the FEP unit number is 7 greater than the SCSI address. For example, if you set up the SCSI disk drive at SCSI address 4, the FEP unit is FEP 11. (Note that if you have multiple I/O boards, and the SCSI disk drive is attached to the second I/O board, then the FEP unit is 23 plus the SCSI address.)

Now the SCSI disk has a FEP Filesystem on it, which you can access from Genera. Once the FEP filesystem has been created, you refer to the disk by the FEP unit number, not the SCSI address.

4. Show the FEP directory, as a test, by using the Genera CP Command:

Command: Show FEP Directory

You should see the new FEP unit appear along with the other FEP units.

Commands for Using SCSI Magnetic Disks

Format SCSI Disk Command

Format SCSI Disk *SCSI-address*

Formats the SCSI disk indicated by *SCSI-address*.

SCSI-address The SCSI address of the SCSI disk drive.

keywords :Controller, :Sector Size

:Controller No documentation supplied

:Sector Size The default is 1280 bytes

Create Initial FEP Filesystem Command

Create Initial FEP Filesystem *FEP-unit*

Creates an initial FEP filesystem on the given *FEP-unit*. This command is used for initializing an optical disk or a SCSI disk.

FEP-unit The number of the FEP unit.

The convention for mapping SCSI addresses to FEP unit number is that the FEP unit number is 7 greater than the SCSI address. For example, if you set up the SCSI disk drive at SCSI address 4, the FEP unit is FEP 11. (Note that if you have multiple I/O boards, and the SCSI disk drive is attached to the second I/O board, then the FEP unit is 23 plus the SCSI address.)

Using SCSI Tape Drives

SCSIQIC-11 tapes and 6250 bpi tape drives now work on XL machines. The initial installation instructions are described in the section "Installing a SCSI Device". Once the tape drive is installed, you can access it in the same way you access other supported tape devices. Note that if your XL is already running Lisp when you connect the new tape drive, you have to warm boot before the new tape drive is recognized.

Some XL and MacIvory machines use the Emulex MT02 controller, a QIC-11 SCSI tape drive. You need to add a PERIPHERAL entry to the namespace object of any host that uses the Emulex MT02 controller. If this entry is not present, only four of the nine tracks will be used. For example, for a controller at SCSI address 1, the entry should look like this:

```
Peripheral: TAPE UNIT SCSI1 MODEL EMULEX-MT02
```

Changes to the Data Compression Facilities in Genera 8.1

The Compression Facility now has a user interface. In addition, many small changes to the substrate have been made.

In Genera 8.0, the only interface to the compression facilities was in the Distribution System. Now it is possible for you to make use of data compression to store files in any file system, and to read and write compressed files from UNIX hosts.

The interface consists of two commands, Compress File and Decompress File:

Compress File Command

Compress File *input-files output-files keywords*

Compresses the data in *input-files* and produces *output-files*. Wildcards are allowed. If *input-files* and *output-files* are the same files, the input files are replaced by the output files.

<i>input-files</i>	{ <i>pathname(s)</i> } One or more files to compress.
<i>output-files</i>	{ <i>pathname(s)</i> } One or more files to contain the compressed data.
<i>keywords</i>	:Copy Properties, :Create Directories, :More Processing, :Output Destination :Preamble Type, :Query, :Translation Strategy

:Copy Properties {*list of file properties*} The properties you want duplicated in the new files. The default is author and creation date.

:Create Directories
{Yes, Error, Query} What to do if the destination directory does not exist. The default is Query.

: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***.

:Preamble Type {Symbolics, UNIX} Type of preamble to use.

:Query {Yes, No, Ask} Whether to ask before compressing each file.

:Translation Strategy

{Text, Binary, Query, Heuristicate} Whether or not to perform character set translation. Text means to do ASCII character set translation, reading each input file as a text file. Binary means not to do ASCII character set translation, reading each input file as a binary file. Query asks, for each file, whether to treat the file as text or binary. Heuristicate attempts to guess whether the file is text or binary based on its name, as follows: The filename is broken up into *words*, where each word is separated by a non-alphanumeric character. A rightmost word of "Z" is removed (if present). Then the current rightmost word is checked against **compression::*likely-unix-binary-formats*** If a match is found, the file is assumed to be binary, otherwise it is assumed to be text.

:Translation Strategy is only useful if you are reading or writing a file with a UNIX-style compression preamble, because Symbolics-style compression preambles record the element type and character set of the compressed data. Using :Translation Strategy with a file having a Symbolics-style compression preamble is ignored with a warning.

Decompress File Command

Decompress File *input-files output-files keywords*

Decompresses the data in *input-files* and produces *output-files*. Wildcards are allowed. If *input-files* and *output-files* are the same files, the input files are replaced by the output files.

input-files {*pathname(s)*} One or more files to decompress.

output-files {*pathname(s)*} One or more files to contain the decompressed data.

- keywords* :Copy Properties, :Create Directories, :More Processing, :Output Destination :Preamble Type, :Query, :Translation Strategy
- :Copy Properties *{list of file properties}* The properties you want duplicated in the new files. The default is author and creation-date.
- :Create Directories
{Yes, Error, Query} What to do if the destination directory does not exist. The default is Query.
- :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} Whether to ask before decompressing each file.
- :Translation Strategy
{Text, Binary, Query, Heuristicate} Whether or not to perform character set translation. Text means to do ASCII character set translation, writing each resulting file as a text file. Binary means not to do ASCII character set translation, writing each resulting file as a binary file. Query asks, for each file, whether to treat the file as text or binary. Heuristicate attempts to guess whether the file is text or binary based on its name, as follows: The filename is broken up into *words*, where each word is separated by a non-alphanumeric character. A rightmost word of "Z" is removed (if present). Then the current rightmost word is checked against **compression::likely-unix-binary-formats**, and if a match is found, the file is assumed to be binary, else it is assumed to be text.
- :Translation Strategy is only useful if you are reading or writing a file with a UNIX-style compression preamble, because Symbolics-style compression preambles record the element type and character set of the compressed data. Using :Translation Strategy with a file having a Symbolics-style compression preamble is ignored with a warning.

These two commands make use of the variable **compression::likely-unix-binary-formats**:

compression::*likely-unix-binary-formats***Variable*

A list of the file types that Compress File and Decompress File assume are binary files and do not need character set translation. This is only important if you are reading or writing a file with a UNIX-style compression preamble and are using :Translation Strategy Heuristicate. Users are encouraged to add or remove items from this list.

The current value of this variable is:

```
"TAR" "ARC" "ZOO" "LZH" "ZIP" "MID"
"GIF" "IFF" "TIF" "TIFF" "GEM" "NEO"
"SPC" "LIB" "OLB" "GL"
```

Changes to Tape in Genera 8.1

- Verifying a Carry Tape now works.
- Previously, when you booted a machine with its (external) SCSI tape turned off, you had to warm boot in order to make the machine notice the tape drive when you turned it on. In Genera 8.1, a proceed option offers to look again for tape drives on the local host. This is useful for SCSI-supporting systems, where tape drives can appear and disappear dynamically.
- Previously, the SCSI tape code did not work for tape controllers which do not support the CCS (Common Command Set); this is now fixed. Also, Show Machine Configuration now prints readable output for non-CCS devices.
- Another bug in SCSI tapes has been fixed, in which only one EOF was written at the end of a tape. This bug affected Exabyte drives.
- Finding SCSI tape drives now works on UX systems.
- There was a bug in the 8.0 SCSI tape formatter which caused "Medium Changed" errors in Format Local SCSI Tape. This bug has been fixed.
- The Distribution Dumper has been changed to dump the version of the system declaration that is appropriate to the system being dumped.

New TAR Tape Commands

The commands to read and write TAR tapes have been improved and new commands have been added to read, write, and show the contents of TAR tapes and files.

Read TAR Tape Command

Read TAR Tape *root-directory-for-relative-pathnames keywords*

Reads a tape in TAR format. It prompts for the tape specification.

root-directory-for-relative-pathnames

{*a directory*} The directory in which to put the files read from the tape.

keywords

:Mode, :More Processing, :Output Destination, :Reroot Absolute Pathnames

:Mode {Binary, Query, Text} The mode in which to perform the copy. The default is Query, ask about each file. Binary means binary bytes with no character set translation. Text means text characters with UNIX character set translation.

: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***.

:Reroot Absolute Pathnames

{Yes, No} Uses the specified root directory as the "root" for absolute pathnames. The default is No. The mentioned default is Yes.

Write TAR Tape Command

Write TAR Tape *pathnames*

Writes a tape in TAR format. It prompts for the tape specification.

pathnames

The pathnames of the files to write on the tape.

keywords

:More Processing, :Output Destination, :Relativize, :Since

:Mode

{Binary, Heuristicate, Query, Text} Mode in which to perform the copy. Binary means binary bytes with no character set

translation. Heuristicate means try to determine automatically per file. Query, the default, means ask for each file. Text manes text characters with UNIX character set translation.

: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***.

:Relativize

{Yes, No} Whether to try to write the pathname as a relative UNIX pathname, that is relative to the current UNIX directory instead of starting at root. The default is No. The mentioned default is Yes.

:Since

{*universal-time-in-the-past*} Writes only files later than the specified date.

Show TAR Tape Command

Show TAR Tape *keywords*

Shows the contents of a TAR tape. It prompts for the tape specification.

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} Enables you to direct your output. The default is the stream ***standard-output***. Note that redirecting output to a printer can be particularly useful.

Read TAR File Command

Read TAR File *tarfile* *root-directory-for-relative-pathnames* *keywords*

Reads a file in TAR format.

tarfile {*a pathname*} The TAR file to read.

root-directory-for-relative-pathnames {*a directory*} The directory in which to put the files read.

keywords :Mode, :More Processing, :Output Destination, :Reroot Absolute Pathnames

:Mode {Binary, Query, Text} The mode in which to perform the copy. The default is Query, ask about each file. Binary means binary bytes with no character set translation. Text means text characters with UNIX character set translation.

: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***.

:Reroot Absolute Pathnames {Yes, No} Uses the specified root directory as the "root" for absolute pathnames. The default is No. The mentioned default is Yes.

Write TAR File Command

Write TAR File *pathnames*

Writes a File in TAR format.

pathnames The pathnames of the file(s) to write.

keywords :More Processing, :Output Destination, :Relativize, :Since

:Mode {Binary, Heuristicate, Query, Text} Mode in which to perform the copy. Binary means binary bytes with no character set translation. Heuristicate means try to determine automatically

per file. Query, the default, means ask for each file. Text manes text characters with UNIX character set translation.

: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***.

:Relativize

{Yes, No} Whether to try to write the pathname as a relative UNIX pathname, that is relative to the current UNIX directory instead of starting at root. The default is No. The mentioned default is Yes.

:Since

{*universal-time-in-the-past*} Writes only files later than the specified date.

Show TAR File Command

Show TAR File *tarfile keywords*

Shows the contents of a TAR tape.

tarfile {*a pathname*} The file to be read.

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} Enables you to direct your output. The default is the stream ***standard-output***. Note that redirecting output to a printer can be particularly useful.

Changes to SCSI and ESDI in Genera 8.1

- The SCSI code now handles SAVE DATA POINTERS properly.
- Although not a standard option, the SCSI and ESDI drivers now support multiple I/O boards (#0 and #1).
- **scsi:with-scsi-port** and **scsi:scsi-port-open** now accept a **:controller** argument. **nil** means the default controller. **0** and **1** indicate which I/O board.
- There are two new SCSI functions:

scsi:map-over-scsi-ports *function* *Function*

Calls the given *function* (which must accept one argument) for each possible SCSI port on the system.

scsi:scsi-port-address *port* *Function*

Returns three values: the controller, the bus address, and the logical unit (always 0).

Changes to Enable CD-ROM Distribution

Genera 8.1 lays the foundation for distributing software on CD-ROMs. One aspect of this is a new kind of pathname:

ISO 9660 Pathnames

An ISO 9660 pathname looks like:

host|CDROMn :>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.

New Memory Error Correction Process for Ivory Machines

There is a memory error *scrubber* process for memory error correction on Ivory machines. It runs in the background, consuming less than 0.1% of CPU looking for soft errors in memory that can be repaired by reading the corrected data and re-

writing it so that there are no further errors. (A soft error that goes uncorrected would otherwise risk becoming a hard error by a second bit degrading.)

The scrubber can also discover uncorrectable errors in memory, before you get bit-ten by them. It sends a notification every time it hits an uncorrectable error, on the assumption you do not really want to be running with broken memory.

If some memory errors put you into the debugger while some uncorrectable errors put you into the FEP, you only go to the FEP if the uncorrectable error happens at a "bad time." If you are in the debugger, it provides the usual resume options. In this case, you should save your work and cold boot in order to reset your world. You can run memory tests in the FEP to determine whether or not the memory is actually broken, or that the error is just a temporary glitch.

See the section "FEP Commands for Systems Experts".

There is a CP command to display the error corrections, see the section "Show Memory Error Corrections Command".

Miscellaneous Changes to Utilities in Genera 8.1

- The Distribute Systems command and Distribute Systems activity have been modified to make it more convenient to distribute to disk. The distribution plan is not divided into parts according to any size limit, when written to disk.
- Genera 8.1 fixes a bug in which the disk driver could hang when saving a world.
- **metering:define-metering-function** and related functions now allow declarations. It is useful to be able to insert **inline** and **notinline** declarations and see the effects.
- There is a new function, **si:show-callers**, which does the same thing as the Show Callers CP command:

si:show-callers *symbol* &key *called-how* *package* *system* (*printer* **t**) *Function*

Tries to find all the functions in the Lisp world that call *symbol*. This is the function underlying the Show Callers CP command.

symbol The name of a defined function whose callers to locate.

:called-how Lists only those callers that refer to *symbol* as a particular type of Lisp Object. The possible types are:

Condition	Flavor-Component	Macro
Constant	Function	Microcoded-Function
Defined-Constant	Generic-Function	Unbound-Function
Flavor	Instance-Variable	Variable

- :package** Lists only those callers in the specified package.
- :system** Lists only those callers from the specified system.

See the function **who-calls**.

- The argument to the CP command Show Expanded Lisp Code is now optional.

Both the CP command Show Expanded Lisp Code and the **mexp** function used to complain if you typed the name of an unbound variable to be expanded, or if you typed a form the **car** of which was the name of an undefined function. Such forms are no longer screened out by the reader in this context.

- If a postscript file being read using the Read Image CP Command contains some text, you could end up in the debugger. This bug has been fixed.
- A bug on Ivory machines in the Metering section of Peek that caused you to go into the debugger if you clicked on Storage Meters has been fixed.
- In a previous release, when the host for a print spooler was not available, you were forced to enter the main debugger. In this release, the behavior has been changed so that you enter the pop-up debugger, since it's rare that the error was due to a program bug and probably you just want to select an alternate printer from the pop-up menu and let the hardcopying continue.
- In the File System Editor (FSMaint), the default for Repatriation of orphans during a Salvage operation has been changed to No. This is considered safer than automatically trying to repatriate orphans.
- In previous versions of Genera, if you used ACLs and got a LMFS capability error, you would end up in the debugger. In Genera 8.1 you can proceed from the error.

Changes to System Construction Tool in Genera 8.1

Change to **sct:copy-system** in Genera 8.1

The **:destination** keyword argument to the function **sct:copy-system** is required. Failing to supply it previously resulted in an error, but the error message was not very intelligible. This argument is now checked for right away and a better error message is signalled if it is missing.

The **:destination** keyword argument to the function **sct:copy-system** has been enhanced to permit general kinds of values. Previously, the only documented possibility was a single non-wild filename (pathname or string) with host, device and directory. The updated documentation for the **:destination** keyword to **sct:copy-system** follows:

: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
```

Change to Compile System for Concordia

Genera 8.1 provides a convenient way to compile a documentation-only system for both Symbolics architectures (Ivory and 3600-family).

Compiling a system consists of compiling (when necessary) its component files, loading them, and recording the exact version of sources and products (if any) for the newly compiled version. If a system consists solely of documentation, the step of compiling the component files is not necessary; compiling the system simply loads the files and records the version information. The loading step can be time-consuming.

In Genera 8.1, you can compile a documentation-only system on one architecture, and automatically "copy compile" it for the other architecture. The Compile System command takes the new keyword argument **:Copy Compile**, which does the following:

```
: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.

Similarly, `compile-system` now takes a new keyword argument:

`: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".

Miscellaneous Changes to System Construction Tool in Genera 8.1

- Genera 8.1 fixes a bug in which `sct:reap-obsolete-system-versions` would get an error when it was about to delete all the files in the directory that contains a certain component dir but then it discovers that the directory itself does not exist. `sct:reap-obsolete-system-versions` now warns the user if the directory was not found, instead of putting the user into the Debugger.
- A change has been made which affects only those users who still need to convert old-style code using `make-system` to System Construction Tool. When you try to recompile code using `make-system`, you get an error which suggests that you use `sct:convert-system-directory`. This conversion tool was last shipped in Genera 7.2, and was unavailable thereafter. In Genera 8.1, it is again made available, and is now in the file `SYS:UNSUPPORTED;convert-journals.*`.

Changes to the X Window System in Genera 8.1

The X server now conforms to MIT X11 R4 (the X client already conformed to R4). Previously, the Symbolics X client (X-Remote-Screen) didn't recognize the Symbolics X server when using a Symbolics keyboard. It claimed that the keyboard type was the machine type and that there was no defined layout for the keyboard. This bug has been fixed.

Genera 8.1 fixes a bug in which if you were not using the main console (for example, you were connected via an X Window) and you used the Namespace Editor

commands Edit Namespace Object or Create Namespace Object, the namespace editor would expose on the main console and not the remote console. (This is not a change to the X Window system itself, but it affects X Window system users.)

Previously, when a local X Window system "connection" timed out, an error would happen, causing entry to the Debugger. This bug has been fixed.

Changes to MacIvory in Genera 8.1

Changes and Improvements in Accessing the Macintosh File System

- The performance of Macintosh file system operations has been improved.
- It is now possible to quote asterisks (and other special characters) in Macintosh filenames when they are typed from Genera. See the section "Macintosh Path-name Quoting".
- The prohibition against renaming across directories (folders) on a Macintosh volume has been lifted in this release. Renaming across volumes is still prohibited since the Macintosh does not natively implement that capability.
- The state of the Finder's bundle bit is now returned as the **:bundle** property of any Macintosh file.
- A number of problems with accessing a Macintosh file system have been fixed. These problems were:
 - ° (directory #p"HOST:Volume:No Such File") signaled a "File not found" error rather than returning **nil**.
 - ° (directory #p"HOST:Volume:No Such Folder:*") signaled "File not found" instead of "Directory not found". Consequently, the proceed option to create the directory and any missing superiors was not offered.
 - ° (directory #p"HOST:Volume:Existing File:*") did not signal "Not a directory".
 - ° (probe-file #p"HOST:Volume:Existing Folder") returned **nil**.
- Closing a Macintosh FS stream with **:abort t** sometimes signaled errors.
- Unknown properties for files on a Macintosh file system are now handled properly. Specifically, the system now signals an error properly built on **fs:change-property-failure** if you attempt to change any property of a Macintosh file other than **:creation-date**, **:backup-date**, **:file-type**, **:creator-signature**, or **:bundle**.

- When listing the contents of a Macintosh directory, the block count reported for a file was not the actual number of blocks occupied by the file on disk. The reported block count was based on a 1024 byte block size rather than the actual block size reported in the directory listing header. This bug has been fixed.

Changes to Booting Genera on MacIvory

- Booting Genera on the MacIvory is more robust, especially in the case where the date and time cannot be obtained from the toolbox server, for example because the wrong file is present or because the folder has been renamed.
- Warm-booting Genera on the MacIvory is more robust. Previously, if a MacIvory had an active serial channel and you warm booted the Ivory without restarting the Macintosh, the Macintosh would crash a short time later.

Bugs in Interaction of Symbolics Keyboard with Macintosh Software Fixed

Several bugs in the interaction of the Symbolics keyboard with non-Symbolics Macintosh software have been fixed. These bugs include the following:

- The right-hand modifier keys (SHIFT, COMMAND (SUPER), OPTION (META), and CONTROL) were not always recognized by Macintosh software. Note that, while most software now recognizes these keys, some software (such as Pyro!) still does not. Perhaps the most important piece of Macintosh software which now recognizes these keys is Apple's Finder. In particular, you can now use the right-hand SHIFT key to perform a multiple selection within the Finder.
- You can now map Apple key functions to keys not directly on the Symbolics keyboard when CloseView is installed using the techniques described in "Mapping of Apple Key Functions to the Symbolics Keyboard". Note that there is still some incompatibility between CloseView and the Symbolics keyboard. See the section "Interoperability of the Symbolics Keyboard and Popular Macintosh Software".
- QuickKeys 2 now reliably interprets the keys typed on a Symbolics keyboard. See the section "Interoperability of the Symbolics Keyboard and Popular Macintosh Software".

Changes to the Genera Application and Other Applications Created on MacIvory

The following bug fixes apply to the Genera, Unassigned, and Mac Dex applications. In addition, they also apply to any applications you create from the aforementioned applications using the Configure MacIvory Application CP command. For simplicity, the bug fix descriptions only mention the Genera application.

- The bug which prevented the Genera application from passing `COMMAND-Shift-n`, where n is any digit, to Lisp has been fixed. On a Symbolics keyboard, the `COMMAND` modifier is entered by pressing the `SUPER` key which meant it was impossible to enter `SUPER-!`, etc. While on an Apple keyboard, the `COMMAND` modifier is normally interpreted as the `SUPER` key, this assignment could be changed. (See the section "Changing the Keyboard Mappings on a MacIvory".) In particular, if you changed the `COMMAND` modifier to map to the `META` key, it became impossible to type `META-!`, etc., including `META-$` which is the Zmacs Spell Word command.
- The bug which prevented Pyro! from running your selected screen savers while the Genera application was the active application has been fixed.
- The Genera application now recognizes that the Apple ISO Extended Keyboard is functionally equivalent to the Apple Extended Keyboard II and treats both keyboards identically.
- When you exit the Genera application or switch to another application under MultiFinder, the Genera application now properly resets the state of the Apple Extended Keyboard II. Consequently, the Finder and other Macintosh applications now recognizes the righthand `SHIFT`, `OPTION`, and `CONTROL` keys as being equivalent to their lefthanded counterparts.

Symantec AntiVirus for Macintosh

Symantec AntiVirus for Macintosh (SAM) is being distributed with Genera 8.1. It is contained on *how many* floppy diskettes labelled *something-or-other*.

You should be sure to install it **and** run it to protect your system against viruses. We think that SAM offers an advantage over most other anti-virus software in that it offers an Automatic Scan feature (SAM Intercept) that continually monitors all system activity and warns you of suspicious activity that could be the result of a virus. (Of course, if you regularly perform certain actions that trigger SAM Intercept warnings, it has a Learn feature that allows you to teach it to ignore those actions.)

In addition, Symantec offers a Virus Hotline so that you can keep informed of new viruses as they appear.

SAM also includes a facility to repair files damaged by viruses, SAM Virus Clinic. This facility can repair known viruses. It is extensible, so that if you learn about a new virus from the Virus Hotline, you can add information about that virus to the Clinic.

Your SAM software contains:

- floppies with the software
- documentation

- registration card

We urge you to be sure to install SAM on your MacIvory.

Miscellaneous Changes to MacIvory in Genera 8.1

- In the past, MacIvory has been used as both a generic term referring to all models of MacIvory and as a specific term referring to the original model MacIvory. As of Genera 8.1, MacIvory is now used only as a generic term. The original model MacIvory is now referred to as a "MacIvory model 1".
- All relevant software in Lisp, the IFEP, and the Macintosh has been changed to use the term "MacIvory model 1". Of particular note, **si:machine-model** now returns `:|MacIvory model 1|` instead of `:|MacIvory|`.
- A bug in the Configure MacIvory Application command which caused the command to reject any attempts to enter an Initial Application Command unless the selected remote program was also running at the time has been fixed.
- The disk throttling mechanism of MacIvory life support can now be completely disabled by setting the value of its "Idle" parameter to zero in the MacIvory control panel file.
- MacIvory life support now maintains more accurate Ethernet meters. In particular, any incoming packets which are discarded because Lisp is unable to accept them are counted as part of the **other_receive_errors** meter.
- MacIvory now displays its run lights at the bottom of its cold load window whenever the cold load window is exposed. When netbooting, MacIvory also displays a progress bar at the bottom of its cold load window.
- The bug which caused a MacIvory without an Ethernet card to hang during booting if not configured as a standalone machine has been fixed.
- The bug in the Disk Copy option of the MacIvory control panel which caused copying of Macintosh disks to always fail with the error code -55 has been fixed. As part of this fix, however, the MacIvory control panel now restarts the Macintosh after successfully copying Macintosh disks.
- The Ivory Breath of Life application has been extended to allow the selection of a source medium if multiple media (i.e., tape and CD-ROM) are present. The application has also been extended to optionally allow the restoration of tape or CD-ROM contents without destroying the existing files on the FEP filesystem.

Documentation Update for MacIvory

Routines for RPC Error Handling in RPC.lib

RPCRemoteError Routine

RPCRemoteError (long *error-number)

Returns the remote error number of the last RPC call that failed. This routine is useful with the individual functions that access remote-error values to allow error handling.

ReportRPCOpenFailure Routine

Boolean ReportRPCOpenFailure (OSErr error, Boolean embeddedP, char* host)

Reports a failure when opening an agent (that is, when calling `emb_agent_open`). The argument `error` is the return code from the call. For now, `embeddedP` should be `TRUE` and `host` should be `0L`. Returns `TRUE` if unable to recover from the error; returns `FALSE` if able to recover from the error and if the agent is open.

ReportRPCCallFailure Routine

Boolean ReportRPCCallFailure (Boolean fatalP, OSErr error, Boolean embeddedP, char *host)

Reports a failure from an RPC call. The argument `error` is the return code from the call. The argument `fatalP` should be `TRUE` or `FALSE` based on whether the error can be recovered from. For now, `embeddedP` should be `TRUE` and `host` should be `0L`. If `fatalP` is `FALSE`, this routine returns `TRUE` if the user decides to give up and returns `FALSE` if he wants to try again. If `fatalP` is `TRUE`, the routine always returns `TRUE`.

Installing and Using Printers with MacIvory

Using an Appletalk Printer From the Macintosh and Genera

- Printing via AppleTalk is supported only for Postscript printers.
- Printing via AppleTalk can not be used with our Print Spooler.

To use an AppleTalk printer do the following:

1. Select your target printer on the Macintosh side through the Chooser.
2. Create a printer object in the namespace that identifies your MacIvory as the host to which the printer is connected.

3. Set the default printer for your MacIvory to the printer object you just created. You can do this either in the namespace to make the effect permanent or with the Set Printer CP command.

You can now use the printer from Lisp using the normal hardcopy facilities of Genera.

The use of separate printer objects is required even if several MacIvories are using the same AppleTalk printer. Of course, each printer object in the namespace must have a unique name and pretty name. Therefore it is a good idea to include the name of the MacIvory in the printer's names.

For example,

Showing PRINTER SOUR-CREAM-LOS-ANGELES-TIMES in namespace SCRC:

```
Type: LGP2
Site: SCRC
Pretty Name: The Los Angeles Times for Sour Cream
Interface: AppleTalk
Host: SOUR-CREAM
Printer Location: SCRC 2 Outside KMP's office
```

The Type attribute should be LGP2, LGP3, or POSTSCRIPT depending on the type of printer. LGP2 corresponds to the old LaserWriter and LaserWriter Plus. LGP3 corresponds to the newer LaserWriter IINT and LaserWriter IINTX. LGP3 also works for Apple's latest printer, the Personal LaserWriter NT. Use POSTSCRIPT for non-Apple Postscript printers (for example, HP).

In the namespace object for the MacIvory, make sure that the BITMAP-PRINTER attribute is empty. Otherwise, Lisp tries to invoke the Print Spooler, which does not work.

If your Macintosh is running MultiFinder, it's a good idea to enable background printing in the Chooser when you select your AppleTalk printer. Otherwise, whenever you print something in Lisp, Genera stops while printing takes place. (You see the familiar Macintosh modal dialog boxes describing the print process appear in front of the Genera screen.)

Using the Print Spooler with Genera

To spool an LGP2/LGP3 printer from a MacIvory machine, you need one 8-pin-male-to-25-pin-male cable with a null modem in it. Alternatively, you can use any combination of cables that provides you with this configuration.

1. Plug the printer into one of the serial ports on the back of the Macintosh computer. You can run the printer from the modem port (UNIT 0) or the printer port (Unit 1).
2. Edit the printer's Namespace Object as follows:

```

Interface: SERIAL
Interface Options: UNIT n BAUD 9600 PARITY :none NUMBER-OF-DATA-BITS 8
                  XON-XOFF-PROTOCOL yes
User Property: DTR-VALID nil

```

More information is available about editing the printer's namespace object. See the section "Attributes for Objects of Class "Printer"". See the section "Using the Namespace Editor".

3. Set the baud rate to 9600 on the LGP2/LGP3 printer.

Note: If you need more information about Macintosh serial ports, check the Unit specification conventions in the MacIvory User's Guide.

For additional information about setting up the print spooler and the LGP2/LGP3 printer, see the section "Installing a Printer".

See the section "Uncrating the Printer".

See the section "Specifying the Switch Settings".

See the section "Loading the Hardcopy and Printer Support Tape".

See the section "Registering a Printer".

Note: If you boot the MacIvory while the print spooler is running, you must then restart the Macintosh computer. Otherwise, the serial stream is left open (and attempts to restart the print spooler will fail with an "unable to access the serial stream" error).

Using the Symbolics Keyboard with Native Macintosh Applications

Use of the Symbolics keyboard with the MacIvory is provided primarily for compatibility with Genera applications. In addition, Symbolics supports the use of the Symbolics keyboard with native Macintosh applications.

Mapping of Apple Key Functions to the Symbolics Keyboard

- All "ordinary" characters (printing graphics, TAB, SPACE, RETURN) work normally.
- Apple's ESC key is entered using our ESCAPE key.
- Apple's COMMAND key is entered using either of our SUPER keys.
- Apple's OPTION key is entered using either of our META keys.
- Apple's CONTROL key is entered using either of our CONTROL keys.
- Apple's ENTER key is entered using our SCROLL key.

- Apple's CLEAR key is entered using our CLEAR INPUT key.
- Apple's HELP key is entered using our HELP key.
- Apple's END key is entered using our END key.
- Function keys (F1 - F15) are entered using the FUNCTION key as a shift key as follows:

<i>Apple Symbolics</i>	<i>Apple Symbolics</i>	<i>Apple Symbolics</i>
F1 FUNCTION-1	F6 FUNCTION-6	F11 FUNCTION--
F2 FUNCTION-2	F7 FUNCTION-7	F12 FUNCTION==
F3 FUNCTION-3	F8 FUNCTION-8	F13 FUNCTION-'
F4 FUNCTION-4	F9 FUNCTION-9	F14 FUNCTION-\
F5 FUNCTION-5	F10 FUNCTION-0	F15 FUNCTION-

- The numeric keypad keys are entered using the SYMBOL key as a shift key as follows:

<i>Apple Symbolics</i>	<i>Apple Symbolics</i>	<i>Apple Symbolics</i>
0 SYMBOL-0	6 SYMBOL-6	* <i>see note.</i>
1 SYMBOL-1	7 SYMBOL-7	- SYMBOL--
2 SYMBOL-2	8 <i>see note</i>	<i>see note</i>
3 SYMBOL-3	9 SYMBOL-9	. SYMBOL-.
4 SYMBOL-4	= <i>see note</i>	
5 SYMBOL-5	/ SYMBOL-/	

Note: In the table above, no mappings are provided for numeric 8, numeric =, numeric *, and numeric +. These four characters require special treatment because * is SHIFT 8 and + is SHIFT =. Symbolics distinguishes NUMERIC SHIFT 8 from NUMERIC * by enabling you to use the SHIFT key on the same side as the SYMBOL for producing the unshifted key with a SHIFT modifier, and using the SHIFT and SYMBOL keys on opposite sides for producing the shifted key without a modifier. Note that both SHIFT keys are required to produce a shifted key with a SHIFT modifier. For example:

```

LEFT SYMBOL 8 produces NUMERIC 8
LEFT SYMBOL LEFT SHIFT 8 produces NUMERIC SHIFT 8
LEFT SYMBOL RIGHT SHIFT 8 produces NUMERIC *
LEFT SYMBOL LEFT SHIFT RIGHT SHIFT 8 produces NUMERIC SHIFT *
LEFT SYMBOL = produces NUMERIC =
LEFT SYMBOL LEFT SHIFT = produces NUMERIC SHIFT =
LEFT SYMBOL RIGHT SHIFT = produces NUMERIC +
LEFT SYMBOL LEFT SHIFT RIGHT SHIFT = produces NUMERIC SHIFT +

```

The remaining keys on the extended keyboard are entered using the SYMBOL key as a shift key as follows:

<i>Apple</i>	<i>Symbolics</i>	<i>Apple</i>	<i>Symbolics</i>
↑	SYMBOL-i	HOME	SYMBOL-k
↓	SYMBOL-,	PAGE UP	SYMBOL-PAGE
←	SYMBOL-j	PAGE DOWN	PAGE
→	SYMBOL-l	DEL FWD	SYMBOL-RUBOUT

Interoperability of the Symbolics Keyboard and Popular Macintosh Software

Certain popular Macintosh software does not interoperate fully with the Symbolics keyboard. In some cases, a simple workaround (for example, renaming a file) exists which enables full function. In other cases, there is no workaround but the limitations are well known and are presented here for your convenience.

The Symbolics Keyboard and CloseView

CloseView, from Apple, allows the visually handicapped to magnify portions of their screen to make reading the screen easier. When using a Symbolics keyboard, however, the keystrokes used to raise and lower the magnification factor, Command-Option-↑ and Command-Option-↓, respectively, can not be entered using the techniques described in "Mapping of Apple Key Functions to the Symbolics Keyboard". Instead, you must disable magnification using Command-Option-X, open the CloseView control panel, change the magnification factor by clicking on the arrow buttons, close the CloseView control panel, and re-enable magnification using Command-Option-X.

The Symbolics Keyboard and Pyro!

Pyro!, from Fifth Generation Systems, is one of the most popular screen saver facilities for the Macintosh. Once running, Pyro! waits for you to press any key or move the mouse as an indication that you wish to restore the normal screen contents. However, when using a Symbolics keyboard, Pyro! will not deactivate itself if you press any of the righthand modifier keys (that is, SHIFT, CONTROL, META, and SUPER). Just use the lefthand modifier keys instead and Pyro! will deactivate.

The Symbolics Keyboard and QuickKeys 2

QuickKeys 2, from CE Software, is one of the most popular keyboard macro facilities for the Macintosh. For proper operation, the Symbolics keyboard software must load after MacIvory's support software but before QuickKeys 2. If you have INIT-Picker, or similar software for controlling the order in which INITs are loaded, see the documentation on said software for details on how to arrange the proper loading order. If you do not have such software, you can use the fact that the Macintosh System Software loads INITs in alphabetical order to get the desired effect. In particular, you can rename either the Symbolics Keyboard INIT or the QuickKeys 2 INIT to obtain the proper loading order. (We recommend that you rename Symbolics Keyboard to MSymbolics Keyboard.)

Macintosh Pathname Quoting

To the Macintosh, all characters are legal in pathnames and only colons delimit directories. This means, for example, that asterisks may appear as a component of a Macintosh pathname. Since Genera considers asterisk to be a wildcard, you cannot type a Macintosh pathname containing an asterisk without quoting it. Therefore, when typing a pathname such as

```
HOST:volume:*Graphics:picture
```

you need to quote the asterisk to indicate to Genera that it is not a wildcard. Circle-Plus (\oplus , sy-sh-+) is the quote character. So you would actually type:

```
Host:volume:\oplus*Graphics:picture
```

This also means that when Genera prints a Macintosh pathname that has an asterisk as part of its name, it quotes the asterisk using the Circle-Plus character.

Changes to UX in Genera 8.1

- Previously, if the user's keyboard had no META key, life support did not provide an alternate. In Genera 8.1, if the keyboard has no META key, and two ALT keys (one on the left, one on the right), then the left ALT key is mapped to META, and the right ALT key is mapped to SYMBOL. If there is only one ALT key, it is mapped to SYMBOL.
- UX400S and UX1200S machines now display their run lights at the bottom of their cold load window whenever the cold load window is exposed. When netbooting, UX400S and UX1200S machines also display a progress bar at the bottom of their cold load windows.
- In the configuration file for a UX, the interface command is no longer required. If no interface command is present, then the UX is only able to communicate with the host Sun, and any other UXes embedded in the same Sun.
- The ivory-life UNIX program uses the UNIX syslog() facility (as well as printing out the message on the console), so that users who are having problems can log any errors for later examination. By default, ivory-life uses the LOG_LOCAL0 facility, which normally causes error messages to be logged in the file /var/adm/messages. Use the -F flag to ivory-life to specify a different facility; the argument to -F is a facility number, or -1 to disable use of syslog().
- The UX Installer program used to automatically re-read the FEP tape (prompting you to change tapes several times) when creating multiple FEP partition files. In Genera 8.1 it quietly and quickly copies world load and FEP files from the first FEP partition file it creates into any subsequently created partitions.
- The UNIX Talk protocol now works with UX-family machines.

Changes to Concordia in Genera 8.1

When the default text printer for a host was an LGP3, Format Pages would get an error because it couldn't get the book design information correctly. Format Pages now works for both LGP2 and LGP3 printers.

Previously, when a picture extended below the bottom of a Document Examiner window, and there was a documentation topic already following the topic being displayed, the Document Examiner did not create enough room for the whole picture to be displayed, so part of the picture was displayed over the following topic. This bug has been fixed.

Change to Compile System for Concordia

Genera 8.1 provides a convenient way to compile a documentation-only system for both Symbolics architectures (Ivory and 3600-family).

Compiling a system consists of compiling (when necessary) its component files, loading them, and recording the exact version of sources and products (if any) for the newly compiled version. If a system consists solely of documentation, the step of compiling the component files is not necessary; compiling the system simply loads the files and records the version information. The loading step can be time-consuming.

In Genera 8.1, you can compile a documentation-only system on one architecture, and automatically "copy compile" it for the other architecture. The Compile System command takes the new keyword argument `:Copy Compile`, which does the following:

`: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.

Similarly, `compile-system` now takes a new keyword argument:

`: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".

Changes to Static in Genera 8.1

The Static released with Genera 8.1 includes some bug fixes and improvements documented below. More important though is that Genera 8.1 includes Static Runtime, the portion of Static which allows you to use, access, and operate a Static database. Static Runtime is a generous subset of Static Developer; Static Runtime does not enable you to develop Static applications, but it does allow you to use them. See the section "Documentation Update: Static Runtime".

Miscellaneous Changes to Static in Genera 8.1

Static used to have its own locking substrate. That has been replaced with the System's locking substrate. This results in greater reliability and somewhat better performance.

The implementation of finishing transactions (whether committing or aborting them) has been changed. Previously, large amounts of code were surrounded by **without-interrupts**. This implementation was particularly burdensome on UX machines, because while the code within **without-interrupts** was running, no other process could interrupt, which could cause network connections to break. The network is important for UX users, because it keeps the console alive. The **without-interrupts** implementation has been replaced with a locking scheme. This results in greater robustness.

Changes to DNA in Genera 8.1

This is not a change to DNA, but an explanatory note about VAX/VMS V5.2.

VAX/VMS V5.2 uses CHECK INPUT messages more frequently than in previous releases. This means that you are likely to encounter this error while you are remotely logged in to a VAX/VMS host via CTERM:

Error: Don't know how to handle CTERM CHECK INPUT messages

**(DEFUN-IN-FLAVOR TELNET:CTERM-CHECK-INPUT-MESSAGE
TELNET:CTERM-TYPEOUT-FILTER)**

Arg 0 (SELF): #<TELNET:CTERM-TYPEOUT-FILTER 334035224>

Arg 1 (SYS:SELF-MAPPING-TABLE):

#<Map to flavor TELNET:CTERM-TYPEOUT-FILTER 114050751>

s-A, : Restart process Terminal 1 Typeout

One example of where this can happen is when you are running NCP, type SHOW LINK, and press RETURN, as follows:

```
VMS> MC NCP
NCP> SHOW LINK
<error occurs here>
```

To recover from this particular error, you can specify a link number on the command line. For example:

```
VMS> MC NCP
NCP> SHOW LINK 666
<output>
NCP>
```

Changes to FORTRAN in Genera 8.1

A bug was fixed which affected Fortran users. Some large Fortran programs could not be compiled because of a "too many constants" error in the Lisp compiler.

Notes and Clarifications in Genera 8.1

For important advice on "housecleaning" FEP files, see the section "Saving Previous FEP Kernels and FLOD Files".

Clarification on Installing NFS

In Genera 8.1, the NFS system is obsolete and should not be loaded.

Instead, users should load NFS Client, NFS Documentation, and/or NFS Server as needed.

Clarifications on Printers

If you are using a 3600-family machine, you must have the systems RPC, Embedding-Support, and UX-Support loaded before you can use a printer spooled from a UNIX machine.

When installing an LGP2 or LGP3, the Other Options field in Interface Options should include the following:

```
NUMBER-OF-DATA-BITS 8 PARITY :NONE XON-XOFF-PROTOCOL YES
```

This is particularly important for printers connected to XL-family machines. This setting must be overridden if you are *not* using an Apple LaserWriter or LaserWriter II.

Clarification on Installing the X Windows Client Facility

The X Windows client includes the systems CLX and X-Remote-Screen. To load the client facility, load the system X-Remote-Screen.

Note also that before you load X-Remote-Screen, it is necessary that RPC, Embedding-Support, and IP-TCP are loaded in your world. To check, use the Show Herald command. If any of those systems are not loaded in your world, load them before loading X-Remote-Screen.

Clarification on RPC Examples

In Genera 8.0, we distributed an example in the file SYS:EMBEDDING;RPC;EXAMPLES;UNIX-APPLICATION-EXAMPLE.C. That example had one error in it. The file distributed in Genera 8.1 is corrected.

In the file SYS:EMBEDDING;RPC;EXAMPLES;UNIX-APPLICATION-EXAMPLE.C the following line:

```
if (tcp_agent_open(host) < 0) {
```

was changed to:

```
if (tcp_agent_open(host, IPPORT_RPC) != 0) {
```

Clarification on Life Examples in Program Development Tutorial

The Genera 8.0 documentation included an example of the Life program, which was first done in Common Lisp, and then extended in various ways. See the section "Developing a Simple Common Lisp Program: Life".

The documentation stated that the code was available in the file SYS:EXAMPLES;COMMON-LISP-LIFE.LISP. Unfortunately, in Genera 8.0 the example file was not distributed.

In Genera 8.1, that file and some related files are distributed. The related files contain code that extends the original example, as described in the documentation. The pathnames are:

```
SYS:EXAMPLES;COMMON-LISP-LIFE.LISP
SYS:EXAMPLES;COMMON-LISP-LIFE-WITH-GRAPHICS.LISP
SYS:EXAMPLES;COMMON-LISP-LIFE-WITH-COMMANDS.LISP
SYS:EXAMPLES;COMMON-LISP-LIFE-WITH-PROGRAM-FRAMEWORK.LISP
```

Note on the Amount of Available Memory Reported by MacIvory

On all MacIvory models, the amount of memory displayed is less than the amount based on available physical memory. Some of this discrepancy is attributable to the boot ROM, some to the communications area, and some to the memory architecture of the MacIvory model involved.

If you compute how much memory should be available on a MacIvory model 1 or MacIvory model 2 with 16MB of memory, your answer will be 2730K words. When you actually look at this configuration, however, you find that it only has 2688K words or 16128K bytes. When you add the size of the boot ROM (64K bytes) and communications area (128K bytes), you end up with 16320K bytes which is 64K bytes short of what's actually available in 16MB. The missing 64K bytes simply is not usable due the memory system architecture on these machines.

A MacIvory model 3 has 2MW (2048KW), 4MW (4096KW), 6MW (6144KW), or 8MW (8192KW) of physical memory. In all cases, 98K words of physical memory will be dedicated to the boot ROM and communications area. Consequently, the actual memory size will display as 1.9MW (1950KW), 3.9MW (3998KW), 5.9MW (6046KW), or 7.9MW (8094KW).

Documentation Update: Static Runtime

Overview of Static Runtime

Static is an object-oriented database system for the Genera programming environment. Static provides client programs with *persistent, shared* storage of information. Persistent information stored in Static exists outside and beyond the boundaries of the Lisp world that created it, and is protected against failure. Shared information is shared by distinct Lisp worlds on different workstations, for writing as well as reading.

The tools to develop Static applications are available in the Static Developer product, which is a layered product. The Static Developer product includes documentation of how to develop Static applications.

Static Runtime is a separately loadable system. To load Static Runtime, type `:Load System Static-Runtime`. Static Runtime enables Genera users to use, access, and maintain a Static database. The benefit of making Static Runtime available to Genera users is to make it easier and more practical to deliver Static-based applications; customers of Static-based applications do not need to buy Static Developer.

Static Runtime is a subset of Static Developer. The functions and macros which enable programmers to develop Static applications are not present in Static Runtime. Static Runtime is documented here.

Operations and Maintenance of Static Databases

The Architecture of Static

Using Static Locally or Remotely

Static can be used on a single host, or between many hosts across the network. This section describes how this works, and explains the terminology used for the participants and the roles they play.

A *host* is a computer, a workstation. A *site* is a collection of hosts (and other things, such as users), all at more or less the same physical place. Every host is at one particular site. We assume that every host at a site is connected to a network, so that each host can communicate with every other host. Hosts, sites, and networks are all described by the namespace database. For more information about the namespace database and the things it describes: See the section "Concepts of Symbolics Networks". See the section "Setting Up and Maintaining the Namespace Database".

A *Static File System* is a file system that holds Static databases. Every Static File System is at one particular site. Every Static File System resides on one particular host at that site. Each Static File System is described in the namespace

database by a File System namespace object: See the section "How a Static File System is Described in the Namespace". See the section "Attributes for Objects of Type "File System"".

Suppose that at some site there are two hosts named Mars and Venus, and there are two Static File Systems named Rose and Iris. Rose resides on host Mars, and Iris resides on host Venus. Fig. 1 shows hosts represented as rectangles, and Static File Systems represented as circles.

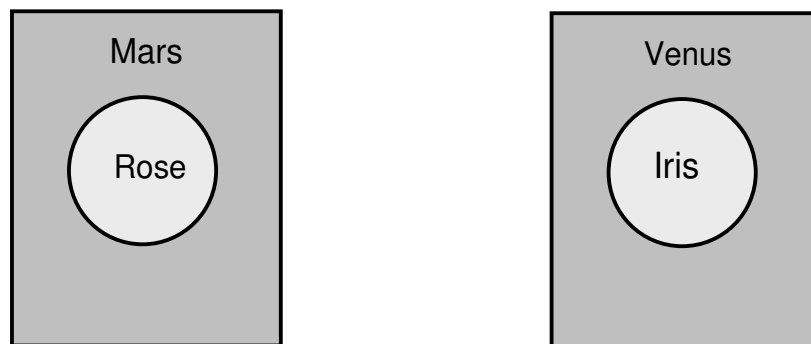


Figure 1. Hosts Mars and Venus, with File Systems Rose and Iris

Now, suppose a process running on host Mars begins using Static, doing **static:with-database** and **static:with-transaction**, calling accessor functions, and so forth. This process might be a Dynamic Lisp Listener, a process associated with some program defined by **dw:define-program-framework**, or any process at all. A process that calls Static functions and special forms is a *client* process.

If the client process uses a database that resides in the Static File System named Rose, Static notices that Rose is on the same host as the client process itself. We say that the client process is using Static *locally*, or that the client is accessing a *local database*. When a client is using Static locally, the client manipulates the database directly, invoking the disk driver to access the host's disks, etc.

If the client process uses a database that resides in the Static File System named Iris, Static notices that the Iris is on some other host than the client process. We say that the client process is using Static *remotely*, or that the client process is accessing a *remote database*. When a client uses Static remotely, a *server process* is created on the *remote host*, and a network connection is created to allow the client process and the server process to communicate. The client process cannot directly access the disks of another host, and so it delegates this work to the server process.

If a second client process, running on host Venus, accesses the same database on Iris, this second client is using Static locally. In Fig 2, we have two client processes, both using the same database, one locally and one remotely. In general, there might be any number of client processes accessing a database, many locally and many remotely.

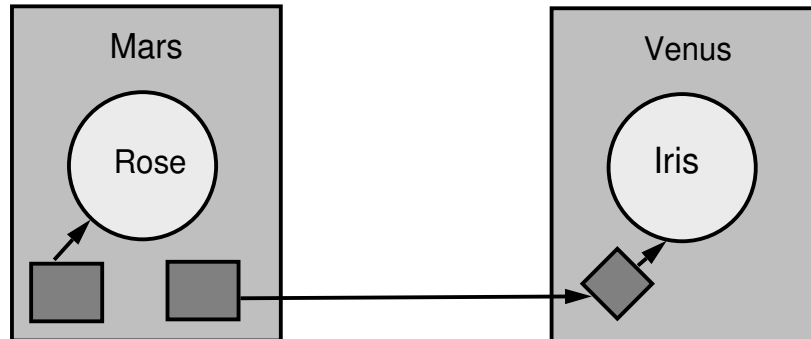


Figure 2. Local and Remote Use of Static

How a Static File System is Described in the Namespace

Every Static File System is described by an object in the namespace database. For more information about the namespace database and the things it describes: See the section "Concepts of Symbolics Networks". See the section "Setting Up and Maintaining the Namespace Database".

The namespace object for a Static File System is of type File-System (as opposed to Host, Network, User, et. al.), and would typically look similar to this:

```
Host: CHICOPEE
Type: DBFS
Root Directory: FEP1:>Static>iris.UFD
Pretty Name: IRIS
User Property: PARTITION0 FEP1:>Static>iris-part0.file.newest
User Property: PARTITION1 FEP1:>Static>iris-part1.file.newest
User Property: PARTITION2 FEP1:>Static>iris-part2.file.newest
User Property: LOG-DESCRIPTOR-FILE-ID 1015371-1311996-1048993
User Property: DBFS-DIR-ROOT-FILE-ID 1014172-1311996-1048993
```

When you use the Create Static File System command, Static automatically creates a file-system object. See the section "Create Static File System Command".

The most important attribute is the Host, which says where this file system lives. In this example, Iris lives on host Mars.

The Type field always has the value DBFS. Any other values for this field are reserved for future expansion.

The Root Directory field contains a pathname of a FEP FS file. That file contains the directory of the Static File System. The pathname should always start with FEPn: and end with the .UFD file extension.

The Pretty Name and Short Name mean the same thing as they do in other namespace objects. The Short Name can be used on input as an abbreviation; in particular, you can use it in pathnames. The pretty name is used to display the name of the file system.

The User Properties named PARTITION have values that are pathnames of the FEP files which are the partitions that make up the file system. The database is stored in a number of partitions. For more information on partitions: See the section "Create Static File System Command".

The User Property named LOG-DESCRIPTOR-FILE-ID is the unique ID of Static's "log descriptor" file, an internal file used to store various per-file-system information.

The User Property named DBFS-DIR-ROOT-FILE-ID contains, in the form of a string, the internal unique ID of the special database in the file system that stores the hierarchical directory structure of the file system. This is established when the Static File System is created, and you should never change it.

For reference documentation on the File-System object and its attributes: See the section "Attributes for Objects of Type "File System"".

Why is there a separate File-System namespace object? Why doesn't Static use the host object, as LMFS does? There are three reasons:

- Pathnames starting with "MARS:" already mean "the LMFS found on host MARS". They can't also mean a Static File System. We must have a different name in order to specify the Static File System, because the name "MARS" has already been taken. This is the most compelling reason.
- It's possible to have more than one Static File System on the same host, each with its own name, using different areas of the host's disks. Such a configuration might be desirable for various administrative reasons.
- You can move a Static File System from one host to another, using removable disk packs, magnetic tape copies, or moving disks. If you do this, you need to change the file-system namespace object for the Static File System to refer to the new host. All software that refers to the Static File System by name continues to work, because the file-system namespace object provides the link to the new location of the Static File System.

Static Database Pathnames

Many Static commands take a pathname as an argument, indicating a database. This should be a *database pathname*. The most striking difference between a database pathname and an ordinary pathname is the first component: for a database pathname, this is a Static File System; for an ordinary pathname, this is a host.

Every Static File System contains a set of files. Each file contains a Static database, and is named by a database pathname.

Database pathnames resemble LMFS pathnames. The "host name" part is the name of the Static File System, followed by a colon. After that are the names of the directories, separated by greater-than characters. The last part of the pathname is the name of the file. For example, the pathname **Iris:>george>financial>ledger**

names a file in the Static File System named Iris. The root directory on Iris contains a directory named **george**, which in turn contains a directory named **financial**, which in turn contains a file named **ledger**.

There are some important differences between database pathnames and LMFS pathnames. Database pathnames have a name, but no type or version. Static File System directories store fewer properties than LMFS directories; there is no modification date, reference date, do-not-reap flag, and so on. There are also no links, only files and directories. Static File System directories support the following properties, which can be examined with directory listings (see **fs:directory-list** or **fs:file-properties**).

:author	The user ID of the creator of the file.
:creation-date	The date and time at which the file was created, expressed as a universal time.
:comment	An arbitrary string that appears in directory listings.
:directory	t if this is a directory, nil if this is a file.
:length-in-blocks	The length of the file, in blocks. A block is 1152 (decimal) bytes, the size of a FEP file system block. (FEP blocks are 1152 bytes on 3600-family machines, and 1280 bytes on Ivory machines.)

The **:author**, **:creation-date**, and **:comment** properties can be set using **fs:change-file-properties**. **:comment** can be set for files but not directories. The other properties cannot be set.

Database pathnames are case-insensitive for lookup, like those of LMFS. When you make a new file, the Static File System directory remembers the case you used and stores this in the directory. To look up a file that already exists, you can use either case for any character. A directory cannot have one file named **Foo** and another named **foo**; these are considered to be the same name.

Database pathnames support relative pathname syntax, wildcard syntax, and completion, just like LMFS pathnames. **<foo>bar** means the file named bar in the directory named foo in the directory that is the parent of the default pathname's directory. **>foo>*** means all the files in the directory named foo. **>foo>***>*** means all the files in the directory named foo and its descendants. **>foo>b*** means all the files whose names start with **b** in the directory named **foo**.

There is no undeletion, and no expunging. When you delete a file, it is immediately and permanently removed.

Files cannot be opened by the Lisp **open** function because they are not really files in the sense of the Lisp stream system. The contents of files are accessed only via Static operations, never by streams. The only exception is if they are opened with a **:direction** of **:probe**, **:probe-directory**, or **:probe-link**. This lets programs use **probe-file** to check for the existence of files in a Static File System.

Database pathnames work correctly with the Genera directory manipulation tools, such as Dired, FSEdit (the File System Editor), and commands such as Show Di-

rectory, Create Directory, Delete File, and Rename File. However, they do not work with commands that attempt to open files, such as Copy File and Show File.

The root directory of every Static File System directory contains an entry named **Directory**, which refers to the directory itself. This gives you a way to name the root directory as a file, in order to perform operations on it. This is rarely necessary, and you can usually just ignore the **Directory** entry.

Dealing with Databases by Their Pathnames

Many operations on databases can be done by using normal file commands on the database pathname. See the section "Static Database Pathnames".

- What databases are stored in a directory of a given Static file system? Use the Show Directory command:

```
Show Directory beet:>fred>*
```

- What are all the databases stored an entire Static file system? Use the Show Directory command:

```
Show Directory beet:>*>*
```

- How can I rename a database? Use the Rename File command.

```
Rename File beet:>university beet:>harvard
```

- How can I remove a database? Use the Delete File command on a database pathname.

```
Delete File beet:>university
```

Although this method permanently removes the file, you can restore the file from backup tapes (if you have any). See the section "Selective Restore Command".

Services and Protocols Used by Static

Static uses several network services and protocols. The commands that install Static at your site add new information to your namespace host objects, indicating that these services and protocols are supported. In this section, we briefly describe the new namespace information; this information is not required in order to write Static programs, but it might be useful to help you debug any unusual namespace-related problems.

DBFS-PAGE Service

DBFS-PAGE is the network service provided by a Static server for the benefit of Static clients. When a Static client first accesses a particular Static server host, it invokes the DBFS-PAGE service on that host. From then on, this client uses

this connection to communicate with that host, even if it accesses more than one Static File System on that host.

When the transaction ends, the connection is returned to a free pool, so subsequent transactions can use that connection. This helps reduce the amount of time spent opening network connections.

Static servers use the same connection scavenger mechanism used by General file servers, so that if a connection hasn't been used for a long time, the server process is killed and the connection goes away. See the section "The File Control Lifetime Host Attribute".

The namespace entries for the DBFS-PAGE service should be present in the host object of every host used as a Static File System server. The Add DBFS Page Service command sets up this service entry in the namespace database. See the section "Add DBFS PAGE Service Command".

See the section "Using Static for the First Time".

DBFS-Page Protocol

DBFS-PAGE is the name of a network protocol that implements the DBFS-PAGE service. This protocol is built on top of the byte-stream-with-mark network medium. See the section "BYTE-STREAM-WITH-MARK Network Medium". It can be used with Chaosnet or TCP/IP networks. For Chaosnet, the contact name is "DBFS-PAGE"; for TCP/IP, the port number is 569.

ASYNCH-DBFS-PAGE Service

ASYNCH-DBFS-PAGE works in the reverse direction: the Static server uses this service to form a connection back to Static File System client hosts. The server uses this connection to notify the user when pages have been modified by another client; the client acts on this information by invalidating its cache.

The service that ASYNCH-DBFS-PAGE provides is not necessary for correct operation of Static. If the server finds that it cannot form a connection to the client, it simply gives up and tries again later. The cache coherency protocol within Static makes sure that invalid data is never used. However, Static will be more efficient if ASYNCH-DBFS-PAGE is working properly. You should try to make sure that all hosts that use Static as a client have the proper namespace entries for ASYNCH-DBFS-PAGE in the namespace database. The command Add ASYNCH DBFS PAGE Service should be run on each client host: See the section "Using Static for the First Time".

ASYNCH-DBFS-PAGE Protocol

ASYNCH-DBFS-PAGE is the name of the network protocol that implements the ASYNCH-DBFS-PAGE service. This protocol is built on top of the byte-stream-with-mark network medium. See the section "BYTE-STREAM-WITH-MARK Network Medium". It can be used with Chaosnet or TCP/IP networks. For Chaosnet, the contact name is "ASYNCH-DBFS-PAGE"; for TCP/IP, the port number is 568.

Attributes for Objects of Type "File System"

Host

Specifies the host that the file system resides on; a host object (required).

Host: MARS

Type

Must always be DBFS (required). Other values are reserved for future expansion.

Type: DBFS

Root Directory

Specifies a pathname of a FEPFS file. That file contains the directory of the Static File System. The pathname should always start with FEPn: and end with the .UFD file extension.

Root Directory: FEP1:>Iris.UFD

Pretty Name

Specifies a name for the file-system to use when showing the name; a token (required).

Pretty Name: Iris

Nickname

Specifies alternate names for the network; a set of names. The file system may be found by these names.

Nickname: IRE

Short Name

Specifies additional nicknames; a set of names. A short-name is used when a program wants to display a host's name without

using up too much space. A short-name is used for both input and output. This is also used in the printed representation of pathnames.

Short Name: I

User Property

User-Property All objects contained within the namespace (hosts, sites, namespaces, printers, and users) are eligible to have a User-Property attribute. It consists of a pair whose first element is an indicator (like that of a property list) and whose second element is a token. The User-Property attribute holds any information that users choose to associate with an object. For example:

User-Property: ID-number 123-45-6789

Stattice automatically places several user properties into file-system objects. The User Properties named PARTITION have values that are pathnames of the FEP files which are the partitions that make up the file system. The database is stored in a number of partitions. For more information on partitions: See the section "Create Stattice File System Command".

The User Property named LOG-DESCRIPTOR-FILE-ID is the unique ID of Stattice's "log descriptor" file, an internal file used to store various per-file-system information.

The User Property named DBFS-DIR-ROOT-FILE-ID contains, in the form of a string, the internal unique ID of the special database in the file system that stores the hierarchical directory structure of the file system. This is established when the Stattice File System is created, and you should never change it.

FEP File for Generating Stattice Unique IDs

When you first run Stattice on a machine, it automatically creates a small file (2 blocks) in the FEP file system, whose name is:

FEP n :>UNIQUE-ID.FEP.1

where n is the lowest fixed-medium disk unit, or the lowest disk unit if all units are removable-medium. (In almost all configurations, n is zero.)

This file is used internally by Stattice to generate unique IDs. We recommend that you leave it there, and don't delete it. If you do delete it, Stattice will re-create it next time Stattice is run.

Static File System Operations Program

The Static File System Operations program is an interactive utility for maintaining and manipulating Static File Systems. It serves primarily as the user interface to the backup system. It also provides commands for enabling and shutting down a Static File System, and other functions.

We first present several sections that give background information, and then describe how to use the program itself, in the section "Using the Static File System Operations Program".

Some operations on databases can be done by using normal file commands on the database pathname. For details: See the section "Dealing with Databases by Their Pathnames".

Overview of the Static Backup Facilities

The Static File System backup facilities let you make backup copies of the databases stored in a Static File System, onto another medium. If the disk holding the Static File System is damaged or destroyed, the copy of the Static File System can be restored onto a fresh disk.

It is very important to back up your Static File System on a regular basis. Disks are fragile and subject to failure. Doing backups is the only way to protect your data against disk failures.

The backup facilities copy database information to tertiary storage media. Currently, industry-standard magnetic tape and cartridge tapes are supported. The software names and catalogs the media into groups called volume sets.

Two forms of backup are provided: complete backups, and continuous archive backup. (Currently only complete backup is supported.) Complete backup makes a complete copy of a database file system onto tertiary storage, and assures the copy is transaction-consistent. If the database file system is destroyed, you can restore the copy, losing only changes made since the latest backup copy was produced. Archive logging continuously copies all database changes to tertiary storage. If the database file system is destroyed, you can restore the latest complete backup copy, and then replay the archive changes, so that no information is lost.

A backup tape from a Static File System stored on a 3600-family Static server cannot be reloaded into an Ivory Static File System, and vice versa. Also, if you want to move whole databases between file systems of different block size, you have to use the high-level dumper (the Dump Database and Load Database commands) to convert the data into a text file, and then move the text file.

See the section "High-level Dumper/Loader of Static Databases".

Kinds of Tertiary Storage

Backup copies are kept on *tertiary* storage. (Primary storage is main memory, and secondary storage is disk.) The Static File System backup facilities back up to and restore using a *generic tertiary storage protocol*, so that different kinds of ter-

tiary storage can be used interchangeably, and new kinds can be added in the future.

With Symbolics 36xx systems, two kinds of tertiary storage are currently supported: 1/4-inch cartridge tapes, and 1/2-inch industry-standard reel-to-reel magnetic tapes. (The fraction of an inch refers to the width of the tape itself.) Both kinds of tapes can be used either locally or remotely. In local usage, the tape drive hardware is physically connected to the workstation doing the backup or restore. In remote usage, the tape drive hardware is connected to some other computer, which communicates with the workstation over the network.

In future releases, we we plan to support write-once optical disks as another tertiary storage medium. We also intend to support other formats of magnetic tape as hardware support becomes available.

The physical integrity of backup copies is very important. If it's necessary to restore a file system from a backup copy, it is imperative that the data be readable from the copy. Unfortunately, magnetic tapes are an imperfect physical medium. Periodically, tapes are found to be unreadable, or to contain data errors.

To protect against problems with tapes, the backup system always writes data using a powerful error-correcting coding technique, a simple version of Youngquist's algorithm. The technique is effective at recovering from large damage spots on tape; this is important, since it is not uncommon for 100 sequential bits to "drop out". The cost of the algorithm is that approximately 3/2 as much tape is required to store the same amount of information. We believe that the protection is worth the cost of the extra tape. We tested this coding technique by scratching a tape with a razor, and the data were still recovered.

If you have a choice between industry-standard and cartridge tape, industry-standard tape is preferable, because:

- Industry-standard tape runs faster than cartridge tape.
- More data fits on a single industry-standard tape than on a cartridge tape, so you don't have to change tapes as often.

The primary advantage of cartridge tape is its lower cost. Every Symbolics site has at least one cartridge tape drive, because cartridge tape is used for distributing software.

The benefits of using an industry-standard tape drive increase if you have large Static File Systems, or if you write new data frequently.

Choosing the Kind of Tertiary Storage to Use

If you have a choice between local and remote usage, local usage is preferable, because:

- Network connections are inherently unreliable. It is inconvenient to have a network connection fail during a backup or restore operation.

- Local usage runs faster than remote usage.

Therefore, we recommend that you put your Static File System on a workstation that has its own tape drive, if possible.

When a command of the Static File System Operations program prompts for a "device specification", it wants to know which tape drive to use. (The prompt doesn't say "tape drive" because there will be other kinds of tertiary storage in the future.) The default is to use the cartridge tape drive on the local workstation. You can change the host, to use a tape drive on another computer, and you can change the device type to industry-standard tape.

If you select industry-standard tape, the prompt expands to let you specify a unit number and a density. The unit number parameter is provided because it is possible to attach more than one industry standard tape drive to a computer; the unit number distinguishes which one you want. The default value is zero, and if there is only one tape drive, it should be unit zero.

Industry standard tapes can be written at several different densities, measured in bits per inch. We support 1600, 3200, and 6250 bits per inch, defaulting to 3200. Not every tape drive can read and write at every density! You must check the hardware you intend to use, and determine what densities it supports.

Volume Capacity

When you make a backup copy, you don't need to know in advance how many volumes will be needed to hold the copy. Whenever the dumper reaches the end of one volume, it asks you to mount another volume, until the copy is completed.

However, you might want to be able to estimate, in advance, how many tapes will be needed to hold a copy. Here is some information to help you make such an estimate. (There is no requirement that you make such an estimate, but it is sometimes desirable.)

It's difficult to make an accurate estimate of how many volumes are needed for a backup copy, because the number depends on many factors. More volumes are needed if there are many small files than one large file. Different tape drives have different characteristics; industry-standard tape drives don't all write interrecord gaps the same way, and cartridge tape drives are affected by the quality of the tape medium and the cleanliness of the heads. So the following numbers should be treated as approximate figures.

The numbers below show the actual data capacity for several different kinds of tape. The actual data capacity is smaller than the raw capacity primarily because of the overhead of the error-correcting coding used in Static File System backup tapes. Capacities are given in megabytes.

DC300XL/P cartridge tape: 30 MB

DC600A or DC600XTD cartridge tape: 40 MB

600-foot industry standard tape at 3200 bpi: 14 MB

2400-foot industry standard tape at 3200 bpi: 60 MB

Industry-standard tapes can be written at different densities; the numbers above assume a density of 3200 bpi. If you are using 1600 bpi, divide the numbers by two; if you are using 6250 bpi, double the numbers. Similarly, if you use a reel of tape with some other length, apply the appropriate ratio.

The sizes of the databases in a Static File System are shown in Static File System directory listings, in blocks. A block is 1152 bytes. So, for example, if you had a Static File System containing four databases, two 1000 blocks long and two 2000 blocks long, the total amount of data is about 6 MB, which will easily fit on one tape of any kind.

Tertiary Volumes and Volume Sets

A *volume* means one physical tertiary storage medium, such as one reel or one cartridge of tape. (In future releases, a single write-once optical disk will also be referred to as a volume.)

A *volume set* is a set of one or more volumes. Volumes are grouped into sets because each volume is of a fixed size, whereas you can keep expanding the size of a volume set by adding more volumes.

Each complete dump of a database file system is put on its own volume set. The number of volumes in the volume set depends on the size of the database file system. If the Static File System is not very large, a complete dump fits on a single volume, which constitutes a single volume set.

Each volume set has a *name*, and each volume within the volume set has a *sequence number*. The name is a character string. No two volume sets can have the same name. Names are compared ignoring case, so you must not have one volume named "Foo" and another named "foo". Volume set names may not use character styles or non-standard character sets. The first volume of a volume set should be numbered 1, and each sequential volume is assigned the next number.

When you're using the backup system and a tape is being mounted, you are prompted for a "mount specification". This consists of a volume set name, a sequence number, and a device specification. It means that you are mounting a particular volume on a particular device. To specify the device, you are prompted for a host name, a device type, and further parameters depending on the type of the device. For details of device specs: See the section "Choosing the Kind of Tertiary Storage to Use".

Labels on Volumes

When a volume is being used by the Static File System backup system, some special identification information is written at the front of the volume, called the *label*. The label includes the volume set name and the sequence number, which uniquely identify the volume. The backup software uses the label to help assure that you have mounted the tape you intended to mount, in order to guard against mistakes.

When you first use a brand-new tape for Static File System backup, you should write a label on it, by using the Initialize Backup Volume command in the Static File System Operations program. Be careful to only use this command on fresh, unused tapes, because it will destroy any information already on the tape.

When the backup system is making a backup copy, and it is ready to write onto a new tape, it first reads the label of the tape, to make sure that this tape is the right one. For example, suppose you are making a backup copy to a volume set named FULL0013, and the backup system just finished writing volume number 2 of the volume. It prompts you with the message:

End of volume FULL0013/2. Enter mount specifications for the next volume:

You enter a mount specification, in which you enter a volume set name and sequence number. The default is volume set FULL0013, sequence number 3. You also physically mount the corresponding tape on the device that you specify in the mount specification. After you click on Done or press End, the backup system reads the volume label, to make sure that this volume is really volume number 3 of volume set FULL0013. If the label is present and contains what backup expects, the backup copy continues. Otherwise, you are given several options:

Accept the information on the volume

(You believe you entered the wrong thing, and you want to use what the tape says.) This choice tells the backup system to use the volume set name and sequence number that were found in the label on the tape, even though they aren't what we originally expected. This sets the backup system's concept of "current volume", so the backup system will expect the next volume to follow this one.

Remount a different volume

(You believe you mounted the wrong tape, and wish to try mounting a different one.) This choice lets you remove this volume from the drive and try another one. The backup system prompts you with "Is the desired volume mounted?" When you answer "y", it proceeds to read the label of the new tape and check it.

Reenter different volume specifications

(You believe you entered the wrong thing, and you want to try to enter it again.) This choice makes the backup system return to the point where you were prompted for mount specifications for this tape; you are prompted again. Use this if the reason for the problem is that you didn't provide the right mount specification.

Overwrite the volume with the specified information

(You believe that what the tape says is wrong, and you want to use what you entered.) This choice tells the backup system to ignore what the label says, and write the tape using the mount specification that you provided. This overwrites the label and can change the volume set name and sequence number in the label.

Abort the current operation

(You don't want to proceed further.) This choice returns you to the Static File System Operations program.

If the tape you mount is a fresh, unused tape, you'll get this list of choices, and you can select [Overwrite] to write a new label on the tape. So it is not actually necessary to use the Initialize Backup Volume command to write an initial label; [Overwrite] also does it for you. However, there's a drawback to relying on [Overwrite]. When the backup system tries to read the label of the blank tape, the tape drive attempts to read data from the tape, but cannot find any. Some drives react to this lack of data by reading further and further down the tape, hoping to find data eventually, which can take a long time. So if you don't run Initialize Backup Volume on new tapes, backup copying might be substantially slower.

This behavior is part of the tape drive and tape controller, and cannot be modified by software. Currently, all industry-standard tape drives sold by Symbolics exhibit this behavior, but the cartridge tape drives do not. Therefore, it is particularly time-saving to use Initialize Backup Volume on industry-standard tapes.

Volume Libraries

When you do Static File System operations, Static maintains a database called the volume library. This database is stored within the file system. The volume library stores the following information:

For every backup volume that holds information from this file system:

- Volume name and sequence number
- Completion date, which is the date and time this tape was last written
- Type, which is either "Industry Standard Tape" or "Cartridge Tape"
- The tape spec that was used when the tape was last written, which includes the host, and also unit and density for industry-standard tapes

For every backup run (that is, for every time that a dump is made):

- Completion date, which is the date and time this run was performed
- The set of volumes that were written.
- Whether the run is "valid", or whether something went wrong during the dump

For every file that has been dumped:

- Name of the file
- The set of backup-notes (see below)

For every copy of a file that exists on tape, a "backup note", consisting of:

- The file attributes (length, creation-date, author, comment)
- Which volume the copy resides on
- Which backup run this copy was part of

Using the Static File System Operations Program

Before using the Static File System Operations program, you must load it by entering:

```
Load System DBFS-Utilities
```

You enter the Static File System Operations program by entering:

```
SELECT symbol-sh-D
```

The commands appear a menu in the top pane. You can click on them, or type the names of the commands. Typically, an AVV menu will be displayed, prompting you for several fields. When you finish entering the information and press END, the command is executed.

We summarize the commands here. For complete documentation on each command: See the section "Dictionary of Static File System Operations Commands".

Backing Up and Restoring a File System

The most common operations are Complete Backup and Complete Restore. Selective Restore can also be used, to restore one or more databases from tape to the file system.

Complete Backup Command

Copies a Static File System (and all databases in it) to tape.

Complete Restore Command

Copies a file system (and all databases in it) from a tape to a Static File System.

Selective Restore Command

Copies selected databases from a tape to a Static File System.

Initialize Backup Volume Command

Writes the volume number on the tape label itself.

Getting Information

Describe Static File System Command

Displays information about a file system.

Describe Backup Volume Command

Displays information about a backup volume stored on a tape.

Show Backup History Command

Displays information about all backup runs done on a file system.

Compare Backup Volume Set Command

Compares a file system stored on tape with a file system stored on the local machine.

Enabling and Disabling a File System or All of Static

Enable Static File System Command

Enables a file system: allows transactions to be started.

Disable Static File System Command

Disables a file system: aborts any transactions in progress and disallows any transactions to occur until the file system is enabled again.

Enable Static Command

Re-enables Static activities.

Disable Static Command

Entirely disables all Static activities.

File System Manipulation

Create Static File System Command

Creates a new Static File System on the local host.

Delete Static File System Command

Expunges an entire Static file system, and removes all traces of it, including every database in it; this is a very dangerous command.

Show All Static File Systems Command

Lists all the file system objects and the host that they reside on in a namespace.

Add Static Partition Command

Adds a new partition to an existing Static file system.

Show Static Partitions Command

Lists the partitions of a Static file system, and shows the amount of free space remaining in each partition.

Dictionary of Static File System Operations Commands

This section documents the commands that are available only in the Static File System Operations program.

The following commands are available from the Static File System Operations program, but are described elsewhere because they are also available at top level.

- "Add Static Partition Command"
- "Create Static File System Command"
- "Delete Static File System Command"
- "Show All Static File Systems Command"
- "Show Static Partitions Command"

For documentation on the other Static commands that can be used at top level: See the section "Dictionary of Static Commands".

Complete Backup Command

Complete Backup

Copies all databases in a Static File System to tape. This command is available in the Static File System Operations Program.

The AVV menu looks like this:

```
Enter complete backup parameters (volume name required):
```

```
File system to backup: DLW-UFS-3
Volume set name: abc
Volume sequence number: 1
Device: Industry-Standard-Tape Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes No
```

The file system to backup must be stored on the local host. We discuss volume set names and sequence numbers elsewhere: See the section "Labels on Volumes".

The volume host is the host that will write the data to tape; it need not be the local host. If that host does not have TAPE service in its namespace object, you will get this error:

```
Error: Host does not support TAPE service.
```

You can use one of the proceed options to try TAPE service on various mediums such as TCP or CHAOS. You can also edit the host's namespace object to add that service entry, so the error won't happen again.

Symbolics recommends RTAPE via TCP rather than RTAPE via ChAOS (TCP is more reliable). Add the service TAPE TCP RTAPE to the host's namespace object.

If you are writing to a blank tape, you will get an error stating that the volume set name does not match that of the tape itself. This is to be expected (because the tape doesn't have a volume set name at all yet). One way to prevent this error

is to use the Initialize Backup Volume command before doing Complete Backup, to write the volume set name on the tape. In any case, one of the choices is to Overwrite the tape, which is the correct choice for a blank tape.

Compare Backup Volume Set Command

Compare Backup Volume Set

Compares a file system stored on tape with a file system stored on the local machine. This command is available in the Staticce File System Operations Program.

The AVV menu looks like this:

```
Enter specifications for compare:

File system to compare: DLW-UFS-3
Volume set name: abc
Volume sequence number: 1
Device: Industry-Standard-Tape Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes No
```

If anyone has made changes to a database in the file system since the backup was done, Staticce will report how many pages are different. The output looks like this:

```
File SCRC|DLW-UFS-3:>Volume-Library>%volume-library has 21 pages
different from the tertiary image.
These differences could be caused by updates which occurred
between the time the backup volume finished writing and the time
the comparison finished.
```

Complete Restore Command

Complete Restore

Copies all databases from a tape to a Staticce File System. This command is available in the Staticce File System Operations Program.

The AVV menu looks like this:

```
Enter specifications for restore:

File system to restore: DLW-UFS-3
Volume set name: abc
Volume sequence number: 1
Device: Industry-Standard-Tape Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes No
```

The choices are the same as for the Complete Backup command: See the section "Complete Backup Command".

The first thing that a Complete Restore does is delete all databases in the file system. It then copies the databases from tape to the file system. Note that if you are writing to a file system that already exists, you will get this message:

```
Before a file system can be restored, the existing files must be
destroyed. All the files in file system SCRC|DLW-UFS-3 are
about to be destroyed. Are you sure you want to continue? (Yes
or No)
```

If you want to keep the existing file system, choose No. If you want to restore the file system from tape, choose Yes.

Describe Backup Volume Command

Describe Backup Volume

Displays information about a backup volume stored on a tape. This command is available in the Static File System Operations Program.

This command gets its information from the tape itself.

The AVV menu requests the type of tape (whether cartridge or industry standard), and the volume host (the host where the tape is mounted).

Describe Static File System Command

Describe Static File System *file-system-name*

Displays information about a file system. This command is available in the Static File System Operations Program.

This command gets information from the file-system namespace object. If the file system is local, it also displays information about how much data is stored in each partition.

Disable Static Command

Disable Static

Entirely disables all Static activities:

- Every currently-running transaction is aborted, signalling the error **dbfs::system-shutdown-transaction-abort**, whose error message is "Transaction aborted due to a system shutdown."
- All server processes executing on the local host are killed.

- All local file systems are shut down. See the section "Disable Static File System Command".
- The local host is marked as disabled, so that any new file systems created on the host are automatically disabled.
- All network connections associated with Static services are killed.

Any attempts to start a transaction while Static is disabled signal an error. Any attempts by a remote host to connect to this host are rejected. This command is available in the Static File System Operations Program.

Disable Static File System Command

Disable Static File System *file-system-name*

Disables a file system: aborts any transaction in progress that owns a lock on any page of any file in the file system. Disallows any transactions to occur until the file system is enabled again. If any transaction attempts to use a database in the file system, the error **dbfs::file-system-disabled** is signalled. This command is useful if you want to perform administrative activities on a database, and want to make sure nobody else accesses the database. Note that it is not necessary to disable a Static file system to do a backup dump.

file-system-name must be the name of a Static file system on the local host. This command is available in the Static File System Operations Program.

Enable Static Command

Enable Static

Re-enables Static activities. Undoes the effect of Disable Static, returning Static to normal. This command is available in the Static File System Operations Program.

Enable Static File System Command

Enable Static File System *file-system-name*

Enables a file system: allows transactions to be started. Undoes the effects of Disable Static File System, returning the file system to normal. This command is available in the Static File System Operations Program.

Initialize Backup Volume Command

Initialize Backup Volume

Writes the volume number on the tape label itself. This command is available in the Static File System Operations Program.

This command is not a necessary step, because Complete Backup will also write the information on the tape, but it prevents the mismatch that can occur when writing a blank tape: the volume number you specify won't match that of the tape (because a blank tape won't have any volume number on it).

Selective Restore Command

Selective Restore

Copies selected databases from a tape to a Static File System. This command is available in the Static File System Operations Program.

The AVV menu looks like this:

```

Enter specifications for selective restore:

File system: DLW-UFS-3
Disable file system: Yes No
Paths to restore: >test2, >test7, and >test8
Repatriate action: Yes No Query
Name conflict resolution action: Leave
                                Rename Existing File
                                Replace Existing File
                                Load Into Unique File
                                Query
Volume selection mode: Automatic Manual
Device: Industry-Standard-Tape Cartridge-Tape
Volume host: ALDERAAN
Show Detailed Progress: Yes No

```

If you choose Yes for "Disable file system", Static does a Disable Static File System before doing the Selective Restore, and an Enable Static File System afterwards. This option is useful if you are performing some kind of delicate operation on the file system, such as recovering from a problem, and you want to make sure that no other users make any changes in the file system while the Selective Restore is in progress. In the general case, it is not necessary to do this.

The "Paths to restore" are pathnames of databases within the file system. They should start with the greater-than sign, >. They are separated by commas. These pathnames can contain wildcards.

Once you press END, Static searches the volume library to figure out which volume the file is on. You will then be prompted to mount that volume:

```

Is volume abc/1 mounted for restoring? (Y or N)

```

When Selective Restore is searching the volume library to figure out which volume to retrieve a file from, there might be more than one volume that has this file on

it. In such a case, it chooses the volume with the most recent completion date (the date and time at which the dump was completed); that is, it uses the most recent backed-up copy.

The repatriation action should never be needed by applications programmers, so you should use the default (no) for this choice.

Show Backup History Command

Show Backup History *file-system-name*

Displays information about all backup runs done on a file system. This command is available in the Static File System Operations Program.

This command gets its information from the volume library:

Volume Libraries

When you do Static File System operations, Static maintains a database called the volume library. This database is stored within the file system. The volume library stores the following information:

For every backup volume that holds information from this file system:

- Volume name and sequence number
- Completion date, which is the date and time this tape was last written
- Type, which is either "Industry Standard Tape" or "Cartridge Tape"
- The tape spec that was used when the tape was last written, which includes the host, and also unit and density for industry-standard tapes

For every backup run (that is, for every time that a dump was made):

- Completion date, which is the date and time this run was performed
- The set of volumes that were written.
- Whether the run is "valid", or whether something went wrong during the dump

For every file that has been dumped:

- Name of the file
- The set of backup-notes (see below)

For every copy of a file that exists on tape, a "backup note", consisting of:

- The file attributes (length, creation-date, author, comment)
- Which volume the copy resides on
- Which backup run this copy was part of

This command first iterates over all the backup runs, in order of completion date (most recent first). For each run, it tells you whether the run is valid, and what the completion date is. Then it iterates over all the volumes in that backup run, sorted by completion date. For each volume, it prints out the name/sequence-number, the type, the completion date, the host, the unit (if any), and the density (if any).

High-level Dumper/Loader of Static Databases

The Dump Database and Load Database commands invoke a "high-level" database dump/load tool. High level means that it dumps and restores the data in a "source" format, rather than in a binary page format as does the dump/restore tool available from the Static Operations Menu. The high-level dump/load tool can be used for certain types of database reorganization operations.

Dump Database always dumps the database in a transaction-consistent state, because it uses one long transaction to do its job.

The high-level dumper/loader is useful for several purposes. It enables you to:

- Move a database from one place to another, over a channel that can handle only ordinary text.
- Store the contents of a database on some kind of storage medium (e.g. a particular tape format) that can handle only ordinary text.
- Edit the text file to reorganize the database.

Limitations of the High-level Dumper/Loader

- It does not dump to tape, so the size of the dump is limited to the amount of available file server disk space. Further, since the format is "source" level, it may actually take more disk space to dump a database using the Dump Database command than it does to store it.
- You shouldn't do dumping while other users are operating on the database, since it dumps everything in one large transaction (which will lock them out, or else cause the dump/restore facility to abort a transaction). Loading data back into a database can be rather slow since it does many small transactions (to avoid growing the log).
- Because of LMFS file size limitations, it may not be appropriate to dump a database to a LMFS file. This size limitation is approximately 15MB. Instead, you may have to dump to a UNIX or VMS machine with enough disk space.

Clustering is Maintained

Clustering is maintained across dumps. The actual entities may be grouped differently within the clusters, but they will all be in the same cluster that they were before.

Format of the Dump File

The dump file consists of lists which contain information about the contents of the database. At the beginning of the file is information about the real schema, and following that is information about the actual contents of the database. The format of each list is that the first element of it is a keyword symbol specifying what the list is about and the rest of the list is information specific to that type of list.

Schema information is contained in lists which begin with the `:DOMAIN`, `:RELATION`, `:COMMIT-DOMAINS`, and `:INDEX` keywords. For the most part, users should not modify these lines unless it is obvious what they mean. For example, consider the following definition, which is taken from the file `SYS:STATIC;EXAMPLES;BOOKS.LISP`:

```
(define-entity-type account ()
  ((name string :unique t :cached t :inverse account-named)
   (balance single-float :cached t)
   (type (member checking owner) :cached t)))
```

The corresponding information in the dump file looks like this:

```
(:RELATION "ACCOUNT" 1 NIL ((%"$OF" "ACCOUNT" T T NIL NIL NIL) ("TYPE"
  (CL:MEMBER BOOKS:CHECKING BOOKS:OWNER) NIL NIL NIL NIL NIL) ("BALANCE"
  CL:SINGLE-FLOAT NIL NIL NIL NIL NIL) ("NAME" STRING T NIL NIL NIL NIL)))
```

This information appears on one line in the dump file. If we wanted to change the balance attribute's data type from single-float to double-float, then we'd edit the "CL:SINGLE-FLOAT" piece of text above. Of course if you change the data type of an element like this, you'd also have to change all the data in the file, too.

After the schema information, the actual data in the file is dumped to the file using `:ENTITY` and `:RELATION-DATA` entries. The format of an `:ENTITY` entry is:

```
(:ENTITY ("ENTRY" 14352 3388287 7463818 3147232 NIL))
```

This is a list of an entity's type name (`ENTRY`), its internal record ID (14352), its unique ID (three 32-bit fixnums), and its cluster ID. For the most part, these should be of little interest to the users.

Users are more likely to be interested in the `:RELATION-DATA` entries. An account entity might look like this:

```
(:RELATION-DATA "ACCOUNT" NIL (17441 BOOKS:OWNER 200.53 "Lane"))
```

To change the value of the balance for the "Lane" account, you'd change the value 200.53 to another value. You can find out the order of the attributes of this by looking at the real-schema data at the beginning of the file. The attribute order information will be embodied in the `:RELATION` entry. For example, the `:RELATION` line above shows that the order of the attributes in the above `:RELATION-`

DATA entry are %\$OF (an internal attribute), the type attribute, the balance attribute, and the name attribute.

For more detailed information, you can read the comments in the code in the file SYS:STATICE;UTILITIES;MODEL-DUMPER.LISP.

Dictionary of Static Commands

This section documents the commands that are available in the command processor. Static also offers a set of commands that are available only in the Static file System Operations menu. For documentation on those commands: See the section "Dictionary of Static File System Operations Commands".

Add ASYNCH DBFS PAGE Service Command

Add ASYNCH DBFS PAGE Service *host-name keywords*

Updates a host's namespace object to contain the ASYNCH-DBFS-PAGE service.

keywords :TCP Not Present

 :TCP Not Present

 {Yes No} If yes, no service entry is added for the TCP medium

Static uses the ASYNCH-DBFS-PAGE service for communicating various signals and commands back to each of the client hosts, and hence should be present on all Static clients. It need not be present on Static servers however, unless they are clients to some other server.

This command adds the service-medium-protocol triplet for both the TCP and CHAOS mediums to the namespace object for the host. You should only need to perform this command once, when the file system is installed on a server. If the host does not support TCP, supply Yes to the :TCP Not Present keyword option.

Add DBFS PAGE Service Command

Add DBFS PAGE Service *host-name keywords*

Updates a host's namespace object to contain the DBFS-PAGE service.

keywords :TCP Not Present

 :TCP Not Present

 {Yes No} If yes, no service entry is added for the TCP medium.

Static uses the DBFS-PAGE service uses for communicating database pages and requests over the network, and hence should be present on all Static File System server hosts. It need not be present on client hosts, however.

This command adds the service-medium-protocol triplet for both the TCP and CHAOS mediums to the namespace object for the host. You should only need to perform this command once, when the file system is installed on a server. If the host does not support TCP, supply Yes to the :TCP Not Present keyword option.

Add Static Partition Command

Add Static Partition *file-system-name partition-pathname size*

Enables you to add partitions dynamically to a Static file system (e.g. when it is running out of space).

file-system-name Name of a Static file system that is stored on the local host.

partition-pathname Pathname of a partition; this pathname must name a FEPPFS file on the local host (although the file need not exist).

size The size of the new partition, in blocks.

This command creates the new partition, allocates the space from the size given, and makes the new partition available to Static for allocating.

This command may be given when a Static server process receives a file system full error. For example, if a server process signals the following error, you can add a new partition and resume the operation:

```
Error: The File System "Squash" is full.
```

```
(FLAVOR:METHOD UFS::FIND-FREE-BLOCKS UFS:UFS-FILE-SYSTEM-MIXIN)
```

```
proceed options...
```

```
:Add Static Partition (a file-system) SQUASH
```

```
(the pathname of a file) FEP2:>Static>SQUASH-part2.file.newest
```

```
(Size in blocks [default 1000]) 1000
```

```
Updating file-system object SCRC|SQUASH in namespace... Done
```

After adding the partition, select the proceed option that resumes the operation.

Copy Static Database Command

Copy Static Database *from-database to-database keywords*

Copies all the pages of one database to the other (possibly new database) inside a transaction.

from-database Pathname of the database to copy.

to-database Pathname of the destination, where the database should be copied.

keywords :Copy Properties, :Create Directories, :Query

:Copy Properties {any combination of: Author, Comments, Creation Date} This indicates which properties should be copied to the new database(s). The default is Author and Creation Date.

:Create Directories {Yes, Error, Query} Yes means that directories that do not exist should be created silently, Query will ask, and Error will cause an error if they do not exist.

:Query {Yes, No, Ask} Whether to ask before copying each file.

Create Static File System Command

Create Static File System *file-system-name keywords*

Creates a new Static File System on the local host. You cannot use this command to create a file system on a remote host.

file-system-name A symbol naming the new file system.

keywords :Locally

:Locally {Yes, No} Whether to update only local namespace information (Yes), or to update the namespace database server as well (No). The default is No. See the section "The Locally Namespace Editor Command".

The command displays an AVV menu in which you specify the names of various parameters. Above the menu, you will see a list of all the disk drives on the local host and the amount of free space available on each of them. The AVV menu asks for the following items:

Directory Partition: This entry specifies the FEPFS file in which the internal file system directory resides. Its size is determined by the number of directory entries which you specify in the Maximum Directory Entries field. The default file name for the directory partition is FEPn:>Static>fs-name-partm.UFD. *n* is the highest mounted disk on the system, *fs-name* is the file-system name specified for the command, and *m* is the number of the partition.

Maximum Directory Entries:

This entry specifies the maximum number of databases which may reside in the file system at any time. Note that Static always takes two of these entries for itself—one for the log file, and one for the Directory database. These entries are reusable, so if a database is deleted, using the Delete File command (in

conjunction with a database pathname, not a FEPFS pathname), that entry in the file system directory is reusable for another database. On the Symbolics 36xx, there are 71 directory entries in each FEPFS block. The directory is organized as a hash file, so it's desirable to make the directory large enough that it's not densely filled.

Partition: These entries specify the partitions to be used for the file system. There may be as many partitions as you want, and they can live on any of the disks. In general, there should be as few partitions as possible in order to avoid disk fragmentation. The default pathname for a partition is `FEPm:>Static>fs-name-partn.file`, where *m* is the highest mounted unit number, *fs-name* is the name of the file system, and *n* is the partition number in the ordering of all the partitions entered in the AVV menu.

Blocks: This entry specifies the number of blocks to allocate for the partition. When you enter a value for this field, the values in the available disk space headings will change accordingly to take into account how much of the free space you have allocated. You may click on None in this field to remove the partition the menu (and hence not include it as part of the file system when it is created).

When all the parameters have been entered, pressing END will cause the file system to be created. First, the file system object will be created in the namespace database (permanently, unless :Locally Yes was specified). Note that the messages printed by the command do not indicate whether the namespace was updated locally or globally. Second, all of the partitions are created in the FEPFS, and their :DONT-DELETE properties are set. You don't need to create any of the partitions yourself—this is done automatically for you, including the proper allocation of space. Third, the log file in the file system is initialized. Finally, the directory database is created.

Here's a sample run:

```
Command: Create Static File System SQUASH
FEP0: 21464 Available (Originally: 21464 free, 88696/110160 used (81%))
FEP1: 137 Available (Originally: 137 free, 146743/146880 used (100%))
FEP2: 70727 Available (Originally: 71742 free, 38418/110160 used (35%))
```

```
Directory Partition: FEP2:>Static>SQUASH.UFD
Maximum Directory Entries: 1000
Initial Log Size in Blocks: 500
Partition: FEP2:>Static>part0.file.newest
  Blocks (None to remove): None 1000
Partition: FEP2:>Static>part1.file.newest
  Blocks (None to remove): None an integer
```

Creating file-system object SCRC|SQUASH in namespace... Done.
 Initializing local UFS with associated directory structure... Done.
 Creating local DBFS with associated directory structure... Done.
 Initializing DBFS Directory database... Done.

Delete Static File System Command

Delete Static File System *file-system-name keyword*

Expunges an entire Static file system, and removes all traces of it, including every database in it; this is a very dangerous command.

file-system-name A symbol naming a file system that is resident on the local host.

keywords :Locally

:Locally {Yes, No} Whether to update only local namespace information (Yes), or to update the namespace database server as well (No). The default is No. See the section "The Locally Namespace Editor Command".

Because this command permanently removes the Static File System, and all databases in it, it is a dangerous command and it asks for confirmation. If you answer Yes, the file system and all the databases in it are destroyed by removing the file system partitions from the FEP directory in which they were placed by the Create Static File System command. The command destroys the file system partitions, even though they may have the :DONT-DELETE flag set for them in the FEPFS (the Create Static File System command sets the :DONT-DELETE property for each of the partitions in a file system).

If you have done a complete backup dump, you can restore the contents of a deleted file system by using the Complete Restore command of the Static File System Operations activity. If you have not done a complete backup, the data cannot be restored.

Dump Database Command

Dump Database *database-pathname destination-pathname*

Writes all the information in the database into a text file.

database-pathname A pathname indicating the location of a Static database.

destination-pathname

A pathname of a file on any file system; this need not be stored on a Symbolics machine.

This command is useful for several purposes. It enables you to:

- Move a database from one place to another, over a channel that can handle only ordinary text.
- Store the contents of a database on some kind of storage medium (e.g. a particular tape format) that can handle only ordinary text.
- Edit the text file to reorganize the database.

We discuss the details of the Dump Database and Load Database commands elsewhere: See the section "High-level Dumper/Loader of Static Databases".

Load Database Command

Load Database *database-pathname destination-pathname keywords*

Takes a text file produced by Dump Database, and makes a new database containing the same information.

database-pathname A pathname indicating the location of a Static database.

destination-pathname

A pathname of a file on any file system; this need not be stored on a Symbolics machine.

keywords :If Exists

:If Exists {Error, Create} Specifies the action to be taken if the database specified by the *database-pathname* already exists. Error signals an error, and Create causes the old database to be erased and replaced by the database being loaded.

Unless you have edited the text file, Load Database makes an exact copy of the original database that was dumped, including keeping the unique ids the same.

We discuss the details of the Dump Database and Load Database commands elsewhere: See the section "High-level Dumper/Loader of Static Databases".

Set Database Schema Name Command

Set Database Schema Name *pathname new-schema-name*

Informs the database that its schema name is now the given *new-schema-name*.

pathname A pathname indicating the location of a Static database.

new-schema-name A symbol.

If you move a Static program from one package to another, and the database already exists, it is necessary to use this command to update the database to inform it of the new schema name.

See the section "Warning About Changing the Package of a Static Program".

Show All Static File Systems Command

Show All Static File Systems *namespace*

Lists all the file system objects in the namespace, and the host on which each one resides.

namespace A symbol that specifies a namespace in which to search. By default, all namespaces in the namespace search path are searched

Show Database Schema Command

Show Database Schema *pathname*

Prints the definition of the schema of the database specified by pathname. This command is useful if you see a database in a Static file system and don't know what it is. It's also useful for seeing what indexes currently exist in a database.

Not all of the information from the template schema is stored in the database itself, so when Show Database Schema reconstructs the schema definition from the database, not all of the original information is recovered. Specifically:

The following attribute options are reconstructed: **:unique**, **:index**, **:index-average-size**, **:inverse-index**, **:inverse-index-average-size**, **:inverse-index-exact**, **:inverse-cached**, **:area**, **:attribute-set**, and **:no-nulls**.

The following attribute options are not reconstructed: **:cached**, **:initform**, **:inverse**, **:inverse-exact**, **:cluster**, **:accessor**, **:reader**, **:writer**, and **:read-only**.

The following entity-type options are reconstructed: **:area**, **:type-set**, **:multiple-index**, and **:multiple-index-exact**.

The following entity-type options are not reconstructed: **:conc-name**, **:constructor**, **:default-init-plist**, **:documentation**, **:init-keywords**, **:instance-variables**, and **:own-cluster**.

See the section "Examining the Schema of a Static Database".

Show Static Partitions Command

Show Static Partitions *file-system-name*

Shows the amount of free space remaining in a file system's partition(s).

file-system-name Name of a Static file system which resides on the local host.

This command may be done only for a file system stored on the local host.

For example:

```
Show Static Partitions (a file-system) SQUASH
```

<i>Partition</i>	<i>Free Space</i>
FEP1:>squash>squash.file.newest	0/1056
FEP0:>squash>squash.file.newest	0/3000
FEP2:>Static>SQUASH-part2.file.newest	54/1000

Update Database Schema Command

Update Database Schema *database-pathname*

Used when you have modified a schema; this command compares the template schema to the real schema in the database, and updates the real schema to match the template schema.

database-pathname A pathname indicating the location of a Static database.

We discuss this subject in detail elsewhere: See the section "Modifying a Static Schema".

Documentation Update: future-common-lisp:loop

Using future-common-lisp:loop

What is future-common-lisp:loop?

The macro **future-common-lisp:loop** performs iteration by executing a series of forms one or more times. Loop keywords are symbols recognized by **future-common-lisp:loop**. They provide such capabilities as control of direction of iteration, accumulation of values inside the loop body, and evaluation of expressions that precede or follow the loop body.

For **future-common-lisp:loop** without clauses, each form is evaluated in turn from left to right. When the last form has been evaluated, then the first form is evaluated again, and so on, in a never-ending cycle. **future-common-lisp:loop** establishes an implicit block named **nil**. The execution of **future-common-lisp:loop** can be terminated explicitly, by using **return**, **throw** or **return-from**, for example.

How `future-common-lisp:loop` Works

Expansion of the `future-common-lisp:loop` macro produces an implicit block named `nil` unless `named` is supplied. Thus, `return` and `return-from` can be used to return values from `future-common-lisp:loop` or to exit `future-common-lisp:loop`. Within the executable parts of loop clauses and around the entire `future-common-lisp:loop` form, variables can be bound by using regular lisp mechanisms, such as `let`.

When Lisp encounters a `future-common-lisp:loop` form, it invokes the loop facility, which expands the loop expression into simpler, less abstract code that implements the loop. The loop facility defines clauses that are introduced by loop keywords. The loop clauses contain forms and loop keywords.

Loop keywords are not true keywords; they are ordinary symbols, recognized by print name, that are meaningful only to the `future-common-lisp:loop` facility. Because loop keywords are recognized by their names, they may be in any package. If no loop keywords are supplied, the loop facility repeatedly executes the loop body.

The `future-common-lisp:loop` macro translates the given form into code and returns the expanded form. The expanded form is one or more lambda expressions for the local binding of loop variables and a `block` and a `tagbody` that express a looping control structure. The variables established in `future-common-lisp:loop` are bound as if by `let` or `lambda`. The assignment of the initial values is always calculated in the order specified by the user. (A variable is sometimes bound to a meaningless value of the correct type, and then later in the prologue it is set to the true initial value by using `setq`.)

After the form is expanded, it consists of three basic parts in the tagbody:

prologue

The loop prologue contains forms that are executed before iteration begins, such as any automatic variable initializations prescribed by the variable clauses, along with any `initially` clauses in the order they appear in the source.

body

The loop body contains those forms that are executed during iteration, including application-specific calculations, termination tests, and variable stepping.

epilogue

The loop epilogue contains forms that are executed after iteration terminates, such as `finally` clauses, if any, along with any implicit return value from an accumulation clause or an end-test clause.

Some clauses from the source form contribute code only to the loop prologue; these clauses must come before other clauses that are in the main body of the `future-common-lisp:loop` form. Others contribute code only to the loop epilogue. All other clauses contribute to the final translated form in the same order given in the original source form of the `future-common-lisp:loop`.

Syntax of `future-common-lisp:loop`

Notational Conventions and Syntax for `future-common-lisp:loop`

The syntax for loop constructs is represented as follows:

- Loop keyword names are in **boldface**.
- Symbols are enclosed by braces { }. Braces followed by an asterisk (*) indicate that the contents enclosed by the braces can appear any number of times or not at all.

`{z}*` zero or one occurrences of *z*

A plus sign (+) indicates that the contents enclosed by the braces must appear at least once.

`{z}+` one or more occurrences of *z*

- Brackets [] indicate that the contents enclosed by the brackets is optional and can appear only once.

`[z]` zero or more occurrences of *z*

- Elements separated by a vertical bar | indicate that either of the elements can appear, but not both.

```
{for | as} var [type-spec] [{from | downfrom | upfrom} form1]
                        [{to | downto | upto | below | above} form2]
                        [by form3]
```

An overview of loop syntax is provided in the following paragraphs. More detailed syntax descriptions of individual clauses are provided in the section "`future-common-lisp:loop`". Syntax for `future-common-lisp:loop` with keyed clauses:

```
future-common-lisp:loop [named name] [variables]* [main]
```

```
variables ::= with | initial-final | for | as
```

```
initial-final ::= initially | finally
```

```
main ::= unconditional | accumulation | conditional | end-test | initial-final
```

```
unconditional ::= do | doing | return
```

```
accumulation ::= collect | collecting | append | appending
                | nconc | nconcing | count | counting | sum | summing
                | maximize | maximizing | minimize | minimizing
```

```
conditional ::= when | if | unless
```

end-test::= while | until | always | never | thereis | repeat

Syntax for **future-common-lisp:loop** without clauses:

future-common-lisp:loop [*tag* | *form*]*

Syntax for **future-common-lisp:loop-finish**:

(**future-common-lisp:loop-finish**)

Parsing **future-common-lisp:loop** Clauses

The syntactic parts of the **future-common-lisp:loop** construct are called clauses. The parsing of keywords determines the scope of clause. The following example shows a **future-common-lisp:loop** construct with six clauses:

```
(loop for i from 1 to (compute-top-value)      ; first clause
     while (not (unacceptable i))           ; second clause
     collect (square i)                      ; third clause
     do (format t "Working on ~D now" i)      ; fourth clause
     when (evenp i)                          ; fifth clause
       do (format t "~D is an even number" i)
     finally (format t "About to exit!"))    ; sixth clause
```

Each loop keyword introduces either a compound or a simple loop clause that can consist of a loop keyword followed by a single form. The number of forms in a clause is determined by the loop keyword that begins the clause and by the auxiliary keywords in the clause. The keywords **do**, **initially**, and **finally** are the only loop keywords that can take any number of forms and group them as if in a single **progn** form.

Loop clauses can contain auxiliary keywords, sometimes called prepositions. For example, the first clause in the code above includes the prepositions **from** and **to**, which mark the value from which stepping begins and the value at which stepping ends.

Order of Execution in **future-common-lisp:loop**

With the exceptions listed below, clauses are executed in the loop body in the order in which they appear in the source. Execution is repeated until a clause terminates the **future-common-lisp:loop** or until a **return**, **go**, or **throw** form is encountered. The following actions are exceptions to the linear order of execution:

- All variables are initialized first, regardless of where the establishing clauses appear in the source. The order of initialization follows the order of these clauses.

- The code for any **initially** clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop prologue after any implicit variable initializations.
- The code for any **finally** clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop epilogue before any implicit values from the accumulation clauses are returned. Explicit returns anywhere in the source, however, will exit the **future-common-lisp:loop** without executing the epilogue code.
- A **with** clause introduces a variable binding and an optional initial value. The initial values are calculated in the order in which the **with** clauses occur.
- Iteration control clauses implicitly initialize variables, step variables (generally between each execution of the loop body), and perform termination tests (generally just before the execution of the loop body).

Kinds of future-common-lisp:loop Clauses

Loop clauses fall into one of the following six categories:

Variable initialization and stepping

- | | |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| for, as | The for and as constructs provide iteration control clauses that establish a variable to be initialized. for and as clauses can be combined with the loop keyword and to get parallel initialization and stepping. Otherwise, the initialization and stepping are sequential. The for and as constructs provide a termination test that is determined by the iteration control clause. |
| with | The with construct is similar to a single let clause. with clauses can be combined using the loop keyword and to get parallel initialization. |
| repeat | The repeat construct causes iteration to terminate after a specified number of times. It uses an internal variable to keep track of the number of iterations. |

Value accumulation

- | | |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| collect | The collect construct takes one <i>form</i> in its clause and adds the value of that <i>form</i> to the end of a list of values. By default, the list of values is returned when the future-common-lisp:loop finishes. |
|----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

```

;; Collect all the symbols in a list.
(loop for i in '(bird 3 4 turtle (1 . 4) horse cat)
      when (symbolp i) collect i)
=> (BIRD TURTLE HORSE CAT)

;; Collect and return odd numbers.
(loop for i from 1 to 10
      if (oddp i) collect i)
=> (1 3 5 7 9)

;; Collect items into local variable,
;; but don't return them.
(loop for i in '(a b c d) by #'caddr
      collect i into my-list
      finally (print my-list))
(A C)
=> NIL

```

append

The **append** construct takes one *form* in its clause and appends the value of that *form* to the end of a list of values. By default, the list of values is returned when the **future-common-lisp:loop** finishes.

```

;; Use APPEND to concatenate some sublists.
(loop for x in '((a) (b) ((c)))
      append x)
=> (A B (C))

```

nconc

The **nconc** construct is similar to the **append** construct, but its **list** values are concatenated as if by the function **nconc**. By default, the **list** of values is returned when the **future-common-lisp:loop** finishes.

```

;; NCONC some sublists together. Note that only lists
;; made by the call to LIST are modified.
(loop for i upfrom 0
      as x in '(a b (c))
      nconc (if (evenp i) (list x) nil))
=> (A (C))

```

sum

The **sum** construct takes one *form* in its clause that must evaluate to a **number** and accumulates the sum of all these **numbers**. By default, the cumulative sum is returned when the **future-common-lisp:loop** finishes.

```
(loop for i fixnum in '(1 2 3 4 5)
      sum i)
=> 15
(setq series '(1.2 4.3 5.7))
=> (1.2 4.3 5.7)
(loop for v in series
      sum (* 2.0 v))
=> 22.4
```

count

The **count** construct takes one *form* in its clause and counts the number of times that the *form* evaluates to a non-**nil** value. By default, the count is returned when the **future-common-lisp:loop** finishes.

```
(loop for i in '(a b nil c nil d e)
      count i)
=> 5
```

minimize

The **minimize** construct takes one *form* in its clause and determines the minimum value obtained by evaluating that *form*. By default, the minimum value is returned when the **future-common-lisp:loop** finishes.

```
(loop for i in '(2 1 5 3 4)
      minimize i)
=> 1
;; In this example, FIXNUM applies to the variable RESULT.
(loop for v float in series
      minimize (round v) into result fixnum
      finally (return result))
=> 1
(loop for i in '(2 1 5 3 4)
      minimize i)
=> 1
```

maximize

The **maximize** construct takes one *form* in its clause and determines the maximum value obtained by evaluating that *form*. By default, the maximum value is returned when the **future-common-lisp:loop** finishes.

```
(loop for i in '(2 1 5 3 4)
      maximize i)
=> 5

;; In this example, FIXNUM applies to the internal
;; variable that holds the maximum value.
(setq series '(1.2 4.3 5.7))
=> (1.2 4.3 5.7)
(loop for v in series
      maximize (round v) fixnum)
=> 6
```

Termination conditions

- while** The **while** construct takes one *form*, a **condition**, and terminates the iteration if the **condition** evaluates to **nil**. A **while** clause is equivalent to the expression (**if** (not **condition**) (**future-common-lisp:loop-finish**)).
- until** The **until** construct is the inverse of **while**; it terminates the iteration if the condition evaluates to any non-**nil** value. An **until** clause is equivalent to the expression (**if condition** (**future-common-lisp:loop-finish**)).
- always** The **always** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**.
- never** The **never** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **non-nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**.
- thereis** The **thereis** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **non-nil**; in this case, it returns that value.

future-common:loop-finish

The **future-common-lisp:loop-finish** macro terminates iteration and returns any accumulated result. Any **finally** clauses that are supplied are evaluated.

Unconditional execution

- do** The **do** construct evaluates all forms in its clause.
- return** The **return** construct takes one form and returns its value. It is equivalent to the clause **do (return it value)**.

```
;; Print numbers and their squares.
;; The DO construct applies to multiple forms.
(loop for i from 1 to 3
      do (print i)
          (print (* i i)))
1
1
2
4
3
9
=> NIL
```

Conditional execution

if The **if** construct takes one *form* as a predicate and a clause that is executed when the predicate is true. The clause can be a value accumulation, unconditional, or another conditional clause; it can also be any combination of such clauses connected by the loop keyword **and**.

when The **when** construct is a synonym for the **if** construct.

```
;; Signal an exceptional condition.
(loop for item in '(1 2 3 a 4 5)
      when (not (numberp item))
      return (error "Enter a new value" "Non-numeric value: ~s" item))
Error: Non-numeric value: A
```

```
;; The previous example is equivalent to the following one.
(loop for item in '(1 2 3 a 4 5)
      when (not (numberp item))
      do (return
          (error "Enter a new value" "Non-numeric value: ~s" item)))
Error: Non-numeric value: A
```

```
;; This example parses a simple printed string representation from
;; BUFFER (which is itself a string) and returns the index of the
;; closing double-quote character.
(let ((buffer "\"foo\" \"bar\""))
  (loop initially (unless (char= (char buffer 0) #\\")
                    (loop-finish))
        for i fixnum from 1 below (string-length buffer)
        when (char= (char buffer i) #\\")
        return i))
=> 4
```

```
;; The FINALLY clause prints the last value of I.
;; The collected value is returned.
(loop for i from 1 to 10
      when (> i 5)
      collect i
      finally (print i))
=> 4
(6 7 8 9 10)
```

```

;; Return both the count of collected numbers and the numbers.
(loop for i from 1 to 10
      when (> i 5)
        collect i into number-list
        and count i into number-count
      finally (return (values number-count number-list)))
=> 5
(6 7 8 9 10)

```

unless	The unless construct is similar to when except that it complements the predicate.
else	The else construct provides an optional component of if , when , and unless clauses that is executed when the predicate is false. The component is one of the clauses described under if .
end	The end construct provides an optional component to mark the end of a conditional clause.

Miscellaneous operations

named The **named** construct gives a name for the block of the loop.

```

;; Just name and return.
(loop named max
      for i from 1 to 10
      do (print i)
      do (return-from max 'done))
1
=> DONE

```

initially The **initially** construct causes its forms to be evaluated in the loop prologue, which precedes all **future-common-lisp:loop** code except for initial settings supplied by the constructs **with**, **for**, or **as**.

finally The **finally** construct causes its forms to be evaluated in the loop epilogue after normal iteration terminates. An unconditional clause can also follow the loop keyword **finally**.

Constructs in future-common-lisp:loop

This section describes the constructs provided by **future-common-lisp:loop**. The constructs are grouped according to function into the following categories:

- Iteration Control
- End-Test Control

- Value Accumulation
- Variable Initializations
- Conditional Execution
- Unconditional Execution
- Miscellaneous Features

Iteration Control in `future-common-lisp:loop`

Iteration control clauses allow direction of `future-common-lisp:loop` iteration. The loop keywords **for**, **as**, and **repeat** designate iteration control clauses. Iteration control clauses differ with respect to the specification of termination conditions and to the initialization and stepping of loop variables. Iteration clauses by themselves do not cause the loop facility to return values, but they can be used in conjunction with value-accumulation clauses to return values.

All variables are initialized in the loop prologue. The scope of the variable binding is lexical unless it is proclaimed **special**; thus, the variable can be accessed only by forms that lie textually within the `future-common-lisp:loop`. Stepping assignments are made in the loop body before any other forms are evaluated in the body.

The variable argument in iteration control clauses can be a destructuring list. A destructuring list is a tree whose non-**null** atoms are **symbols** that can be assigned a value. See the section "Destructuring in `future-common-lisp:loop`".

The iteration control clauses **for**, **as**, and **repeat** must precede any other loop clauses, except **initially**, **with**, and **named**, since they establish variable bindings. When iteration control clauses are used in a `future-common-lisp:loop`, termination tests in the loop body are evaluated before any other loop body code is executed.

If multiple iteration clauses are used to control iteration, variable initialization and stepping occur sequentially by default. The **and** construct can be used to connect two or more iteration clauses when sequential binding and stepping are not necessary. The iteration behavior of clauses joined by **and** is analogous to the behavior of the macro **do** with respect to **do***.

for and **as** Constructs

The **for** and **as** clauses iterate by using one or more local loop variables that are initialized to some value and that can be modified or stepped after each iteration. For these clauses, iteration terminates when a local variable reaches some supplied value or when some other loop clause terminates iteration. At each iteration, variables can be **word** stepped by an increment or a decrement or can be assigned a new value by the evaluation of a form. Destructuring can be used to assign initial values to variables during iteration.

The **for** and **as** keywords are synonyms; they can be used interchangeably. There are seven syntactic formats for these constructs. In each syntactic format, the type of *var* can be supplied by the optional *type-spec* argument. If *var* is a destructuring list, the type supplied by the *type-spec* argument must appropriately match the elements of the list.

Syntax 1:

```
{for | as} var [type-spec] [{from | downfrom | upfrom} form1]
                        [{to | downto | upto | below | above} form2] [by form3]
```

The **for** or **as** construct iterates from the value supplied by *form1* to the value supplied by *form2* in increments or decrements denoted by *form3*. Each expression is evaluated only once and must evaluate to a number.

The variable *var* is bound to the value of *form1* in the first iteration and is stepped by the value of *form3* in each succeeding iteration, or by 1 if *form3* is not provided.

The following loop keywords serve as valid prepositions within this syntax, and at least one must be used in any **for** or **as** construct:

from *form1* The loop keyword **from** marks the value from which stepping begins, as supplied by *form1*. Stepping is incremental by default. If decremental stepping is desired, the preposition **downto** or **above** must be used with *form2*. For incremental stepping, the default **from** value is 0.

downfrom, upfrom *form1* The loop keyword **downfrom** indicates that the variable *var* is decreased in decrements supplied by *form3*; the loop keyword **upfrom** indicates that *var* is increased in increments supplied by *form3*.

to *form2* The loop keyword **to** marks the end value for stepping supplied in *form2*. Stepping is incremental by default. If decremental stepping is desired, the preposition **downto**, **downfrom**, or **above** must be used with *form2*.

downto, upto *form2* The loop keyword **downto** allows iteration to proceed from a larger number to a smaller number by the decrement *form3*. The loop keyword **upto** allows iteration to proceed from a smaller number to a larger number by the increment *form3*. Since there is no default for *form1* in decremental stepping, a value must be supplied with **downto**.

below, above *form2* The loop keywords **below** and **above** are analogous to **upto** and **downto** respectively. These keywords stop iteration just before the value of the variable *var* reaches the value supplied by *form2*; the end value of *form2* is not included. Since there is

no default for *form1* in decremental *stepping*, a value must be supplied with **above**.

by form3 The loop keyword **by** marks the increment or decrement supplied by *form3*. The value of *form3* can be any positive *number*. The default value is 1.

In an iteration control clause, the **for** or **as** construct causes termination when the supplied limit is reached. That is, iteration continues until the value *var* is stepped to the exclusive or inclusive limit supplied by *form2*. The range is exclusive if *form3* increases or decreases *var* to the value of *form2* without reaching that value; the loop keywords **below** and **above** provide exclusive limits. An inclusive limit allows *var* to attain the value of *form2*; **to**, **downto**, and **upto** provide inclusive limits.

By convention, **for** introduces new iterations and **as** introduces iterations that depend on a previous iteration specification.

```
;; Print some numbers.
(loop as i from 1 to 3
  do (print i))
1
2
3
=> NIL

;; Print every third number.
(loop for i from 10 downto 1 by 3
  do (print i))
10
7
4
1
=> NIL

;; Step incrementally from the default starting value.
(loop as i below 3
  do (print i))
0
1
2
=> NIL
```

Syntax 2:

```
{for | as} var [type-spec] in form1 [by step-fun]
```

The **for** or **as** construct iterates over the contents of a list. It checks for the end of the list as if by using **endp**. The variable *var* is bound to the successive elements of the list in *form1* before each iteration. At the end of each iteration, the

function *step-fun* is applied to then list; the default value for *step-fun* is **cdr**. The loop keywords **in** and **by** serve as valid prepositions in this syntax. The **for** or **as** construct causes termination when the end of the list is reached. For example:

```
;; Print every item in a list.
(loop for item in '(1 2 3) do (print item))
1
2
3
=> NIL

;; Print every other item in a list.
(loop for item in '(1 2 3 4 5) by #'cddr
      do (print item))
1
3
5
=> NIL

;; Destructure a list, and sum the x values using fixnum arithmetic.
(loop for (item . x) (t . fixnum) in '((A . 1) (B . 2) (C . 3))
      unless (eq item 'B) sum x)
=> 4
```

Syntax 3:

```
{for | as} var [type-spec] on form1 [by step-fun]
```

The **for** or **as** construct iterates over the contents of a list. It checks for the end of the list as if by using **endp**. The variable *var* is bound to the successive tails of the *list* in *form1*. At the end of each iteration, the function *step-fun* is applied to the list; the default value for *step-fun* is **cdr**. The loop keywords **on** and **by** serve as valid prepositions in this syntax. The **for** or **as** construct causes termination when the end of the list is reached. The following example demonstrates the **for-as-on-list** subclause:

```
;; Collect successive tails of a list.
(loop for sublist on '(a b c d)
      collect sublist)
=>((A B C D) (B C D) (C D) (D))

;; Print a list by using destructuring with the loop keyword ON.
(loop for (item) on '(1 2 3)
      do (print item))
1
2
3
=> NIL
```

Syntax 4:

```
{for | as} var [type-spec] = form1 [then form2]
```

The **for** or **as** construct initializes the variable *var* by setting it to the result of evaluating *form1* on the first iteration, then setting it to the result of evaluating *form2* on the second and subsequent iterations. If *form2* is omitted, the construct uses *form1* on the second and subsequent iterations. The loop keywords = and **then** serve as valid prepositions in this syntax. This construct does not provide any termination conditions. For example:

```
; Collect some numbers.
(loop for item = 1 then (+ item 10)
  for iteration from 1 to 5
  collect item)
=> (1 11 21 31 41)
```

Syntax 5:

```
{for | as} var [type-spec] across vector
```

The **for** or **as** construct binds the variable *var* to the value of each element in the array *vector*. The loop keyword **across** marks the array *vector*; **across** is used as a preposition in this syntax. Iteration stops when there are no more elements in the supplied array that can be referenced. Some implementations might recognize a **the** special form in the *vector* form to produce more efficient code. For example:

```
(loop initially (terpri) "foo"
  do (write-char char stream))
foo
=> NIL
```

Syntax 6:

```
{for | as} var [type-spec] being {each | the}
  {hash-key[s] | hash-value[s]}
  {in | of} hash-table [using ({hash-key | hash-value}
  other-var)]
```

The **for** or **as** construct iterates over the elements, keys, and values of a hash table. In this syntax, a compound preposition is used to designate access to a hash table. The variable *var* takes on the value of each hash key or hash value in the supplied hash table. The following loop keywords serve as valid prepositions within this syntax:

- | | |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| being | The keyword being introduces either the loop method hash-key or hash-value . |
| each, the | The loop keyword each follows the loop keyword being when hash-key or hash-value is used. The loop keyword the is used with hash-keys and hash-values only for ease of reading. This agreement isn't required. |

hash-key, hash-keys

These loop keywords access each key entry of the *hash-table*. If the name *hash-value* is supplied in a **using** construct with one of these loop methods, the iteration can optionally access the keyed value. The order in which the keys are accessed is undefined; empty slots in the *hash-table* are ignored.

hash-value, hash-values

These loop keywords access each value entry of a **hash-table**. If the name **hash-key** is supplied in a **using** construct with one of these loop methods, the iteration can optionally access the key that corresponds to the value. The order in which the keys are accessed is undefined; empty slots in the **hash-table** are ignored.

using

The loop keyword **using** introduces the optional key or the keyed value to be accessed. It allows access to the hash key if iteration is over the hash values, and the hash value if iteration is over the hash keys

in, of

These loop prepositions introduce *hash-table*.

In effect the following expression is a compound preposition:

```
being [each | the] [hash-value | hash-values | hash-key |
hash-key] [in | of ]
```

Iteration stops when there are no more hash keys or hash values to be referenced in the supplied hash table.

Syntax 7:

```
{for | as} var [type-spec] being {each | the}
      {symbol[s] | present-symbol[s] | external-symbol[s]}
      [{in | of} package]
```

The **for** or **as** construct iterates over the symbols in a package. In this syntax, a compound preposition is used to designate access to a package. The variable *var* takes on the value of each symbol in the supplied package. The following loop keywords serve as valid prepositions within this syntax:

being

The keyword **being** introduces either the loop method **symbol**[s], **present-symbol**[s], or **external-symbol**[s].

each, the

The loop keyword **each** follows the loop keyword **being** when **symbol**, **present-symbol**, or **external-symbol** is used. The loop keyword **the** is used with **symbols**, **present-symbols**, and **external-symbols** only for ease of reading. This agreement isn't required.

present-symbol, present-symbols

These loop methods iterate over the symbols that are present but not external in a given *package*. The package to be iterated over is supplied in the same way that package arguments to **find-package** are supplied. If the package for the iteration is not supplied, the current package is used. If a package that does not exist is supplied, an error of type **package-error** is signalled.

symbol, symbols

These loop methods iterate over **symbols** that are accessible from a given *package*. The package to be iterated over is supplied in the same way package arguments to **find-package** are supplied. If the package for the iteration is not supplied, the current package is used. If a package that does not exist is supplied, an error of type **conditions:package-error** is signalled.

external-symbol, external-symbols

These loop methods iterate over the external **symbols** of the given *package*. The package to be iterated over is supplied in the same way package arguments to **find-package** are supplied. If the package for the iteration is not supplied, the current package is used. If a package that does not exist is supplied, an error of type **package-error** is signalled.

in, of

These loop prepositions mark the package *package*.

In effect

```
[being] [each | the] [[present | external] symbol] |
[[present | external] symbols]] [in | of]
```

is a compound preposition. Iteration stops when there are no more symbols to be referenced in the supplied *package*.

The repeat Construct in future-common-lisp:loop**repeat form**

The **repeat** construct causes iteration to terminate after a specified number of times. The loop body executes *n* times, where *n* is the value of the expression *form*. The *form* argument is evaluated once in the loop prologue. If the expression evaluates to 0 or to a negative number, the loop body is not evaluated. The following example demonstrates the **repeat** construct:

```
(loop repeat 3
  do (format t "~&What I say three times is true.~%"))
  What I say three times is true
  What I say three times is true
  What I say three times is true
=> NIL
```

```
(loop repeat -15
  do (format t "~&What you see is what you expect.~%"))
=> NIL
```

End-Test Control in `future-common-lisp:loop`

The following loop keywords designate constructs that use a single test condition to determine when loop iteration should terminate:

always, never, thereis

while, until

The constructs **always**, **never**, and **thereis** provide specific values to be returned when a loop terminates. Using **always**, **never**, or **thereis** in a loop with value-returning accumulation clauses that are not **into** causes an error of type **conditions:program-error** to be signalled. Since **always**, **never**, and **thereis** use the macro **return** to terminate iteration, any **finally** clause that is supplied is not evaluated. In all other respects these constructs behave like the **while** and **until** constructs.

The macro **future-common-lisp:loop-finish** can be used at any time to cause regular termination. In regular termination, **finally** clauses are executed and default return values are returned.

always, never, thereis

always *form* The **always** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**. If the value of the supplied *form* is never **nil**, some other construct can terminate the iteration. Otherwise, it provides a default return value of **t**.

```
;; Make sure I is always less than 11 (two ways).
;; The FOR construct terminates these loops.
(loop for i from 0 to 10
  always (< i 11))
=> T
```

never *form* The **never** construct terminates iteration the first time that the value of the supplied **form** is non-**nil**; the **future-common-lisp:loop** returns **nil**. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **t**.

```
(loop for i from 0 to 10
      never (> i 11))
=>T
```

thereis *form* The **thereis** construct takes one **form** and terminates the **loop** if the *form* ever evaluates to non-**nil**; in this case, it returns that value. The **thereis** construct terminates iteration the first time that the value of the supplied *form* is non-**nil**; the **future-common-lisp:loop** returns the value of the supplied *form*. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **nil**.

```
;; If I exceeds 10 return I; otherwise, return NIL.
;; The THEREIS construct terminates this loop.
(loop for i from 0
      thereis (when (> i 10) i) ) => 11
```

There are two differences between the **thereis** and **until** constructs:

- The **until** construct does not contribute a return value based on the value of the supplied *form*.
- The **until** construct executes any **finally** clause. Since **thereis** uses the macro **return** to terminate iteration, any **finally** clause that is supplied is not evaluated.

;;; The FINALLY clause is not evaluated in these examples.

```
(loop for i from 0 to 10
      always (< i 9)
      finally (print "you won't see this"))
```

=> NIL

```
(loop never t
      finally (print "you won't see this"))
```

=> NIL

```
(loop thereis "Here is my value"
      finally (print "you won't see this"))
```

=> "Here is my value"

;; The FOR construct terminates this loop, so the FINALLY clause
;; is evaluated.

```
(loop for i from 1 to 10
      thereis (> i 11)
      finally (print i))
```

11

=> NIL

;; If this code could be used to find a counterexample to Fermat's
;; last theorem, it would still not return the value of the
;; counterexample because all of the THEREIS clauses in this example
;; only return T. Of course, this code does not terminate.

```
(loop for z upfrom 2
      thereis
        (loop for n upfrom 3 below (log z 2)
              thereis
                (loop for x below z
                      thereis
                    (loop for y below z
                          thereis (= (+ (expt x n) (expt y n))
                                       (expt z n))))))
```

; The finally clause is not evaluated.

```
(loop never t
      finally (print "You won't see this."))
```

=> NIL

while, until

while *form*

The **while** construct allows iteration to continue until the supplied *form* evaluates to **nil**. The supplied *form* is reevaluated at the location of the **while** clause.

```

(loop while (hungry-p) do (eat))

;; UNTIL (NOT...) is equivalent to WHILE.
(loop until (not (hungry-p)) do (eat))

;; Collect the length and the items of STACK.
(let ((stack '(a b c d e f)))
  (loop while stack
        for item = (length stack) then (pop stack)
        collect item))
=> (6 A B C D E F)

```

until form

The **until** construct is equivalent to **while (not form)** dots. If the value of the supplied *form* is non-**nil**, iteration terminates.

```

;; Use WHILE to terminate a loop that otherwise
;; wouldn't terminate.
;; Note that WHILE occurs after the WHEN.
(loop for i fixnum from 3
      when (oddp i) collect i
      while (< i 5))
=> (3 5)

```

The **while** and **until** constructs can be used at any point in a **future-common-lisp:loop**. If an **until** or **while** clause causes termination, any clauses that precede it in the source are still evaluated. If the **until** and **while** constructs cause termination, control is passed to the loop epilogue, where any **finally** clauses will be executed.

There are two differences between the **never** and **until** constructs:

- The **until** construct does not contribute a return value based on the value of the supplied *form*.
- The **until** construct executes a **finally** clause. Since **never** uses the macro **return** to terminate iteration, any **finally** clause that is supplied is not evaluated.

In most cases it is not necessary to use **future-common-lisp:loop-finish** because other loop control clauses terminate the **future-common-lisp:loop**. The macro **future-common-lisp:loop-finish** is used to provide a normal exit from a nested condition inside a **future-common-lisp:loop**.

In normal termination, **finally** clauses are executed and default return values are returned. Since **future-common-lisp:loop-finish** transfers control to the loop epilogue, using **future-common-lisp:loop-finish** within a **finally** expression can cause infinite looping. It is implementation dependent whether or not, in a particular

future-common-lisp:loop invocation, **future-common-lisp:loop-finish** is a global *macro* or a local one (created as if by **macrolet**).

End-test control constructs can be used anywhere within the loop body. The termination conditions are tested in the order in which they appear.

Value Accumulation in **future-common-lisp:loop**

Accumulating values during iteration and returning them from a loop is often useful. Some of these accumulations occur so frequently that special loop clauses have been developed to handle them.

The following loop keywords designate clauses that accumulate values in lists and return them:

- **append, appending**
- **collect, collecting**
- **ncconc, nconcing**

The following loop keywords designate clauses that accumulate and return numerical values:

- **count, counting**
- **maximize, maximizing**
- **minimize, minimizing**
- **sum, summing**

Value-returning accumulation clauses can be combined in a **loop** if all the clauses accumulate the same type of object. By default, the loop facility returns only one value; thus, the objects collected by multiple accumulation clauses as return values must have compatible types. For example, since both the **collect** and **append** constructs accumulate objects into a list that is returned from a **future-common-lisp:loop**, they can be combined safely.

```
;; Collect every name and the kids in one list by using
;; COLLECT and APPEND.
(loop for name in '(fred sue alice joe june)
      for kids in '((bob ken) () ()) (kris sunshine) ())
      collect name
      append kids)
=> (FRED BOB KEN SUE ALICE JOE KRIS SUNSHINE JUNE)
```

Multiple clauses that do not accumulate the same type of object can coexist in a **future-common-lisp:loop** only if each clause accumulates its values into a different user-specified variable.

collect, collecting

collect[ing] *form* During each iteration, the constructs **collect** and **collecting** collect the value of the supplied form into a list. When iteration terminates, the list is returned. The argument *var* is set to the list of collected values; if *var* is supplied, the **future-common-lisp:loop** does not return the final list automatically. If *var* is not supplied, it is equivalent to supplying an internal name for *var* and returning its value in a **finally** clause. The *var* argument is bound as if by the construct **with**. A type cannot be supplied for *var*; it must be of type list.

append, appending, nconc, nconcing

append[ing] *form* [**into** *var*], **nconc**[ing] *form* [**into** *var*]

The constructs **append**, **appending**, **nconc**, and **nconcing** are similar to **collect** except that the values of the supplied form must be lists.

- The **append** keyword causes its list values to be concatenated into a single list, as if they were arguments to the function **append**.
- The **nconc** keyword causes its list values to be concatenated into a single list, as if they were arguments to the function **nconc**.

The argument *var* is set to the list of concatenated values; if *var* is supplied, **future-common-lisp:loop** does not return the final list automatically. The *var* argument is bound as if by the construct **with**. A type cannot be supplied for *var*; it must be of type **list**. The construct **nconc** destructively modifies its argument lists. The **append** construct is similar to **collect** except the values of the supplied form must be lists. These lists are not modified but are concatenated together into a single list, as if they were arguments to **append**. The argument *var* is bound to the list of concatenated values; if *var* is supplied, the loop does not return the final list automatically. The *var* argument is bound as if by the construct **with**. A type cannot be supplied for *var*; it must be of type **list**.

count, counting

count[ing] *form* [**into** *var*] [*type-spec*]

The **count** construct counts the number of times that the supplied *form* has a non-**nil** value. The argument *var* accumulates the number of occurrences; if *var* is supplied, **future-common-lisp:loop** does not return the final count automatically. The *var* argument is bound as if by the construct **with**. If **into** *var* is

used, a type can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no **into** variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default type is implementation-dependent; but it must be a *subtype* of (or integer **float**).

maximize, maximizing, minimize, minimizing

maximize | **maximizing** *form* [**into** *var*] [*type-spec*]

The **maximize** construct compares the value of the supplied *form* obtained during the first iteration with values obtained in successive iterations. The maximum value encountered is determined and returned. If **future-common-lisp:loop** never executes the body, the returned value is unspecified. The argument *var* accumulates the maximum or minimum value; if *var* is supplied, **future-common-lisp:loop** does not return the maximum or minimum automatically. The *var* argument is bound as if by the construct **with**. If **into** *var* is used, a type can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric type is supplied. If there is no **into** variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default type must be a subtype of (or integer **float**).

minimize | **minimizing** *form* [**into** *var*] [*type-spec*]

The **minimize** construct is similar to **maximize**; it determines and returns the minimum value. The **minimize** construct compares the value of the supplied *form* obtained during the first iteration with values obtained in successive iterations. The minimum value encountered is determined and returned. If **future-common-lisp:loop** never iterates, the returned value is not meaningful. The argument *var* is bound to the minimum value; if *var* is supplied, the **future-common-lisp:loop** does not return the minimum automatically. The *var* argument is bound as if by the construct **with**. The *type-spec* argument supplies the type for *var*; the default type is **fixnum**. The consequences are unspecified if a nonnumeric type is supplied for *var*.

sum, summing

sum[**ing**] *form* [**into** *var*] [*type-spec*]

The **sum** construct forms a cumulative sum of the values of the supplied *form* at each iteration. The argument *var* is used to accumulate the sum; if *var* is supplied, **future-common-lisp:loop** does not return the final sum automatically. The *var* argument is bound as if by the construct **with**. If **into** *var* is used, a type can be supplied for *var* with the *type-spec* argu-

ment; the consequences are unspecified if a nonnumeric type is supplied. If there is no **into** variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default type must be a subtype of **number**.

The loop preposition **into** can be used to name the variable used to hold partial accumulations. The variable is bound as if by the loop construct **with**. If **into** is used, the construct does not provide a default return value; however, the variable is available for use in any **finally** clause.

Local Variable Initializations in `future-common-lisp:loop`

At the time when the loop facility is invoked, the local variables are bound and are initialized to some value. These local variables exist until **future-common-lisp:loop** iteration terminates, at which point they cease to exist. Implicitly, variables are also established by iteration control clauses and the **into** preposition of accumulation.

with

with *var1* [*type-spec*] [= *form1*] [**and** *var2* [*type-spec*] [= *form2*]

The **with** construct initializes variables that are local to a loop. The variables are initialized one time only. If the optional *type-spec* argument is supplied for the variable *var*, but there is no related expression to be evaluated, *var* is initialized to an appropriate default value for its type. For example, for the types **t**, **number**, and **float**, the default values are **nil**, **0**, and **0.0** respectively. The consequences are unspecified if a *type-spec* argument is supplied for *var* if the related expression returns a value that is not of the supplied type.

Sequential and Parallel Initialization

By default, the **with** construct initializes variables sequentially; that is, one variable is assigned a value before the next expression is evaluated. However, by using the loop keyword **and** to join several **with** clauses, initializations can be forced to occur in parallel; that is, all of the supplied forms are evaluated, and the results are bound to the respective variables simultaneously.

Sequential binding is used when it is desirable for the initialization of some variables to depend on the values of previously bound variables. For example, suppose the variables **a**, **b**, and **c** are to be bound in sequence:

```
;; These bindings occur in sequence.
```

```
(loop with a = 1
      with b = (+ a 2)
      with c = (+ b 3)
      return (list a b c))
=> (1 3 6)
```

```
;; These bindings occur in parallel.
```

```
(setq a 5 b 10)
=> 10
(loop with a = 1
      and b = (+ a 2)
      and c = (+ b 3)
      return (list a b c))
=> (1 7 13)
```

The execution of the previous example of **future-common-lisp:loop** is equivalent to the execution of the following code:

```
(let* ((a 1)
      (b (+ a 2))
      (c (+ b 3)))
  (block nil
    (tagbody
      (next-loop (return (list a b c))
                 (go next-loop)
                 end-loop))))
```

```
;; This example shows a shorthand way to declare local variables
;; that are of different types.
```

```
(loop with (a b c) (float integer float)
      return (format nil "~A ~A ~A" a b c))
=> "0.0 0 0.0"
```

```
;; This example shows a shorthand way to declare local variables
;; that are the same type.
```

```
(loop with (a b c) float
      return (format nil "~A ~A ~A" a b c))
=> "0.0 0.0 0.0"
```

If the values of previously bound variables are not needed for the initialization of other local variables, an **and** clause can be used to force the bindings to occur in parallel:

```
(loop with a = 1
      and b = 2
      and c = 3
      return (list a b c))
(1 2 3)
```

The execution of the above loop is equivalent to the execution of the following code:

```
(let ((a 1)
      (b 2)
      (c 3))
  (block nil
    (tagbody
      (next-loop (return (list a b c))
                 (go next-loop)
                 end-loop))))
```

Conditional Execution in future-common-lisp:loop

If the supplied condition is true, the succeeding loop clause is executed. If the supplied condition is not true, the succeeding clause is skipped, and program control moves to the clause that follows the loop keyword **else**. If the supplied condition is not true and no **else** clause is supplied, control is transferred to the clause or construct following the supplied condition. The following keywords designate constructs that are useful when you want loop clauses to operate under a specified condition:

when, if, unless

```
{if | when | unless} form clause1 [and clause]* [end]
```

```
{if | when | unless} form clause1 [and clause]*
                             else clause2 [and clause]* [end]
```

The constructs **if** and **when** allow execution of loop clauses conditionally. These constructs are synonyms and can be used interchangeably. If the value of the test expression *form* is non-**nil**, the expression *clause1* is evaluated. If the test expression evaluates to **nil** and an **else** construct is supplied, the statements that follow the **else** are evaluated; otherwise, control passes to the next clause. If **if** or **when** clauses are nested, each **else** is paired with the closest preceding **if** or **when** construct that has no associated **else**.

The **unless** construct is equivalent to **when (not form)** and **if (not form)**. If the value of the test expression form is **nil**, the expression **clause1** is evaluated. If the test expression evaluates to **non-nil** and an **else** construct is supplied, the statements that follow the **else** are evaluated; otherwise, no conditional statement is evaluated. The clause arguments must be either accumulation, unconditional, or conditional clauses.

Clauses that follow the test expression can be grouped by using the loop keyword **and** to produce a conditional block consisting of a compound clause.

The loop keyword **it** can be used to refer to the result of the test expression in a clause. If multiple clauses are connected with **and**, the **it** construct must be the first clause in the block. Since **it** is a loop keyword, **it** cannot be used as a local variable within **future-common-lisp:loop**.

The optional loop keyword **end** marks the end of the clause. If this keyword is not supplied, the next loop keyword marks the end. The construct **end** can be used to distinguish the scoping of compound clauses.

Unconditional Execution in future-common-lisp:loop

The following loop construct evaluates its specified expression wherever it occurs in the expanded form of loop:

do, doing

The following loop construct takes one form and returns its value. It is equivalent to the clause (**do (return value)**).

return

do, doing

do[ing] [form]*

The *form* argument can be a nonatomic Common Lisp form. Each *form* is evaluated in every iteration. The constructs **do**, **initially**, and **finally** are the only loop keywords that take an arbitrary number of forms and group them as if by using an implicit **progn**.

Miscellaneous Features in future-common-lisp:loop

future-common-lisp:loop provides the **name** construct to name a loop so that special form **return-from** can be used.

The loop keywords **initially** and **finally** designate loop constructs that cause expressions to be evaluated before and after the loop body, respectively.

The code for any **initially** clauses is collected into one **progn** in the order in which the clauses appeared in the loop. The collected code is executed once in the loop prologue after any implicit variable initializations.

The code for any **finally** clauses is collected into one **progn** in the order in which the clauses appeared in the loop. The collected code is executed once in the loop epilogue before any implicit values are returned from the accumulation clauses. Explicit returns in the loop body, however, will exit the loop without executing the epilogue code.

Data Types in **future-common-lisp:loop**

Many loop constructs take a *type-spec* argument that allows you to specify certain data types for loop variables. While it is not necessary to specify a data type for any variable, by doing so you ensure that the variable has a correctly typed initial value. The type declaration is made available to the compiler for more efficient **future-common-lisp:loop** expansion. The *type-spec* argument has the following syntax:

```
type-spec::= of-type d-type-spec
```

```
d-type-spec::= type-specifier | d-type-spec . d-type-spec
```

The *type-specifier* argument can be any Common Lisp type specifier. The *d-type-spec* argument is used for destructuring, as described in the section "Destructuring in **future-common-lisp:loop**". If the *d-type-spec* argument consists solely of the types **fixnum**, **float**, **t**, or **nil**, the **of-type** is optional.

The **of-type** construct is optional in these cases to provide backwards compatibility; thus, the following two expressions are the same:

```
;;; This expression uses the old syntax for type specifiers.
(loop for i fixnum upfrom 3 ...)

;;; This expression uses the new syntax for type specifiers.
(loop for i of-type fixnum upfrom 3 ...)

;; Declare X and Y to be of type VECTOR and FIXNUM respectively.
(loop for (x y) of-type (vector fixnum)
      in 1 do ...)
```

Destructuring in **future-common-lisp:loop**

Destructuring allows binding of a set of variables to a corresponding set of values anywhere that a value can normally be bound to a single variable. During **future-common-lisp:loop** expansion, each variable in the variable list is matched with the values in the values list. If there are more variables in the variable list than there are values in the values list, the remaining variables are given a value of **nil**. If there are more values than variables listed, the extra values are discarded.

To assign values from a list to the variables **a**, **b**, and **c**, the **for** clause could be used to bind the variable **numlist** to the **car** of the supplied form, and then another **for** clause could be used to bind the variables **a**, **b**, and **c** sequentially.

```
;; Collect values by using FOR constructs.
(loop for numlist in '((1 2 4.0) (5 6 8.3) (8 9 10.4))
      for a integer = (first numlist)
      and b integer = (second numlist)
      and c float = (third numlist)
      collect (list c b a))
=> ((4.0 2 1) (8.3 6 5) (10.4 9 8))
```

Destructuring makes this process easier by allowing the variables to be bound in each loop iteration. Types can be declared by using a list of *type-spec* arguments. If all the types are the same, a shorthand destructuring syntax can be used, as the second example illustrates.

```
;; Destructuring simplifies the process.
(loop for (a b c) (integer integer float) in
      '((1 2 4.0) (5 6 8.3) (8 9 10.4))
      collect (list c b a))
=> ((4.0 2 1) (8.3 6 5) (10.4 9 8))
```

```
;; If all the types are the same, this way is even simpler.
(loop for (a b c) float in
      '((1.0 2.0 4.0) (5.0 6.0 8.3) (8.0 9.0 10.4))
      collect (list c b a))
=> ((4.0 2.0 1.0) (8.3 6.0 5.0) (10.4 9.0 8.0))
```

If destructuring is used to declare or initialize a number of groups of variables into types, the loop keyword **and** can be used to simplify the process further.

```
;; Initialize and declare variables in parallel by using the AND construct.
(loop with (a b) float = '(1.0 2.0)
      and (c d) integer = '(3 4)
      and (e f)
      return (list a b c d e f))
=> (1.0 2.0 3 4 NIL NIL)
```

A type specifier for a destructuring pattern is a tree of type specifiers with the same shape as the tree of variables, with the following exceptions:

- When aligning the trees, an atom in the type specifier tree that matches a cons in the variable tree declares the same type for each variable.
- A cons in the type specifier tree that matches an atom in the variable tree is a nonatomic *type-specifier*.

If **nil** is used in a destructuring list, no variable is provided for its place.

```
(loop for (a nil b) = '(1 2 3)
      do (return (list a b)))
=> (1 3)
```

Note that nonstandard lists can specify destructuring.

```
(loop for (x . y) = '(1 . 2)
      do (return y))

(loop for ((a . b) (c . d)) ((float . float) (integer . integer)) in
      '(((1.2 . 2.4) (3 . 4)) ((3.4 . 4.6) (5 . 6)))
      collect (list a b c d))
=> ((1.2 2.4 3 4) (3.4 4.6 5 6))
```

An error of type **conditions:program-error** is signalled if the same variable is bound twice in any variable-binding clause of a single **future-common-lisp:loop** expression. Such variables include local variables, iteration control variables, and variables found by destructuring.

Documentation Update: X3J13 Conditions

Genera 8.1 provides support for the Common Lisp Condition System, as specified by the X3J13 Committee. These functions and macros are not documented in the Symbolics documentation set. They behave as documented in version 18 of the Common Lisp Condition System, and as modified by some "cleanup" issues, including CONDITION-RESTARTS, CLOS-CONDITIONS, and BREAK-ON-WARNINGS-OBSOLETE.

Symbolics dpANS Conditions

Genera 8.3 supports the Common Lisp Condition System. The following functions and macros behave as documented in Version 18 of the Common Lisp Condition System and as modified by some "cleanup" issues, including CONDITION-RESTARTS, CLOS-CONDITIONS, and BREAK-ON-WARNINGS-OBSOLETE.

future-common-lisp:with-condition-restarts *condition restarts &body forms* *Macro*

conditions:compute-restarts

Function

Uses the dynamic state of the program to compute a list of the restarts which are currently active. See the macro **conditions:restart-bind**.

Each restart represents a function which can be called to perform some form of recovery action, usually a transfer of control to an outer point in the running program. Implementations are free to implement these objects in whatever manner is most convenient; the objects need have only dynamic extent (relative to the scope of the binding form which instantiates them).

The list which results from a call to **conditions:compute-restarts** is ordered so that the innermost (i.e., more-recently established) restarts are nearer the head of the list.

Note, too, that **conditions:compute-restarts** returns all valid restarts, including anonymous ones, even if some of them have the same name as others and would therefore not be found by **conditions:find-restart** when given a symbol argument.

Implementations are permitted, but not required, to return different (i.e., non-**eq**) lists from repeated calls to **conditions:compute-restarts** while in the same dynamic environment. It is an error to modify the list which is returned by **conditions:compute-restarts**.

conditions:restart-bind *{{(name function {keyword value}*)}*} {form}* Macro*

Executes a body of forms in a dynamic context where the given restart bindings are in effect.

name may be **nil** to indicate an anonymous restart, or some other symbol to indicate a named restart.

Function should evaluate to a function to be used to perform the restart. If invoked, this function may either perform a non-local transfer of control or it may return normally. The function may take whatever arguments you feel are appropriate; it will be invoked only if **conditions:invoke-restart** is used from a program, or if a user interactively asks the debugger to invoke it. In the case of interactive invocation, the **:interactive-function** option comes into play; see below.

The valid keyword/value pairs are the following:

:interactive-function *form*

The *form* will be evaluated in the current lexical environment and should return a function of no arguments which constructs a list of arguments to be used by **conditions:invoke-restart-interactively** when invoking this restart. The function may prompt interactively using ***query-io*** if necessary.

:report-function *form*

The *form* will be evaluated in the current lexical environment and should return a function of one argument, a stream, which prints on the stream a summary of the action that this restart will take. This function is called whenever the restart is printed while ***print-escape*** is **nil**.

conditions:find-restart *identifier* *Function*

Searches for a particular restart in the current dynamic environment.

If *identifier* is a symbol, then the innermost (i.e., most recently established) restart with that name is returned. **nil** is returned if no such restart is found.

If *identifier* is a restart object, then it is simply returned -- unless it is not currently active, in which case **nil** is returned.

Although anonymous restarts have a name of **nil**, it is an error for the symbol **nil** to be given as an *identifier*. Applications which would seem to require this should be rewritten to make appropriate use of **conditions:compute-restarts** instead.

conditions:invoke-restart *restart &rest arguments* *Function*

Calls the function associated with the given *restart*, passing any given *arguments*. The *restart* must be a restart or the non-null name of a restart which is valid in the current dynamic context. If the argument is not valid, an error of type **conditions:control-error** will be signalled. Note that restart functions call this function, not vice versa.

conditions:invoke-restart-interactively *restart* *Function*

Calls the function associated with the given *restart*, prompting for any necessary *arguments*. The *restart* must be a restart or the non-null name of a restart which is valid in the current dynamic context. If the argument is not valid, an error of type **control-error** will be signalled.

conditions:invoke-restart-interactively will prompt for arguments by executing the code provided in the **:interactive** keyword to **conditions:restart-case** or **:interactive-function** keyword to **conditions:restart-bind**.

If no **:interactive** option has been supplied in the corresponding **conditions:restart-bind** or **conditions:restart-case**, then it is an error if the restart takes required arguments. If the arguments are optional, an argument list of **nil** will be used in this case.

Once the arguments have been determined, **conditions:invoke-restart-interactively** will simply do:

```
(APPLY #'INVOKE-RESTART restart arguments)
```

This operation is used internally by the debugger and may also be useful in implementing other portable, interactive debugging tools.

conditions:handler-case *expression* *{(type ([var]) {keyword value}* {form})*}* *Macro*

Executes the given expression in a context where various handlers are active:

The *type* may be any type specifier. If during the execution of the *expression* a condition is signalled for which there is an appropriate clause, i.e. one for which (typep *condition* '*type*) is true. If there are no intervening handler for conditions

of that type, then control is transferred to the body of the relevant clause (unwinding the dynamic state appropriately in the process) and the given *var* is bound to the condition which was signalled. If no such condition is signalled and the computation runs to completion, then the values resulting from the *expression* are returned by the **conditions:handler-case**.

If more than one case is provided, those cases are made accessible in parallel. That is, in

```
(HANDLER-CASE form
  (type1 (var1) form1)
  (type2 (var2) form2))
```

if the first clause (containing *form1*) has been selected, the handler for the second is no longer visible (or vice versa).

The cases are searched sequentially from top to bottom. If there is type overlap between the cases, the earlier of the two cases will be selected.

If *var* is not needed, it may be omitted. That is, a clause such as:

```
(type (var) (DECLARE (IGNORE var)) form)
```

may be written using the following shorthand notation:

```
(type () form)
```

If there are no forms in a selected case, the case returns **nil**. Note that

```
(HANDLER-CASE form
  (type1 (var1) . body1)
  (type2 (var2) . body2) ...)
```

is approximately equivalent to:

```
(BLOCK #:G0001
  (LET ((#:G0002 NIL))
    (TAGBODY
      (HANDLER-BIND ((type1 #'(LAMBDA (TEMP)
                          (SETQ #:G0002 TEMP)
                          (GO #:G0003)))
                    (type2 #'(LAMBDA (TEMP)
                          (SETQ #:G0002 TEMP)
                          (GO #:G0004))) ...))
      (RETURN-FROM #:G0001 form))
    #:G0003 (RETURN-FROM #:G0001 (LET ((var1 #:G0002)) . body1))
    #:G0004 (RETURN-FROM #:G0001 (LET ((var2 #:G0002)) . body2)) ...)))
```

Examples:

```
(HANDLER-CASE (/ X Y)
  (DIVISION-BY-ZERO () NIL))

(HANDLER-CASE (OPEN *THE-FILE* :DIRECTION :INPUT)
  (FILE-ERROR (CONDITION) (FORMAT T "~&Foey: ~A~%" CONDITION)))

(HANDLER-CASE (SOME-USER-FUNCTION)
  (FILE-ERROR (CONDITION) CONDITION)
  (DIVISION-BY-ZERO () 0)
  ((OR UNBOUND-VARIABLE UNDEFINED-FUNCTION) () 'UNBOUND))
```

As a special case, the *type* can also be the symbol **:no-error** in the last clause. If it is, it designates a clause which will take control if the initial *expression* returns normally. In that case, the arguments to the function are like those for **multiple-value-call** on the return value of the form.

```
(HANDLER-CASE form
  (type1 (var1) . body1)
  ...
  (:NO-ERROR (varN-1 varN-2 ...) . bodyN))
```

is approximately equivalent to:

```
(BLOCK #1=#:ERROR-RETURN
  (MULTIPLE-VALUE-CALL #'(LAMBDA (varN-1 varN-2 ...) . bodyN)
    (BLOCK #2=#:NORMAL-RETURN
      (RETURN-FROM #1#
        (HANDLER-CASE (RETURN-FROM #2# form)
          (type1 (var1) . body1) ...))))))
```

conditions:restart-case *expression* *{(case-name arglist {keyword value}* {form}*)}**
Macro

The *expression* is evaluated in a dynamic context where the clauses have special meanings as points to which control may be transferred. If *expression* finishes executing and returns any values, all values returned are simply returned by the **conditions:restart-case** form. While *expression* is running, any code may transfer control to one of the clauses. See the function **conditions:invoke-restart**. If a transfer occurs, the forms in the body of that clause will be evaluated and any values returned by the last such form will be returned by the **conditions:restart-case** form.

If there are no *forms* in a selected clause, **conditions:restart-case** returns *nil*. The *case-name* may be **nil** or a symbol naming this restart. It is possible to have more than one clause use the same *case-name*. In this case, the first clause with that name will be found by **conditions:find-restart**. The other clauses are accessible using **conditions:compute-restarts**.

Each *arglist* is a normal lambda list to be bound during the execution of its corresponding forms. These arguments are used to pass any necessary data from a call to **conditions:invoke-restart** to the **conditions:restart-case** clause.

By default, **conditions:invoke-restart-interactively** will pass no arguments and all arguments must be optional in order to accomodate interactive restarting. However, the arguments need not be optional if the **:interactive** keyword has been used to inform **conditions:invoke-restart-interactively** about how to compute a proper argument list.

The valid *keyword/value* pairs are as follows:

:interactive *exp* *Exp* must be a suitable argument to the **function** special form. The expression (FUNCTION *exp*) will be evaluated in the current lexical environment. It should return a function of no arguments which returns arguments to be used by **cloe::invoke-escape-interactively** when invoking this function. This function will be called in the dynamic environment available prior to any restart attempt. It may do user interaction on the stream in ***query-io***. If a restart is invoked interactively but no **:interactive** option was supplied, the argument list used in the invocation is the empty list.

:report *exp* If *exp* is not a literal string, it must be a suitable argument to the **function** special form. The expression (FUNCTION *exp*) will be evaluated in the current lexical environment. It should return a function of one argument, a stream, which prints on the stream a description of the restart. This function is called whenever the restart is printed while ***print-escape*** is **nil**. If *exp* is a literal string, it is a shorthand for (LAMBDA (STREAM) (WRITE-STRING *exp* STREAM)).

If a named restart is asked to report but no report information has been supplied, the name of the restart is used in generating default report text. It is an error if an unnamed restart is used and no report information is provided since unnamed restarts are generally only useful interactively and an interactive option which has no description is of little value. Implementations are encouraged to warn about this error at compilation time.

When ***print-escape*** is **nil**, the printer will use the report information for a restart. For example, in CLOE, a debugger might announce the action of typing **:continue** by doing:

```
(FORMAT T "~&~S -- ~A~%" ' :CONTINUE SOME-RESTART)
```

which might then display as something like:

```
:CONTINUE -- Return to command level.
```

Note that

```
(RESTART-CASE expression
  (name1 arglist1 ...options1... . body1)
  (name2 arglist2 ...options2... . body2))
```

is essentially equivalent to

```

(BLOCK #:G0001
  (LET ((#:G0002 NIL))
    (TAGBODY
      (RESTART-BIND ((name1 #'(LAMBDA (&REST TEMP)
                            (SETQ #:G0002 TEMP)
                            (GO #:G0003))
                    ...slightly-transformed-options1...)
                  (name2 #'(LAMBDA (&REST TEMP)
                            (SETQ #:G0002 TEMP)
                            (GO #:G0004))
                    ...slightly-transformed-options2...))
      (RETURN-FROM #:G0001 expression))
    #:G0003 (RETURN-FROM #:G0001
                  (APPLY #'(LAMBDA arglist1 . body1) #:G0002))
    #:G0004 (RETURN-FROM #:G0001
                  (APPLY #'(LAMBDA arglist2 . body2) #:G0002))))))

(LOOP
  (RESTART-CASE (RETURN (APPLY FUNCTION SOME-ARGS))
    (NEW-FUNCTION (NEW-FUNCTION)
      :REPORT "Use a different function."
      :INTERACTIVE (LAMBDA ()
                    (LIST (PROMPT-FOR 'FUNCTION "Function: ")))
      (SETQ FUNCTION NEW-FUNCTION))))

(LOOP
  (RESTART-CASE (RETURN (APPLY FUNCTION SOME-ARGS))
    (NIL (NEW-FUNCTION)
      :REPORT "Use a different function."
      :INTERACTIVE (LAMBDA ()
                    (LIST (PROMPT-FOR 'FUNCTION "Function: ")))
      (SETQ FUNCTION NEW-FUNCTION))))

(RESTART-CASE (A-COMMAND-LOOP)
  (RETURN-FROM-COMMAND-LEVEL ()
    :REPORT (LAMBDA (STREAM)
              (FORMAT STREAM "Return from command level ~D." LEVEL))
    NIL))

(LOOP
  (RESTART-CASE (ANOTHER-RANDOM-COMPUTATION)
    (CONTINUE ()
      NIL)))

```

The first and second example are equivalent from the point of view of someone using the interactive debugger, but differ in one important aspect for non-interactive handling. If a handler knows about named restarts, as in:

```
(IF (FIND-RESTART 'NEW-FUNCTION)
    (INVOKE-RESTART 'NEW-FUNCTION THE-REPLACEMENT))
```

then only the first example, and not the second, will have control transferred to its correction clause since only the first example uses a restart named NEW-FUNCTION.

Here's a more complete example:

```
(LET ((MY-FOOD 'MILK)
      (MY-COLOR 'GREENISH-BLUE))
    (DO ()
      ((NOT (BAD-FOOD-COLOR-P MY-FOOD MY-COLOR)))
      (RESTART-CASE (ERROR 'BAD-FOOD-COLOR :FOOD MY-FOOD :COLOR MY-COLOR)
                    (USE-FOOD (NEW-FOOD)
                              :REPORT "Use another food.")
                    (SETQ MY-FOOD NEW-FOOD))
                    (USE-COLOR (NEW-COLOR)
                              :REPORT "Use another color.")
                    (SETQ MY-COLOR NEW-COLOR))))
    ;; We won't get to here until MY-FOOD and MY-COLOR are compatible.
    (LIST MY-FOOD MY-COLOR))
```

Assuming that USE-FOOD and USE-COLOR have been defined as

```
(DEFUN USE-FOOD (NEW-FOOD) (INVOKE-RESTART 'USE-FOOD NEW-FOOD))
(DEFUN USE-COLOR (NEW-COLOR) (INVOKE-RESTART 'USE-COLOR NEW-COLOR))
```

then a handler can then restart from the error in either of two ways. It may correct the color or correct the food. For example:

```
#'(LAMBDA (CONDITION) ... (USE-COLOR 'WHITE) ...) ;Corrects color
or #'(LAMBDA (CONDITION) ... (USE-FOOD 'CHEESE) ...) ;Corrects food
```

Here is an example using **conditions:handler-bind** and **conditions:restart-case** which refers to a condition type FOO-ERROR which was presumably defined elsewhere:

```
(HANDLER-BIND ((FOO-ERROR #'(LAMBDA (IGNORE) (USE-VALUE 7))))
  (RESTART-CASE (ERROR 'FOO-ERROR)
                (USE-VALUE (X) (* X X))))
```

→ 49

conditions:with-simple-restart (*name format-string {format-argument}**) *{form}**

Macro

This is shorthand for one of the most common uses of **restart-case**. If the restart designated by *name* is not invoked while executing *forms*, all values returned by the last form in *forms* are returned. If that restart is invoked, control is transferred to the **conditions:with-simple-restart** form, which immediately returns two values **nil** and **t**.

It is permissible for *name* to be **nil**. In that case, an anonymous restart is established.

conditions:with-simple-restart could be defined by:

```
(DEFMACRO WITH-SIMPLE-RESTART ((RESTART-NAME FORMAT-STRING
                                &REST FORMAT-ARGUMENTS)
                              &BODY FORMS)
  '(RESTART-CASE (PROGN ,@FORMS)
    (,RESTART-NAME ()
      :REPORT (LAMBDA (STREAM)
                (FORMAT STREAM ,FORMAT-STRING ,@FORMAT-ARGUMENTS))
      (VALUES NIL T))))

Lisp> (DEFUN READ-EVAL-PRINT-LOOP (LEVEL)
      (WITH-SIMPLE-RESTART (ABORT "Exit command level ~D." LEVEL)
        (LOOP
          (WITH-SIMPLE-RESTART (ABORT "Return to command level ~D." LEVEL)
            (LET ((FORM (PROG2 (FRESH-LINE) (READ) (FRESH-LINE))))
              (PRIN1 (EVAL FORM)))))))

→ READ-EVAL-PRINT-LOOP
Lisp> (READ-EVAL-PRINT-LOOP 1)
(+ 'a 3)
Error: The argument, A, to the function + was of the wrong type.
      The function expected a number.
      1: Specify a value to use this time.
      2: Return to command level 1.
      3: Exit command level 1.
      4: Return to Lisp Toplevel.
Debug>
```

Compatibility Note: In contrast to the way that Zetalisp has traditionally defined **sys:abort** as a kind of condition to be handled, we define **conditions:abort** as a way to restart.

The lowest level form which creates restart points is called **conditions:restart-bind**. **conditions:restart-case** is an abstraction which addresses many common needs for **conditions:restart-bind** while offering a more palatable syntax.

conditions:abort

Function

This function transfers control to the restart named **conditions:abort**. If no such restart exists, **conditions:abort** signals an error of type *control-error*.

The purpose of the **conditions:abort** restart is generally to allow the return to the innermost command level.

conditions:continue

Function

This function transfers control to the restart named *continue*. If no such restart exists, **conditions:continue** returns **nil**.

The **conditions:continue** restart is generally part of simple protocols where there is a single obvious way to continue, such as in **conditions:break** and

conditions:error. Some user-defined protocols may also wish to incorporate it for similar reasons. In general, however, it is more reliable to design a special purpose restart with a name that more directly suits the particular application.

conditions:muffle-warning

Function

This function transfers control to the restart named **conditions:muffle-warning**. If no such restart exists, **conditions:muffle-warning** signals an error of type **conditions:control-error**.

conditions:warn sets up this restart so that handlers of warning conditions have a way to tell **conditions:warn** that a warning has already been dealt with and that no further action is warranted.

conditions:store-value *value*

Function

This function transfers control (and one value) to the restart named **conditions:store-value**. If no such restart exists, **conditions:store-value** returns **nil**.

The **conditions:store-value** restart is generally used by handlers trying to recover from errors of types such as **conditions:cell-error** or **conditions:type-error**, where the handler may wish to supply a replacement datum to be stored permanently.

conditions:use-value *value*

Function

This function transfers control (and one value) to the restart named **conditions:use-value**. If no such restart exists, **conditions:use-value** returns **nil**.

The **conditions:use-value** restart is generally used by handlers trying to recover from errors of types such as **cell-error**, where the handler may wish to supply a replacement datum for one-time use.

conditions:make-condition *name &rest init-keywords*

Function

This function behaves like **make-condition** except that it is intended to be used in conjunction with **conditions:signal** and **conditions:error** instead of **signal** and **error**.

See the function **make-condition**.

conditions:break *&optional format-string &rest format-arguments*

Function

Prints the message described by *format-string* and *format-arguments* and then goes directly into the debugger without allowing any possibility of interception by programmed error-handling facilities.

If no *format-string* is supplied, a suitable default will be generated.

If continued, **conditions:break** returns **nil**.

Note that **conditions:break** is presumed to be used as a way of inserting temporary debugging breakpoints in a program, not as a way of signalling errors; it is expected that continuing from a **conditions:break** will not trigger any unusual recovery action. For this reason, **conditions:break** does not take the additional format control string that **error** takes as its first argument. This and the lack of any possibility of interception by programmed error-handling are the only program-visible differences between **conditions:break** and **error**. The user interface aspects of these functions are permitted to vary more widely; for example, it is permissible for a read-eval-print loop to be entered by **conditions:break** rather than the conventional debugger.

conditions:break could be defined by:

```
(DEFUN BREAK (&OPTIONAL (FORMAT-STRING "Break") &REST FORMAT-ARGUMENTS)
  (WITH-SIMPLE-RESTART (CONTINUE "Return from BREAK.")
    (INVOKE-DEBUGGER
      (MAKE-CONDITION 'SIMPLE-CONDITION
                      :FORMAT-STRING  FORMAT-STRING
                      :FORMAT-ARGUMENTS FORMAT-ARGUMENTS))))
  NIL)
```

conditions:error *datum &rest arguments*

Function

This function behaves like **error** except that the condition it signals by default when no specific condition is provided (i.e. when given a format string and format arguments) is of type **conditions:simple-error**. See the function **error**.

conditions:signal *condition &rest arguments*

Function

This function behaves like **signal** except that **nil** is returned regardless of the type of the condition. By contrast, **signal** would attempt interactive handling (usually by entering the debugger) if the condition signalled was of type **dbg:debugger-condition**. The argument conventions of **conditions:signal** permit a format string and format arguments. The condition it signals by default in that case is of type **conditions:simple-condition**. See the function **signal**.

conditions:ceerror *continue-string datum &rest arguments*

Function

This function sets up a restart named **conditions:continue**, which can be used to continue from the error. The condition it signals by default when no specific condition is provided (i.e. when given a format string and format arguments) is of type **conditions:simple-error**.

The protocol for specifying its arguments is different, in order to be consistent with the argument conventions for **conditions:signal**, **conditions:error**, and **conditions:warn**. The following examples illustrate the use of correct calls to **conditions:ceerror**:

```

(let ((x 5) (y 4))
  (conditions:error "Use ~S."
                   "Wouldn't ~S be better than ~S?"
                   x y)
  x)

(let ((x 5) (y 4))
  (conditions:error "Use ~3*~1{~S~}."
                   'conditions:simple-error
                   :format-string
                   "Wouldn't ~S be better than ~S?"
                   :format-arguments (list x y))
  x)

(let ((x 5) (y 4))
  (let ((condition (make-condition 'conditions:simple-error
                                  :format-string
                                  "Wouldn't ~S be better than ~S?"
                                  :format-arguments
                                  (list x y))))
    (conditions:error (format nil "Use ~D." x)
                      condition)
    x))

```

See the function **error**.

conditions:warn *datum &rest arguments*

Function

This function behaves like **warn** except that it sets up a restart named **conditions:muffle-warning** which can be invoked by a handler in order to suppress presentation of the warning. This function does not permit the **optional-options** argument, and the condition it signals by default when no specific condition is provided (i.e. when given a format string and format arguments) is of type **conditions:simple-warning**.

See the function **warn**.

conditions:invoke-debugger *condition*

Function

Attempts interactive handling of its argument, which must be a condition.

If the variable **conditions:*debugger-hook*** is not **nil**, will be funcalled on two arguments: the condition is being handled and the value of **conditions:*debugger-hook***. If a hook function returns normally, the standard debugger will be tried.

The standard debugger will never directly return. Return can occur only by a special transfer of control, such as the use of a restart.

conditions:ignore-errors *{form}** *Macro*

Executes its body in a context which handles conditions of type error by returning control to this form. If no such condition is signalled, any values returned by the last form are returned by **future-common-lisp:ignore-errors**. Otherwise, two values are returned, **nil** and the condition which was signalled.

Synonym for

```
(HANDLER-CASE (PROGN . forms)
  (ERROR (CONDITION) (VALUES NIL CONDITION))).
```

conditions:define-condition *name* (*parent-type*) [*{slot}**] [*option*]* *Macro*

Defines a new condition type called *name*, which is a subtype of the given *parent-type*. Except as otherwise noted, the arguments are not evaluated.

Objects of this condition type will have all of the indicated *slots*, plus any additional slots that would be available in objects of type *parent-type*. If the *slots* list is omitted, the empty list is assumed.

A *slot* must have the form:

```
{ slot-name | (slot-name) | (slot-name {slot-option value}* ) }
```

You can use these slot options:

- **:initform**
- **:initarg**
- **:reader**
- **:type**

When you use one of these options, the option you choose has the same meaning as the corresponding slot-option for **clos:defclass**. Note that if you are using a slot specifier in one of the forms without slot-options, no **:initarg** or **:reader** is provided.

The *default-value* is a form which can be evaluated by **conditions:make-condition** to produce a default value when an explicit value is not provided. If no slot default is specified, the contents of the slot will be initialized in an implementation-dependent way. It is an error to attempt to access a slot which has not been explicitly initialized and which has not been given a default value.

If the type being defined and some other type from which it inherits have a slot by the same name, only one slot is allocated in the condition object, but the specified default overrides any default which might otherwise have been inherited from a parent type. If no default value is given, the inherited default (if any) is still visible.

conditions:make-condition will accept keywords (in the keyword package) with the printname of any of the designated slots, and will initialize the corresponding slots in conditions it creates.

Accessors are created according to the same rules as used by **defstruct**.

The valid *options* are:

(:**documentation** *doc-string*)

doc-string should be a string which describes the purpose of the condition type or nil. If this option is omitted, **nil** is assumed. The information is retrieved by (documentation *name* 'type). As with **defstruct**, this sets up automatic prefixing of the names of slot accessors. Also as in **defstruct**, the default behavior is to use the name of the new type, *name*, followed by a hyphen. (interned in the package which is current at the time that the **conditions:define-condition** is processed).

(:**report** *exp*)

If *exp* is not a literal string, it must be a suitable argument to the **function** special form. The expression (FUNCTION *exp*) will be evaluated in the current lexical environment. It should return a function of two arguments, a condition and a stream, which prints on the stream a description of the condition. This function is called whenever the condition is printed while ***print-escape*** is **nil**.

If *exp* is a literal string, it is a shorthand for

```
(LAMBDA (CONDITION STREAM)
  (DECLARE (IGNORE CONDITION))
  (WRITE-STRING exp STREAM))
```

This option is processed after the new condition type has been defined, so use of the slot accessors within the report function is permitted. If this option is not specified, information about how to report this type of condition will be inherited from the *parent-type*.

The following form defines a condition of type PEG/HOLE-MISMATCH which inherits from a condition type called BLOCKS-WORLD-ERROR:

```
(DEFINE-CONDITION PEG/HOLE-MISMATCH (BLOCKS-WORLD-ERROR)
  (PEG-SHAPE :INITARG :PEG-SHAPE :READER
    (PEG/HOLE-MISMATCH-PEG-SHAPE))
  (HOLE-SHAPE :INITARG :HOLE-SHAPE :READER
    (PEG/HOLE-MISMATCH-HOLE-SHAPE))
  (:REPORT (LAMBDA (CONDITION STREAM)
    (FORMAT STREAM "A ~A peg cannot go in a ~A hole."
      (PEG/HOLE-MISMATCH-PEG-SHAPE CONDITION)
      (PEG/HOLE-MISMATCH-HOLE-SHAPE CONDITION))))))
```

The new type has slots PEG-SHAPE and HOLE-SHAPE, so **conditions:make-condition** will accept :PEG-SHAPE and :HOLE-SHAPE keywords. The accessors PEG/HOLE-MISMATCH-PEG-SHAPE and PEG/HOLE-MISMATCH-HOLE-SHAPE will apply

Here is another example. This defines a condition called MACHINE-ERROR that inherits from **error**:

```
(DEFINE-CONDITION MACHINE-ERROR (ERROR)
  (MACHINE-NAME
   :INIT-ARG :MACHINE-NAME
   :READER  MACHINE-ERROR-MACHINE-NAME))
(:REPORT (LAMBDA (CONDITION STREAM)
  (FORMAT STREAM "There is a problem with ~A."
    (MACHINE-ERROR-MACHINE-NAME CONDITION))))
```

Building on this definition, we can define a new error condition which is a subtype of MACHINE-ERROR for use when machines are not available:

```
(DEFINE-CONDITION MACHINE-NOT-AVAILABLE-ERROR (MACHINE-ERROR) ()
  (:REPORT (LAMBDA (CONDITION STREAM)
    (FORMAT STREAM "The machine ~A is not available."
      (MACHINE-ERROR-MACHINE-NAME CONDITION)))))
```

This defines a still more specific condition, built upon MACHINE-NOT-AVAILABLE-ERROR, which provides a default for MACHINE-NAME but which does not provide any new slots or report information. It just gives the MACHINE-NAME slot a default initialization:

```
(DEFINE-CONDITION MY-FAVORITE-MACHINE-NOT-AVAILABLE-ERROR
  (MACHINE-NOT-AVAILABLE-ERROR)
  ((MACHINE-NAME
    :INIT-ARG : "MC.LCS.MIT.EDU"))
  (:READER MY-FAVORITE-MACHINE-NOT-AVAILABLE-ERROR MACHINE-NAME
   ))
```

Note that since no :REPORT clause was given, the information inherited from MACHINE-NOT-AVAILABLE-ERROR will be used to report this type of condition.

conditions:ecase *keyform* {{{({key}*) | key} {form}*)}* *Macro*

This control construct is similar to **case**, but no explicit **otherwise** or **t** clause is permitted. If no clause is satisfied, **conditions:ecase** signals an error (of type **type-error**) with a message constructed from the clauses. It is not permissible to continue from this error. To supply an error message, the user should use **case** with an otherwise clause containing a call to error. The name of this function stands for exhaustive case or error-checking case.

```

Lisp> (SETQ X 1/3)
→ 1/3
Lisp> (ECASE X
      (ALPHA (FOO))
      (OMEGA (BAR))
      ((ZETA PHI) (BAZ)))
Error: The value of X, 1/3, is not ALPHA, OMEGA, ZETA, or PHI.
  1: Return to Lisp Toplevel.
Debug>

```

conditions:ccase *keyplace* *{((key)* | key) {form}*}** *Macro*

This control construct is similar to **case**, but no explicit **otherwise** or **t** clause is permitted.

The *keyplace* must be a generalized variable reference acceptable to **setf**. If no clause is satisfied, **conditions:ccase** signals an error (of type **conditions:type-error**) with a message constructed from the clauses. This error may be continued using the **conditions:store-value** restart. The argument to **conditions:store-value** is stored in *keyplace* and then **ccase** starts over, making the type tests again. Subforms of *keyplace* may be evaluated multiple times. If the **conditions:store-value** restart is invoked interactively, the user will be prompted for the value to be used.

The name of this function is mnemonic for continuable (exhaustive) case.

conditions:etypecase *keyform* *{(type {form}*)}** *Macro*

This control construct is similar to **typecase**, but no explicit **otherwise** or **t** clause is permitted. If no clause is satisfied, **conditions:etypecase** signals an error (of type **conditions:type-error**) with a message constructed from the clauses. It is not permissible to continue from this error. To supply his own error message, the user should use **typecase** with an **otherwise** clause containing a call to **conditions:error**. The name of this function stands for exhaustive type case or error-checking type case.

```

Lisp> (SETQ X 1/3)
→ 1/3
Lisp> (ETYPECASE X
      (INTEGER (* X 4))
      (SYMBOL (SYMBOL-VALUE X)))
Error: The value of X, 1/3, is neither an integer nor a symbol.
  1: Return to Lisp Toplevel.
Debug>

```

conditions:ctypecase *keyplace* *{(type {form}*)}** *Macro*

This control construct is similar to **typecase**, but no explicit **otherwise** or **t** clause is permitted.

The *keyplace* must be a generalized variable reference acceptable to **setf**. If no clause is satisfied, **conditions:ctypecase** signals an error (of type **type-error**) with a message constructed from the clauses. This error may be continued using the **conditions:store-value** restart. The argument to **conditions:store-value** is stored in *keyplace* and then **ctypecase** starts over, making the type tests again. Subforms of *keyplace* may be evaluated multiple times. If the **conditions:store-value** restart is invoked interactively, the user will be prompted for the value to be used.

The name of this function is mnemonic for continuable (exhaustive) type case.

```
Lisp> (SETQ X 1/3)
→ 1/3
Lisp> (CTYPECASE X
      (INTEGER (* X 4))
      (SYMBOL (SYMBOL-VALUE X)))
Error: The value of X, 1/3, is neither an integer nor a symbol.
  1: Specify a value to use instead.
  2: Return to Lisp Toplevel.
Debug> :1
Use value: 3.7
Error: The value of X, 3.7, is neither an integer nor a symbol.
  1: Specify a value to use instead.
  2: Return to Lisp Toplevel.
Debug> :1
Use value: 12
→ 48
```

conditions:check-type *place type &optional type-string* *Macro*

This macro behaves like **check-type** except that it sets up a restart named **conditions:store-value** which can be invoked by a handler in order to suppress presentation of the warning. See the macro **check-type**.

conditions:assert *test-form &optional places datum &rest arguments* *Macro*

This macro behaves like **assert** except that **conditions:assert** sets up a restart named **conditions:continue** which can be used to continue from the error. Instead of a format-string and format-args, it is also permissible to pass a condition argument, or a condition type and initialization arguments. For example,

```
(let ((x 1) (y 2))
  (conditions:assert (= x y) (x y)
    'conditions:simple-error
    :format-string "~S does not equal ~S."
    :format-arguments (list x y) (list x y))
```

See the macro **assert**.

Documentation of Genera 8.0 ECO #1

Overview of Genera 8.0 ECO #1

This is the first ECO to Genera 8.0 and Genera 8.0XL. The purpose of this ECO is to make the following improvements to Genera:

- Provide patches which lay the groundwork for supporting CLIM.
- Improve the integration of CLOS with Flavors.
- Include patches which support the Symbolics XL1200. These changes affect the VME interface on the XL1200 and the XL400. The revised VME documentation is included in this document. See the section "Genera 8.0 XL Documentation Update".
- Provide support for those UX400S customers who want to upgrade to SunOS 4.1. The UNIX software provided with ECO #1 supports SunOS 4.1 and does *not* run with SunOS 4.0. If you have a UX400S and are still running SunOS 4.0 on the Sun, do not load the UNIX software provided with this ECO on your UX400S. Genera and the UNIX software are completely intercompatible between 8.0 and 8.0 ECO #1: you may run Genera 8.0 ECO#1 with the Genera 8.0 UNIX software.
- Fix the problem with MacIvory Ethernet packet transmission that was corrupting packets, causing problems copying worlds and loading files via CHAOSNET.
- Fix the problem with 3600-family cart tape that caused tapes which were not re-wound to give hard tape errors.
- Speed up access to the Macintosh file system from MacIvory.
- Fix an IFEP problem with the XL1200 in Genera 8.0XL where certain errors that should have printed in the cold load stream were causing the processor to reset, forcing a warm boot.
- Fix a bug in Statice that could cause databases to become corrupted.
- Fix some other significant bugs in Genera 8.0.

Improvements and Bug Fixes in Genera 8.0 ECO #1

Improved Integration Between CLOS and Flavors

ECO #1 provides a new capability, in which CLOS generic functions can be invoked on Flavors instances. A parameter specializer name in a CLOS method can be the name of a flavor as well as the name of a class. In fact every flavor is now also a class, of metaclass **clos-internals::flavor-class**. This capability is required for CLIM. It also lays the groundwork for better integration between Static and CLOS, in that CLOS methods can specialize on Static entity handles, which are Flavors instances.

A number of small improvements in the performance and integration of CLOS have been made:

- The Inspector now works on CLOS instances.
- DW and CLOS interact better.
- Zmacs `m-` and CLOS interact better.
- **trace**, **breakon**, and **advise** now work on CLOS generic functions and methods.
- The Find Symbol command can find symbols that are defined as CLOS classes. Show Callers, List Callers (`m-l`), **who-calls** and **what-files-call** can locate callers that instantiate CLOS classes and that are methods.
- There are many improvements to CLOS performance and correctness, especially for relatively obscure corners of the language used by CLIM.
- Method combination has been completely reimplemented and now supports **:arguments** and gives names to the functions that it generates.
- A function name or function spec for a method is now a method object. Old-style (**method ...**) function specs still work.

Users who wish to take advantage of all of these performance improvements should recompile their code.

New FEP for ECO #1

Changes to the FEP flods include:

- A bug in the the NFEP disk flod has been fixed. The bug was in the disk type specification for the XT8760 disk. If a disk was formatted twice, it usually would fail within a few weeks. The XT8760 disk type has been removed, and two new disk types have been added (XT8760-24 and XT8760-25), to support the old and new configurations of this drive.
- The Set Network-Address command has been moved from the Rel-7 flod to the Lisp flod, and the Rel-7 flod has been deleted.

- Some improvements have been made in the FEP debugger.
- The default world load is now found by searching for the world with latest timestamp.
- The default microcode is that required by the default world.
- The Load World command will print the contents of **sys:*lisp-release-string*** if it is of type array. If not, the release is calculated as was done previously.
- The IFU decode rams are loaded from data in the microcode file if the associated microcode block type is present. If not, they are loaded from FEP generated data as before.

Miscellaneous Improvements and Bug Fixes in ECO #1

- The :Before keyword argument to the Show System Modifications command now assumes that the value you specify is a time in the past. This means that a specification like :Before "Tuesday" will now be correctly interpreted as last Tuesday rather than next Tuesday. The new behavior is consistent with the :Since keyword.
- The undo functions for **si:delete-ie-commands** and **si:add-ie-command** have been fixed.
- In Genera 8.0, **package-name** would sometimes return one of the package nicknames rather than the primary name of the package. This bug has been fixed.
- A patch to the Serial system has improved the performance over sync-link gateways. The performance of interactive traffic over a busy gateway has been greatly improved. Larger packets of information are automatically queued to a pending queue enabling smaller packets to be queued immediately.
- MacIvory serial I/O works much more reliably.
- A bug in the UX400S serial interface has been fixed. The bug caused UNIX authentication credentials to be held across logins.
- RPC Authentication now includes checking UNIX passwords.
- Support for SunOS 4.1 NFS automount features has been added to NFS Client.
- TCP performance has been improved by enabling adaptive TCP retransmission. The default for **tcp:*enable-tcp-adaptive-retransmission*** is now **t**.

- Dialnet has been made more robust in cases where two systems get out of synchronization while exchanging characters. This situation is now detected and the connection is aborted. Previously, the two systems exchanged useless data forever, never timing out because there were always characters available, but never making any progress because they were out of synchronization.
- Some peculiarities with incorrectly interning certain Dialnet host objects in the wrong namespace have been corrected.
- Graphics line drawing has been improved for XL machines.
- Some storage system bugs that manifest themselves as garbage collection bugs have been fixed.
- Another bug in the storage system has also been fixed. This bug could cause the system to waste some paging space. The amount of lost paging space without this ECO varies, but is larger when Dynamic or In-Place Garbage collection is used, particularly on Ivory systems. On Ivory systems which heavily use Dynamic GC, loading this ECO before the first Dynamic GC avoids wasting as much as 3000 pages of paging space.
- The use of Laserwriter functionality via Appletalk has been enhanced somewhat.
- A new function, **si:fix-fep-dpn** for fixing ECC errors on Ivory FEPs has been provided. Its behavior is exactly like **si:fix-fep-block** except that its arguments are *unit* and *page* (Disk Page Number).
- The mailer no longer blows out while trying to issue a warning if dialnet registries (obsolete in 8.0) exist. Instead, it successfully issues the warning.
- Dumping a LMFS to a cart tape on a UX400S now works correctly. Tapes written before loading this ECO, while they may show correct output when using the [List Backup Tape], were written incorrectly and cannot be restored. Note that this was only a problem if a UX400S cart tape was being used to back up the LMFS partition.
- Bugs involving unreliable operation of cart tapes on NBS machines, particularly timeouts during rewind, have been fixed.
- Support has been added for the MacinStor version 2.1 (Storage Dimensions) disk driver. Ivory disk partitions are now discovered when the Ivory is started, not when the Macintosh is started. This means that you can use the partition editor or reconfigure your disks while the Ivory is shut down and have the changes take effect without restarting the Macintosh.
- The bug that permitted you to delete a MacIvory disk partition that was in active use by the Ivory has been fixed.

- The ethernet performance problem with MacIvory IICI/IIFX has been fixed.
- The bug that caused the partition editor to give the wrong information about the amount of free space on a Macintosh volume has been fixed.
- The bug that caused **:draw-multiple-lines** on MacIvory to report errors has been fixed.

Documentation of Genera 8.0 ECO #2

Overview of Genera 8.0 ECO #2

This is the second ECO to Genera 8.0, and is also called Genera 8.0.2. Genera 8.0.2 includes Genera 8.0 ECO #1, so this documentation includes the documentation of ECO #1 changes.

Genera 8.0.2 offers two new areas of functionality:

- Software support for the UX1200S.

The UX1200S is a more powerful, higher-performance version of the UX400S. The UX1200S has the same processor architecture as the XL1200, whereas the UX400S has the same processor architecture as the XL400. See the section "Overview of the Symbolics UX1200S".

- Software support for XL1200 Single-Monitor Color Stations.

XL1200 single-monitor color stations include an XL1200 with a FrameThrower color board and a color console. This station requires only a color console, unlike other Symbolics Ivory-based color stations, which require both a color and a black-and-white console. See the section "XL1200 Single-Monitor Color Stations".

Overview of the Symbolics UX1200S

The UX1200S is a more powerful, higher-performance version of the UX400S. The UX1200S has the same processor architecture as the XL1200, whereas the UX400S has the same processor architecture as the XL400.

UX1200S Software Requirements

The UX1200S requires Genera 8.0.2. Note that the UNIX software shipped with Genera 8.0.2 to UX1200S customers is compatible with SunOS 4.1 only.

In a configuration where multiple UX boards are installed in a single Sun, each UX board runs its own copy of Genera. In a configuration that includes mixed UX400S and UX1200S boards in a single Sun, you can run the Genera 8.0.2 software on the UX1200S boards and run the Genera 8.0 or 8.0.1 software on the UX400S boards; the Genera versions are compatible.

Note that Genera 8.0.2 can also be run on UX400S boards, but Genera 8.0.1 and Genera 8.0 cannot be run on UX1200S boards.

UX1200S Documentation Update

The documentation on the UX400S pertains to the UX1200S, except as noted in this section.

On the UNIX side, `/dev/ivory0` is the device for the first UX400S board. Any additional UX400S boards are in `/dev/ivory1`, `/dev/ivory2`, and so on. Similarly, `/dev/ivory16` is the device for the first UX1200S board. Any additional UX1200S boards are in `/dev/ivory17`, `/dev/ivory18`, and so on.

Changes to the FEP in Genera 8.0.2

On Ivory machines with more than one console attached, the FEP now prints a greeting on each console. (This was already done for 3600-family machines.) In Genera 8.0.2, the FEP on Ivory machines prints the greeting, all delayed errors, and all warnings on all consoles. On each non-selected console, the FEP prints the command you can type to select that particular console.

In Genera 8.0.2, some FEP commands which were previously available only for 3600-family machines are now available for Ivory machines. These commands are the Set Console FEP command and the Set Monitor Type FEP command.

Genera 8.0.2 includes some new FEP commands: Set Disk Label and Set FEP Options. See the section "Set Disk Label FEP Command". See the section "Set FEP Options FEP Command".

In Genera 8.0.2, you can use a serial terminal to communicate with the FEP. See the section "Using a Serial Terminal to Communicate with the FEP".

XL1200 Single-Monitor Color Stations

Overview of XL1200 Single-Monitor Color Stations

XL1200 single-monitor color stations include an XL1200 with a FrameThrower color board and a color console. This station requires a color console only, unlike other Symbolics Ivory-based color stations, which require both a color and a black-and-white console. The color console includes a Sony monitor and a color console unit which typically is placed under the monitor. Three cables attach the monitor to the FrameThrower color board in the XL1200. The color console unit is connected via the normal black-and-white cable and supports the mouse, keyboard, and console serial port.

The color screen is used for all displays: device PROM, FEP, Genera, and Color. Because the FrameThrower is completely programmable, there are additional files and commands that support configuring the FrameThrower to a particular color monitor.

The device PROM and FEP use the FrameThrower in a "reduced capability" mode to simulate a black-and-white monitor. Once Genera is running, the full power of the FrameThrower is available.

Notes About Using an XL1200 Single-Monitor Color Station

This section mentions explains some things you will notice when first using an XL1200 single-monitor color station.

- Booting is slower than on non-color stations.

It takes longer to boot a color world because it is a larger world, and because it takes time for Genera to fully initialize the FrameThrower.

- Memory is used up more quickly than in a non-color station.

Think about what happens when you switch windows, by using the `SELECT` key. For example, you are in the Lisp Listener and you enter `SELECT E` to go into the Zmacs editor. Genera saves a copy of the Lisp Listener window; this is called a bit-save array. Genera draws a saved copy (or creates and draws a new copy) of the Zmacs window, another bit-save array. In a color station, a bit-save array is eight times larger than in non-color stations, because one of the dimensions (call it depth) of the array is 1 in non-color and 8 in a color station. Therefore, switching windows uses up a lot of virtual memory.

Some suggestions for coping with this problem are to keep the Dynamic Garbage Collector turned on, and to run a nightly GC with GC cleanups.

Note that the color software makes more intelligent use of the FrameThrower than does Genera, so using the color software features does not use up memory as much as using Genera does.

- Background greying is disabled.

In a non-color station, when you switch from the Lisp Listener window to a Zmacs window, the Zmacs window appears and the portion of the screen which is not covered by the Zmacs window (the right-hand margin of the screen) is greyed. This background greying does not happen on an XL1200 single-monitor color station, because it would be expensive in terms of using up memory. Genera performs background greying by saving a copy of the window and then saving another copy which is a greyed version of the former copy. For reasons discussed above, these bit-save arrays would use up a lot of memory on a color station.

- Flashing the screen takes a long time the first time.

When a **beep** causes the screen to flash, for example when you receive a Notification or a Converse message, you will notice a very long beep and the screen going completely blank for a noticeable period of time. Just be patient and wait, and the screen will be repainted normally. Subsequent beeps and screen flashes do not last as long.

Editing the Disk Label

Edit Disk Label Command

Edit Disk Label *unit-number*

Edits the disk label of the disk identified by *unit-number*. Brings up a menu of choices, which you can click on to change aspects of the disk label, such as the FEP kernel, backup FEP kernel, and the color system startup file (used for color systems). You can also click on choices to deinstall the current FEP kernel, deinstall the FEP backup kernel, and deinstall the color system startup file.

Just as the device PROM finds the FEP kernel by reading the disk label, it also needs to find the file where the FrameThrower color system startup programs are stored. Note that the device PROM does not understand the FEP file system; it just reads the disk label, which points to the correct file.

unit-number{*integer*, All} The number identifying the FEP disk whose disk label you want to edit. All allows you to edit the labels of all the disks attached to the machine.

Normally, the FEP kernel and FEP backup kernels are installed automatically, when you use the Copy Flod Files command. Copy Flod Files installs the new FEP kernel and installs the previous FEP kernel as the backup FEP kernel. Only in unusual debugging circumstances do you need to deinstall the FEP kernel or FEP backup kernel by using Edit Disk Label. When you deinstall the FEP kernel, the FEP backup kernel is installed as the FEP kernel. When you deinstall the FEP backup kernel, the FEP kernel is installed as the FEP backup kernel. (Note that the FEP kernel and FEP backup kernel must always be present, in order for this to work properly.) Thus, when you deinstall either the FEP kernel or FEP backup kernel, the result is that a single FEP kernel is installed as the FEP kernel and the FEP backup kernel; in other words, there is no separate FEP backup kernel. When you next use Copy Flod Files, the new kernel will be installed as the FEP kernel, which means you will again have a different FEP kernel and FEP backup kernel.

FEP Commands Useful for XL1200 Single-Monitor Color Stations

The following FEP commands are particularly useful for XL1200 single-monitor color stations:

- Set Disk Label
- Show Disk Label
- Set Console
- Set Monitor Type
- Set FEP Options

Set Disk Label FEP Command

Set Disk Label *unit-number keywords*

<i>unit-number</i>	A disk unit (must be a number in base 10).
<i>keywords</i>	:Color System Startup File, :FEP Kernel, :Query
:Color System Startup File	Pathname of the file where the color system startup programs are stored.
:FEP Kernel	Pathname of the file where the FEP kernel is stored.
:Query	{Yes, No} Whether the system should ask for confirmation before setting the disk label. The default is No.

The Set Disk Label command is resident in the FEP, so it needn't be loaded from an overlay (flod) file.

Show Disk Label FEP Command

Show Disk Label *unit-number*

Displays the information in the disk label.

unit-number A disk unit (must be a number in base 10).

The Show Disk Label FEP command is resident in the FEP, so it needn't be loaded from an overlay (flod) file.

Set Console FEP Command

Set Console *console keywords*

Allows you to select between the available consoles. For 3600-family machines with CadBuffer2 hardware, this command also switches input from the console's display hardware to the CadBuffer2 keyboard. To make Lisp notice that the console has been changed, you need to reload microcode (on 3600-family machines) and warm or cold boot (on both 3600-family and Ivory machines) after using the Set Console FEP command.

<i>console</i>	On 3600-family machines, the <i>console</i> is Color or Monochrome. On Ivory-based machines, the <i>console</i> can be any of the available and enabled consoles; press HELP for a list of choices.
<i>keywords</i>	:Clear Screen, :Cold Load Too
:Clear Screen	{Yes, No} Whether to clear the screen before selecting it. The default is No.
:Cold Load Too	{Yes, No} Whether Lisp's cold-load stream (and Main console if you warm boot) should be redirected there also. The default is Yes.

The Set Console FEP command is resident in G208 and greater versions of the FEP EPROM, and in I322 or greater versions of the IFEP kernel, so it needn't be loaded from an overlay (flod) file.

Set Monitor Type FEP Command

Set Monitor Type *console type* (for 3600-family machines with G208 or greater FEP EPROMs, and for Ivory machines with I322 or greater FEP kernels)

Set Monitor-Type *console-name* (for machines with V127 and G206 FEP EPROMs, and for Ivory machines with I321 or less FEP kernels)

Users of 3600-family machines should use the Set Monitor Type FEP command when installing a monitor that differs from the one specified by the machine's hardware.

On 3600-family machines with G208 or greater FEP EPROMs, sets the console type and name. On 3600-family machines with V127 and G206 FEP EPROMs, sets the console name. This command also loads the appropriate sync program into the machine's display controller, CadBuffer, or CadBuffer2 hardware.

Users of Ivory-based machines should use the Set Monitor Type FEP command if the FEP options installed by the Set FEP Options FEP command are incorrect. At your first opportunity after using the Set Monitor Type command, you should use Set FEP Options to correct the FEP options, and reset the FEP to make the new FEP options take effect.

<i>console</i>	A particular console; this argument defaults to the current console. Use HELP to find out which consoles are applicable. On Ivory machines, a console is specified by three elements: the console type, the sync program, and the unit number. On 3600-family machines, a console is specified as being color or monochrome
<i>console-name</i>	The list of choices depends on the <i>console</i> . Use HELP to find out which choices are applicable.

The Set Monitor Type FEP command is resident in the FEP, so it needn't be loaded from an overlay (flod) file.

Set FEP Options FEP Command

Set FEP Options *keywords*

Sets options which are used by the FEP.

keywords :Color System Number, :Color System Startup Program, :Color System Type, :Serial Console Type

The Color System keywords are used only when you are using a custom color monitor (other than the default Sony monitor).

- :Color System Type
 {None, FrameThrower} FrameThrower enables the FEP to use the color monitor. None disables the use of the color monitor.
- :Color System Number
 The number of the color system, a decimal integer between 0 and 255. (You can have more than one FrameThrower, and the color system number enables you to distinguish among them.)
- :Color System Startup Program
 Which color system startup program in the color system startup file for the color system to use.
- :Serial Console Type
 {None, ASCII, or X3.64} None prevents the FEP from ever using the serial console; this is appropriate if you have a serial device other than a console connected to the serial port, and know you won't use the serial device as a console. ASCII indicates that the serial console is a dumb terminal. X3.64 indicates that the serial console is an ANSI-standard X3.64 terminal, such as a VT100.

The new options you specify in Set FEP Options take effect when you next reset the FEP with the Reset FEP FEP command.

The Set FEP Options FEP command is resident in the FEP, so it needn't be loaded from an overlay (flod) file.

Troubleshooting an XL1200 Single-Monitor Color Station

Because the XL1200 single-monitor color station uses the FrameThrower as its only display, and because the FrameThrower is both very flexible and complex, there is a greater possibility of problems in using these stations than in monochrome systems or color stations with two monitors. This section describes some common problems, and steps you can take to solve them before calling Customer Service.

Troubleshooting the Power-up and Initialization of an XL1200 Single-Monitor Color Station

Below, we describe the expected steps that would result in successfully powering on and initializing an XL2100 single-monitor color station to familiarize you with what could go wrong at each step:

1. When power is applied by depressing the on/off button on the front panel, the green POWER light in the on/off button should illuminate and the yellow FAULT light in the reset button should illuminate.

You should hear the disks spin up; usually there is a "click" (about two seconds after power is applied) followed by a "whirr" that increases in pitch (lasting for about 10 seconds), and finally a "clunk" as the heads are loaded. The FAULT light indicates that the processor is uninitialized at this point. (If the FAULT light comes on later in the process, it indicates that there has been a fatal error. The processor will attempt to re-initialize itself and restart.)

What can go wrong: If neither the POWER nor FAULT light is illuminated, or you don't hear the disks spin up, you should suspect your power connections.

What to do: Turn off the system, recheck your power connection, and retry. Note that there is a master power switch on the back of the chassis. This switch must be in the 1 position and should be illuminated (indicating power is available).

2. The booting process is a three-stage process. First, the "Boot" program (in ROM on the processor board) initializes the processor and searches for a "Device" program (in ROM on the I/O board) that can load the "FEP" program from one of the attached peripherals. Second, the Device program examines the attached peripherals looking for a console and a device from which to load the color startup program for the FrameThrower (if the console is a FrameThrower console). Finally, the Device program loads the FEP program from disk or tape and the FEP program will start.

In the second stage, the Device program must go through a number of steps to determine how to initialize the console:

- a. First it reads the "Switchpack" from the color console unit. This switchpack has two relevant settings (these should be checked by your Customer Service Engineer on installation and rechecked if the system appears to operate incorrectly).

Switch	Function
1	If on, use the color console. If off, the color console is ignored. Normally, it should be on; off is used for diagnostic purposes only
5	If on, use the default monitor type and number: Sony (1024x1280 CADBuffer) 0 If off, a custom monitor type and number is determined by the setting of the FEP Options (as registered by the Set FEP Options FEP command)

(The remaining switches control the operation of the console unit, setting various diagnostic modes; these should be left as they are.)

What can go wrong: If the console-unit is not plugged in properly or the switchpack is set improperly, the Device program will not attempt to use the color console.

What you should do: In early models of the XL1200 single-monitor color station, the console switches are occasionally mis-read. You can try pressing the RESET button or power-cycling the machine, which will cause the Device program to be restarted and the switches to be re-read. You should also check that your console unit is properly plugged in. If it is not plugged in, the Device program will not be able to read the switches; by default it assumes that it should *not* use the color console if it cannot read the switches (this is because using a console with an incorrectly guessed monitor type can physically damage the monitor).

- b. If the switchpack setting indicates that the color console is to be used, the Device program will probe the VMEbus looking for the FrameThrower hardware. (It does not automatically scan the VMEbus, as there may be some other peripheral board in the address space normally occupied by the FrameThrower that should not be randomly written or read.)

What can go wrong: The FrameThrower system may not be found at the expected VME address. The Device program can't use the FrameThrower if it can't find it. The XL1200 processor may not be in VME slot 1. The processor may hang during VME operations if it is not in slot 1.

What you should do: You can power down your system and attempt to re-seat the FrameThrower, XL1200 Processor, and other boards. Note that the XL1200 processor board *must* be in VME slot 1 (the left-most slot as you face the back of the chassis) for it to be able to detect the FrameThrower board. Your Customer Service Engineer can verify the settings of all jumpers and switches on the FrameThrower, XL1200 processor, and other boards.

- c. If the switchpack setting indicates that the color console is to be used and the FrameThrower hardware is located by the peripherals search, the Device program then looks for the startup program file in the disk label. (Shortly after the "clunk" of the disk heads loading, the Device program is able to use the disk. You should hear a "rattle" or "buzz" as it accesses the disk looking for and loading the startup program. This occurs from 12 to 20 seconds after power-on. Note that if you press the RESET button rather than power-cycling, the disks do not have to spin up or load, hence you may not hear any disk noises.)

If the switchpack indicates the default monitor type is to be used, any startup program found on disk will be accepted and the default monitor type parameters extracted from it and loaded. If the switchpack indicates a custom monitor type, the FEP Options are read from NVRAM and the Device program searches the startup program for the matching type. When an appropriate program is found, default or otherwise, the Framethrower console is initialized (you may see it flash as the sync is loaded) and the Device program greeting is displayed:

```
"Autoloading the IFEP Kernel -- type any character to abort"
```

It takes approximately 21 seconds from the time power is applied until the display is initialized, under normal conditions. If you press the RESET button, the display should initialize in approximately 8 seconds, since the disks are already spun up.

What can go wrong: If the disk label is unreadable, does not have a startup program, or (for custom monitor types) does not have a matching entry, the Device program will not be able to initialize the console.

What you should do: The Device program will look for and load a startup file from the tape drive if there is a tape in the unit. If you suspect the disk label or a corrupted disk, you can try inserting your IFS tape and pressing the RESET button on the front panel. The tape will spin because of the reset, but it should spin a second time when the Device program searches it for the startup program.

NOTE: In a correctly configured station, the Device program does not need to use the console and will automatically load the FEP from the default disk unit. Thus, you need not consider it a problem if the console appears uninitialized at this point, unless you need to interact with the Device program to request loading of a different FEP or loading from tape. See the section "Using a Spare Monitor to Troubleshoot an XL1200 Single-Monitor Color Station".

You can verify the settings in the disk label by using the Show Disk Label FEP command, and change them by using the Set Disk Label FEP command.

3. If the console has been properly initialized by the Device program, you will see the standard Device program dialog as it initializes the disks and looks for the FEP kernel on one of the disks to load. If the Device program was unable to initialize the console, it will still look for and load the FEP kernel. In either case, you should hear the "rattle" or "buzz" of the disk again as the FEP kernel is loaded. The FEP kernel will go through approximately the same procedures as the Device program in attempting to initialize the console, with the exception that it will attempt to initialize all consoles it can find and then choose one console as the default console.

When the FEP succeeds in initializing any console it will print out its standard greeting on that console:

Type "Hello" to initialize the FEP's command databases, etc.

If the console is not the default FEP console, you will see a message to the effect:

Type "Set Console FrameThrower Sony console 0" to select this console

What can go wrong: As with the Device program, the switches on the console unit may be wrong or mis-read. In this case the FEP may not choose it as the default console, however, unlike the Device program, the FEP initializes all the consoles it can find, so you do have a chance to manually tell it to use a particular console, even if the switches are wrong or mis-read.

What you should do: If you see the message about "Set Console" on your color monitor, which indicates the FEP decided not to use the color monitor as the default console, you can simply type the suggested command on the color monitor to get the FEP to use it. Note that because the FEP supports command completion, unless you have more than one of a particular type of console it is usually sufficient to only type the first few letters of a console type to select it. For instance, typing "Se C F" and pressing Return is usually sufficient to select the FrameThrower console.

4. Following the standard FEP greeting will be any errors that occurred during initialization and any warnings about boards or peripherals that are not up to date.

What can go wrong: If you see a message starting "*** WARNING ***" the FEP has determined that some component of your system is out of date and must be upgraded to work properly. This is unlikely to occur if you have a new system, but if you have had your system upgraded, it may be that some component was missed.

What you should do: It is important that you have all components updated to the proper revision level for reliable operation. You should contact Symbolics Customer Service to have any components that are out of date upgraded as soon as possible. You run the risk of faulty operation of the system if you proceed with out-of-date components.

5. When the FEP program has been successfully loaded and started and is ready to accept commands from the keyboard, the yellow FAULT light will go out. This takes approximately 37 seconds from the time the system was powered on, or about 24 seconds from the time the system was reset.

What can go wrong: You might see the yellow FAULT light go off but still have no display on the color monitor. This can be for the same reasons that the Device program could not initialize the console.

What you should do: If neither the Device program nor the FEP can initialize the console, you should go back and check the suggestions under #2. If none of those suggestions are helpful, it might be that the FEP is unable to determine the type of monitor that is connected to the FrameThrower, either because the switches could not be read or because the switches are set incorrectly. In this case, you can type "blind" to the FEP to tell it the type of monitor.

NOTE: the Set Monitor Type FEP command is different in the I322 FEP from previous Ivory FEP versions. It takes two arguments now, a console whose type to set and a type. The first argument defaults to the current console, so if you are typing blind you need to type the following commands.

Whenever you are typing "blind", it is useful to press CLEAR-INPUT and RETURN before you type any commands. Since you cannot see whether there was already some input, this clears any that might be there, so the commands that follow are interpreted from scratch.

```
Press CLEAR-INPUT
Press RETURN
```

Now give a command to ensure that the FrameThrower is selected as the current console. To give the FEP command "Set Console FrameThrower", type the following characters, including pressing the SPACE key and RETURN key where indicated:

```
se SPACE c SPACE f RETURN
```

To give the FEP command "Set Monitor Type *this-console* Sony", type the following characters, including pressing the SPACE key and RETURN key where indicated:

```
s SPACE m SPACE t SPACE SPACE sony RETURN
```

In the command above, note the two SPACES after the letter "T": the first SPACE causes T to be completed to "Type" while giving the Set Monitor Type command. The second SPACE causes the default to be used for the first argument, which is the current console.

What can go wrong: The Device Program makes some simplifying assumptions that may allow it to choose and initialize the color console, but when

the FEP is loaded you may find the color console is not initialized. This is due to a mis-setting of NVRAM options on the processor board.

What you should do: The command to enable the color console is:

```
Set FEP Options :Color System Type FrameThrower
```

The minimum you can type (for typing blind) is:

```
s SPACE f SPACE o SPACE :c SPACE s SPACE t SPACE f RETURN
```

After typing this command, either power-cycle or reset the system and the FEP should now initialize the color monitor.

Boot Procedure for XL1200 Single-Monitor Color Stations

When an XL1200 single-monitor color station boots, the following steps happen:

1. Determining which console should be active

The device PROM queries the hardware to determine which consoles are attached. The possibilities include: a color console, serial console, and a black-and-white console. If more than one console is attached, the device PROM makes a decision about which console should be active. The active console will have the FEP Command: prompt. Other attached consoles will display a message about what to type (the Set Console command) to make that console active.

The color console is active if the FEP notices that the FrameThrower board and the color console unit are present. If the color console is not active, then the black-and-white console is active. If the black-and-white console is not attached, then you won't see the FEP Command: prompt on any of the attached consoles.

2. Loading the color system startup file for the color console

The color system startup file contains a set of color system startup programs. A color system startup program is necessary for initializing the color system. If a color console is attached, the device PROM looks at the disk label to find out where the color system startup file is stored. (The information in the disk label is updated by the Edit Disk Label command, and is also updated by the Copy Flod Files command.)

2A. If the color system startup file is found, it is loaded, and the monitor will be activated. In this case, the next step is 3A.

2B. If the color system startup file is not found, the XL1200 can still boot, but the monitor will not show any display at this point. In this case, the next step is 3B.

3. Loading the FEP

3A. The FEP is loaded and it comes up on the color monitor. In this case, the next step is 4A.

3B. The FEP is loaded, but because the color system startup file was not found, the monitor cannot show any display. In this case, the next step is 4B.

4. Booting Genera

4A. If the XL1200 is set up to autoboot, it will boot itself. Otherwise, you can give the FEP commands to boot the machine.

4B. If the XL1200 is set up to autoboot, the machine will boot even though you cannot see anything on the monitor. You can also give FEP commands from the keyboard, and they will work, even though you cannot see them on the monitor.

Using a Spare Monitor to Troubleshoot an XL1200 Single-Monitor Color Station

If you are able to plug a spare black-and-white monitor into your system in place of the color console unit, you may be able to discover what is preventing the color console from being used. Both the device program and the FEP will print out diagnostic information on the black-and-white monitor when searching for and attempting to initialize the color monitor (if the black-and-white monitor is connected in place of the color console unit).

You can then use the FEP to fix the disk label to point to a valid color startup file, you can set the NVRAM options to enable the color console, and you can experiment switching back and forth between the black-and-white and color consoles. Note that both consoles will use the keyboard attached to the black-and-white monitor.

If you are not able to plug in a spare black-and-white monitor, it is still possible you can type "blind" to the FEP to enable the color console, by typing carefully and using command completion. You will know that the FEP is ready to accept commands when the yellow FAULT light on the front panel goes out.

After verifying that the color monitor is enabled (Set FEP Options :Color System Type FrameThrower) and that the proper color startup file is installed in the disk label, you can RESET or power-cycle the system. Note that if you connected a black-and-white monitor, and leave it connected, the FEP will still not initialize the color monitor, because it cannot read the switches from the color console unit. You can cause the FEP to initialize the color monitor anyway, even with the black-and-white monitor by using another NVRAM setting:

```
Set FEP Options :Color System Startup Program Sony :Color System Number 0
```

The FEP will still not choose the color console as the default console, as long as the black-and-white monitor is connected, but you can use the black-and-white monitor to debug your setup and then use the Set Console FEP command to switch to the color console before you start Lisp.

When you are satisfied that everything is working, disconnect the black-and-white monitor, reconnect the color monitor, and reset the system. (If Lisp is running, you can Halt Machine and then use the RESET button to re-initialize the FEP. You can then warm-boot Lisp and it will use the color console).

Using a Serial Terminal to Communicate with the FEP

You can use a serial terminal to communicate with the FEP. One case in which this can be particularly useful is in troubleshooting an XL1200 single-monitor color station. For example, if the color monitor cannot come up, you can connect a serial terminal to the serial port on the color console unit, and use that terminal to give FEP commands such as Show Disk Label, or Set FEP Options.

In addition to connecting the serial terminal physically, you need to give the Set Console FEP command to tell the FEP to use the serial console. You might need to also use the Set Monitor Type FEP command to tell the FEP whether the serial console is ASCII (a dumb terminal) or X3.64 (such as a VT100).

You need to set the serial terminal's parameters for 9600 baud, 8 bits, and no parity.

Keep in mind that serial terminals don't have all the special keys of the Symbolics keyboard. If you need to transmit special Symbolics characters, such as Meta or Super characters, you need to understand how serial terminal keys are mapped to the Symbolics keys.

Mapping of Serial Terminal Keys to Symbolics Keys

The keyboard of a serial terminal does not have the same set of keys as does the Symbolics keyboard. For example, such a keyboard typically lacks a Meta key, Super key, Hyper key, and Symbol key. These keyboards do, however, have a Control key, however, most such keyboards can handle only Control-A, Control-Z, and a few other characters, most of which are reserved for escapes.

Accessing the Symbolics Character Set

The following characters may be used to access the Symbolics character set:

```
c-^ = Toggles the Control bit    c-] = Toggles the Super bit
ESC = Toggles the Meta bit       c-\ = Toggles the Hyper bit
c-@ = Toggles the Shift bit
```

For example, to enter c-m-C, you need to set both the Control bit and the Meta bit, by entering c-^ and ESC; you can then press C to enter c-m-C.

Similarly, you might need to enter `c-sh-C`. The serial keyboard has both a Control key and a Shift key, but you cannot press them both at once to enter `c-sh-C`. You can enter `c-^` to set the Control bit, then press the Shift key while typing C. Or, you can enter `c-@` to set the Shift bit, and then press the Control key while typing C.

Entering Special Symbolics Keys

The character `c-_` (that is, the Control key and the underscore `_` key) is used as a prefix to enter special characters as follows:

H = <Help>	L = <Line>
E = <End>	P = <Page>
A = <Abort>	F = <Refresh>
S = <Suspend>	B = <Back-Space>
R = <Resume>	N = <Network>
C = <Complete>	1 = <Square>
I = <Clear-Input>	2 = <Circle>
X = <Escape>	3 = <Triangle>

For example, if you press `c-_` followed by H (that is, two keystrokes) on the keyboard of a serial terminal, you get the effect of the HELP key on a Symbolics keyboard.

Entering Symbol Characters

`c-_ _` is the prefix for Symbol characters. (That is, Control Underscore followed by Underscore, two keystrokes.)

For example, you can enter `c-_ _ P` (that is, three keystrokes) to get the effect of SYMBOL-P.

`c-_ ?` displays the `c-_` dispatch table.

Genera 8.0 XL Documentation Update

This section pertains to XL-family machines only.

Changes to the VMEbus Interface in Genera 8.0 XL

- There are two new slave buffer variables, **sys:*vme-slave-buffer-base*** and **sys:*vme-slave-buffer-end***. They are the starting and ending addresses of the slave buffer in words, not bytes.
- The slave buffer base address is now settable in the FEP via the `:Slave-Buffer Base` keyword to the Set Boot Options command. This address is in bytes. The system defaults are correct unless the jumpers on the processor board have been moved.

- To let an application allocate a portion of the slave buffer, use (**sys:allocate-slave-buffer-memory** *:name :words &key from-end*). The function returns a starting and ending address in words. There is no enforcement, but this a simple check-out scheme for slave buffer memory so you do not accidentally use memory allocated by the system (as on the UX400) or by another application (like FrameThrower). Specific allocations can be added to an initialization list. System allocations take place on the system initialization list.
- Flonum data tagging is not supported on the XL1200.
- There are two new functions to enable and disable bus interrupts: **sys:enable-bus-interrupt** and **sys:disable-bus-interrupt**. They take a bus interrupt level from 1 to 7. They are somewhat easier to use than **sys:logior-bus-interrupt-mask** and **sys:logand-bus-interrupt-mask**.
- **sys:install-bus-interrupt** has been changed. It now takes a status ID rather than a level. Interrupt functions are dispatched by the status ID rather than the interrupt level. This change is incompatible with Genera 8.0.
- The default slave buffer address for the XL1200 is #xFAC00000.
- **sys:make-bus-address** of an address within the range of the slave buffer addresses will create the appropriate Ivory address to access the slave buffer. This is based on the variables **sys:*vme-slave-buffer-base*** and **sys:*vme-slave-buffer-end***.
- You can initiate VME SYSReset by calling the function **cli::merlin-ii-sysreset**. **Warning:** The effect of using this function is the same as pressing the RESET button on the machine.
- **sys:bus-read** and **sys:bus-write** now update the system bus parameters such as address-modifier and release-mode.
- The slave buffer on the XL1200 processor is 256K words long. Reservations of slave buffer memory are handled by **sys:allocate-slave-buffer-memory**.
- VMEbus errors on the XL1200 are not signalled on writes. Reads receive errors, as do polled reads using **bus-read**.
- The VMEbus interrupt handler does not disable the interrupt level for an incoming interrupt. The user process must disable and reenale the interrupt, if necessary.
- VMEbus interrupt status-ID #x5F on the XL1200 is reserved for Symbolics use. The Slave Trigger Interrupt uses this vector.

VMEbus Interface

Introduction to the XL-Family VMEbus Interface

The XL-family system is based on a 7-slot, 9U form factor VMEbus backplane and card cage. The VMEbus is a versatile, standard 32-bit bus which provides power and clock distribution, asynchronous data transfer, and interrupt delivery and acknowledgment. Although the performance requirements of modern processor/memory interconnects have exceeded the design range of the VMEbus, it remains popular as a peripheral I/O bus, and is often used in tandem with a separate high-speed memory bus. This is the strategy used in the XL: a private 48-bit bus connects the processor with its memory and I/O board, and the VMEbus interface is used to communicate with optional I/O peripherals and other 32-bit wide devices.

The XL processor is a VMEbus master, meaning that it can issue requests to read and write locations in other VMEbus cards, and can deliver interrupts. The interface provides a flexible *polled access* facility with which nearly all possible data transfer operations may be performed. It also provides a *direct-access* facility with which a portion of the VMEbus address space may be mapped into the XL's physical address space, for high-speed access to 32-bit slaves.

The XL processor is also a VMEbus slave, meaning that other bus masters can issue requests to read and write locations in it, and that it can receive interrupts issued by other bus masters. However, other bus masters may not directly access the XL processor's main memory; they may access only a dedicated memory in the XL VMEbus interface called the *slave buffer*. This design eliminates the hardware complication of arbitration deadlock between the VMEbus and the XL private bus, and eliminates the software complication of negotiating with the Genera virtual memory system to reserve contiguous portions of main memory.

The VMEbus is a flexible bus with many options and modes. In the design of the XL VMEbus interface, particular attention was paid to optimizing both the master and slave for high speed 32-bit data transfer, but the interface supports nearly all possible modes and can accommodate virtually any VMEbus device. The interface hardware includes the following features:

- On the XL400, the slave appears on the VMEbus as a 32K by 32-bit memory. On the XL1200, the slave is one megabyte in size.
- The master can transparently map up to 267 megawords of VMEbus address space into the Ivory physical address space (for 32-bit transfers only).
- Both the master and slave implement 8, 16, 24, and 32-bit data transfers, including transfers not aligned on an address boundary.
- Both the master and slave may specify that data be shuffled to compensate for differences in system bit, nibble, or byte ordering.

- Both the master and slave may request that data returned to Ivory be tagged as either integers or IEEE 32-bit floating-point numbers on the XL400. (On the XL1200, data returned can only be tagged as integers.)
- The master may specify an arbitrary address modifier for a data transfer.
- The arbitration parameters (arbitration level, bus release behavior) of the master are programmable.
- The interface can issue and receive all seven interrupt levels, and supports 8-bit interrupters.
- The slave implements the VMEbus block transfer protocol.
- The interface includes a system controller (containing VMEbus arbiter, clock drivers, and so on), which may be disabled by a jumper.

The interface hardware does not support the following features:

- The master does not implement the VMEbus block transfer protocol, but uses address pipelining to achieve equivalent performance.
- The master cannot issue ADDRESS-ONLY data transfer requests.
- The master cannot issue READ-MODIFY-WRITE data transfer requests, but atomic operations are supported by inhibiting bus release.

Software provided with Genera supports efficient access to all interface features, while shielding the client software from irrelevant hardware details. Data transfers may be performed in isolation via function calls that read or write specified bus locations, or by obtaining a physical address that the interface will map to a range of VMEbus addresses, or by using a Lisp indirect array which may be manipulated by normal array operations and facilities such as BITBLT. Full support for delivering and handling interrupts is provided.

For more information about the VMEbus hardware specification, see "The VMEbus Specification", Revision C.1, published by Printex. For more information about the electrical and mechanical characteristics of the XL VMEbus card cage, contact your Symbolics sales representative.

VMEbus Data Transfers

There are four basic techniques for performing VMEbus data transfers:

- Isolated transfers may be performed by calling the functions **sys:bus-read** and **sys:bus-write**, specifying the desired VMEbus address (and perhaps data) and any optional parameters. This technique is the most flexible, since it uses the polled access hardware in the interface which supports non-32-bit transfers.

However, it incurs some overhead programming the hardware and is therefore not very efficient.

- The function **sys:make-bus-address** may be called to map a portion of the VMEbus into the Ivory address space, and return a physical address pointing to it. That address may be manipulated using subprimitives such as **sys:%pointer-plus**, **sys:%p-ldb**, **sys:%p-dpb**, **sys:%block-read**, and **sys:%block-write**. This technique is very efficient, but is rather cumbersome. It also works for 32-bit slaves only.
- The function **sys:make-bus-array** may be called to map a portion of the VMEbus into the Ivory address space, and return an indirect array pointing to it. This allows high-level, bounds-checked access to array elements of any type, and the array may also be passed to Lisp facilities such as **bitblt**. This technique also works for 32-bit slaves only.
- Atomic operations may be performed by calling **sys:bus-store-conditional**, which works for VMEbus locations the same way **store-conditional** works for virtual memory locations.

All these techniques provide some way to configure the interface hardware to enable options such as bit shuffling, nonstandard address modifiers, and arbitration parameters. For polled transfers (via **sys:bus-read** and **sys:bus-write**), the options are specified as simple keyword arguments. For direct transfers (via **sys:make-bus-address** and **sys:make-bus-array**), most of the options are specified by the **sys:with-bus-mode** macro, which must surround any use of VMEbus addresses. See the section "Summary of VMEbus Transfer Options" for a description of the available options.

In general, clients should use polled transfers to refer to isolated registers on the VMEbus, and direct transfers to map memories, frame buffers, large register banks, etc., into the Ivory physical address space. See the section "VMEbus Direct Data Transfers".

VMEbus Direct Data Transfers

The VMEbus master can perform direct data transfers, in which a portion (called a *window*) of the VMEbus address space is mapped into the Ivory physical address space, and accessed as though it were (32-bit wide) Ivory memory. For direct transfers, some of the data transfer options, such as the arbitration parameters, are controlled by hardware registers that must be set up prior to the data transfer. Others, such as data shuffling, are controlled by fields within the Ivory physical address decoded by the VMEbus interface. **sys:with-bus-mode** and the address-generating functions **sys:make-bus-address** and **sys:make-bus-array** conspire to keep the hardware parameters consistent with the client's intent.

sys:with-bus-mode establishes a context within which VMEbus addresses may be generated and used; it is illegal to use a VMEbus address returned by **sys:make-bus-address** or **sys:make-bus-array** outside the dynamic scope of the **sys:with-**

bus-mode in which it was created. **sys:with-bus-mode** programs the VMEbus interface according to the specified options, and guarantees that those parameters will be maintained throughout the dynamic extent of the macro, even if some other process is trying to use the VMEbus simultaneously in a completely different manner.

The first time an address is generated (that is, **sys:make-bus-address** or **sys:make-bus-array** is called) within a given **sys:with-bus-mode**, the direct access window in the VMEbus interface is programmed to encompass the specified addresses. A subsequent attempt to generate an address that doesn't lie within the same 267-megaword window will signal an error. If this restriction causes problems, they can often be resolved by using polled transfers to refer to some of the disparate locations.

Note that **sys:bus-read**, **sys:bus-write**, and **sys:bus-store-conditional** are polled transfers and are therefore not affected by **sys:with-bus-mode**; they may be used at any time.

Summary of VMEbus Transfer Options

The following options may be specified to **sys:bus-read**, **sys:bus-write**, **sys:make-bus-address**, **sys:make-bus-array**, and **sys:with-bus-mode**:

:shuffle

One of **:none**, **:byte**, **:nibble**, or **:bit**, this specifies the permutation to be applied to the data words received or transmitted by Ivory. **:bit** shuffling reverses the order of all 32 bits. **:byte** shuffling reverses the order of the four 8-bit bytes in a word, but preserves the order within each byte. **:nibble** does the same for 4-bit groups. The default is **:none**.

:data-type

One of **:fixnum** or **:single-float**, this specifies the tag to be appended to data received by Ivory. **:fixnum** is the default, **:single-float** might be useful when communicating with an array processor or similar device. This is meaningful only for the XL400.

The following options may be specified to **sys:bus-read**, **sys:bus-write**, and **sys:with-bus-mode**:

:address-modifier

The 6-bit numeric VMEbus address modifier code to be driven onto the bus during a data transfer cycle. The default is #x09, indicating that the address is 32 bits wide, for a data cycle.

:ownership

One of **:release-when-done**, **:release-on-request**, or **:bus-hog**, this specifies the condition under which the VMEbus interface will relinquish ownership of the bus once it has control. The default is **:release-on-request**.

:arbitration-priority

An integer from 0 to 3, indicating the priority the VMEbus interface will assert when requesting access to the bus. The default is 3.

The following options may be specified to **sys:bus-read** and **sys:bus-write**:

:byte-size

One of 1, 2, 3, or 4, this specifies the number of bytes of VMEbus data. The VMEbus interface will issue an 8, 16, 24, or 32 bit operation as necessary to perform the transfer.

:byte-offset

One of 0, 1, 2, or 3, this specifies the first significant byte of the VMEbus data.

When using the **:byte-size** and **:byte-offset** options, note that all the specified bytes must be contained within an aligned 32-bit word. That is, the size plus the offset must be greater than zero and less than five.

VMEbus Interrupts

Interrupts may be posted on the VMEbus using the function **sys:post-bus-interrupt**, which issues an interrupt at a specified level, waits for the receiver to acknowledge, then delivers the specified status byte to the interrupt handler. Note that the VMEbus interface cannot deliver an interrupt to itself.

The VMEbus interface will interrupt the Ivory processor upon receipt of any VMEbus interrupt for an enabled level. Which levels are enabled is controlled by a mask in the interface hardware, which may be examined and altered using **sys:logior-bus-interrupt-mask** and **sys:logand-bus-interrupt-mask**. The mask contains a 1 in each bit for which an interrupt is enabled; for example, if the mask were #b00001010, interrupts at levels 1 and 3 would be received, and all others would be ignored. Upon receipt of an interrupt request, the VME software issues an interrupt acknowledge cycle to retrieve the status byte, and calls the appropriate client interrupt handler in a Genera simple process.

You can also use **sys:enable-bus-interrupt** and **sys:disable-bus-interrupt** to enable interrupts. Both functions take a level as an argument.

Client software may supply a handler function for a specific status/ID using **sys:install-bus-interrupt-handler**. An interrupt handler function is a normal Lisp function that takes one argument: the status/id byte received during the interrupt acknowledge cycle. The interrupt level is not disabled when the interrupt is received; the programmer must manage this.

VMEbus Slave Interface

The VMEbus interface for the XL400 and Symbolics UX-family contains a 32K by 32 bit memory that appears on the VMEbus as a slave device. The slave supports A24 and A32 address modes, and D08(EO), D16, and D32 data transfers.

The VMEbus interface for the XL1200 has a 256K by 32 bit buffer that supports the same modes.

The XL slave buffer responds to the following VMEbus address modifiers:

<i>Modifier</i>	<i>Description</i>
#x39	Standard normal data access
#x3A	Standard normal program access
#x3B	Standard normal block transfer
#x3D	Standard supervisor data access
#x3E	Standard supervisor program access
#x3F	Standard supervisor block transfer
#x09	Extended normal data access
#x0A	Extended normal program access
#x0B	Extended normal block transfer
#x0D	Extended supervisor data access
#x0E	Extended supervisor program access
#x0F	Extended supervisor block transfer

The XL400 slave buffer responds to VMEbus address #xFADC0000 (extended) and #xDC0000 (standard). The XL1200 slave buffer responds to VMEbus address #xFAC00000 (extended) and #xC00000 (standard).

The UX-family machine slave buffer responds to the following VMEbus addresses:

UX400S Board Extended Address

0	#xFADC0000
1	#xFAEC0000
2	#xFAF40000
3	#xFAF80000
4	#xFABC0000
5	#xFA9C0000
6	#xFAAC0000
7	#xFAB40000
8	#xFAB80000

UX1200S Board Extended Address

1	#xFD000000
2	#xFD200000
3	#xFD400000
4	#xFD600000
5	#xFD800000
6	#xFDA00000

```

7          #xFDC00000
8          #xFDE00000

```

The slave buffer may be accessed from Ivory using **sys:make-bus-address** or **sys:make-bus-array**, simply by specifying a VMEbus address that falls within the range of the slave buffer. Note that data transfers to such an address don't actually incur any VMEbus traffic; internal data paths are used. The **:shuffle** and **:datatype** options are supported for slave buffer transfers, and work just as they do for normal VMEbus transfers. See the section "Summary of VMEbus Transfer Options".

- The slave buffer address on XL1200 boards can be set via jumpers on the processor board. They are set at the factory to #xFAC00000.
- The mailbox address for each board is at #x100000 beyond the slave-buffer. For example, for UX1200S #1, (+ #xFD000000 #x100000) → #xFD100000
- Lisp keeps track of the location of the slave buffer in the variables **sys:*vme-slave-buffer-base*** and **sys:*vme-slave-buffer-end***. These addresses are in words, so use (**lsh sys:*vme-slave-buffer-base* 2**) to get the VME address of the slave buffer.

If a different VME address is used for the slave buffer, you can inform Lisp of the change by using the keyword **:Slave Buffer Address** to the **Set Boot Options FEP** command.

Note: Sections of slave buffer memory are reserved for use by Symbolics for certain hardware configurations. For more information, see the function **sys:allocate-slave-buffer-memory**.

Resetting the XL-Family and Symbolics UX400S VMEbus

The VMEbus SYSreset signal is asserted on the XL backplane shortly after initial powerup, and whenever the RESET button on the front panel is pressed. The XL processor board responds to SYSreset by initializing the Ivory processor and the I/O board, and cold-booting the FEP. The contents of the XL main memory are preserved, and the FEP software should be able to warm boot Genera if it was running prior to the SYSreset.

The XL400 does not generate or respond to the VMEbus SYSfail signal; the XL1200 does generate SYSfail.

Sun systems also assert SYSreset on powerup. The UX-family machine's processor board responds to SYSreset by initializing the Ivory processor and sending a signal to the machine's life support. When the UX-family machine's life support becomes available, it will cooperate with the UX-family machine's processor board in cold-booting the FEP. The contents of UX-family machine's main memory are preserved, and the FEP software should be able to warm boot Genera if it was running prior to SYSreset.

You can initiate `SYSreset` on an XL1200 by using the function `cli::merlin-ii-sysreset`. This is equivalent to pressing the RESET button on the front panel.

Examples of Using the VMEbus Interface

This section shows several different ways to perform a simple VMEbus data transfer operation in which the goal is to copy a contiguous block of 32-bit words from one VMEbus address to another, reversing the 4 8-bit bytes with each word.

```
;;; Given an A32 D32 slave, use polled transfers to copy each
;;; word. The bytes are shuffled by the interface hardware as
;;; each word is read from the source. Simple but slow.
(defun copy-VME-memory-shuffling (source-bus-address
                                destination-bus-address words)
  (loop repeat words
        for s from source-bus-address
        for d from destination-bus-address
        do
        (sys:bus-write d (sys:bus-read s :shuffle :byte))))
```

```
;;; Given an A32 D16 slave, use polled transfers to copy each
;;; 32-bit word in two halves. The bytes within each 16-bit word
;;; are shuffled by the interface hardware as each word is read
;;; from the source, but we have to manually interchange the two
;;; halves of each 32-bit word.
(defun copy-VME-memory-shuffling (source-bus-address
                                destination-bus-address words)
  (loop repeat words
        for s from source-bus-address
        for d from destination-bus-address
        do
        (let ((v (sys:bus-read s :shuffle :byte :byte-size 2
                            :byte-offset 0)))
          (sys:bus-write d v :byte-size 2 :byte-offset 2))
        (let ((v (sys:bus-read s :shuffle :byte :byte-size 2
                            :byte-offset 2)))
          (sys:bus-write d v :byte-size 2 :byte-offset 0))))
```

```

;;; Given an A16 D8 slave, use polled transfers to copy each
;;; 32-bit word in four separate bytes. We have to do the byte
;;; swapping manually. We have to use the :address-modifier
;;; option to specify short (A16) addresses.
(defun copy-VME-memory-shuffling (source-bus-address
                                destination-bus-address words)
  ;; This with-bus-mode isn't actually required, we could instead
  ;; specify an :address-modifier option to every bus-read and
  ;; bus-write. But the options for those operations take their
  ;; defaults from the ambient with-bus-mode, so this is
  ;; syntactically cleaner.
  (sys:with-bus-mode (:address-modifier #x29)
    (loop repeat words
          for s from source-bus-address
          for d from destination-bus-address
          do
            (let ((v (sys:bus-read s :byte-size 1 :byte-offset 0)))
              (sys:bus-write d v :byte-size 1 :byte-offset 3))
            (let ((v (sys:bus-read s :byte-size 1 :byte-offset 1)))
              (sys:bus-write d v :byte-size 1 :byte-offset 2))
            (let ((v (sys:bus-read s :byte-size 1 :byte-offset 2)))
              (sys:bus-write d v :byte-size 1 :byte-offset 1))
            (let ((v (sys:bus-read s :byte-size 1 :byte-offset 3)))
              (sys:bus-write d v :byte-size 1 :byte-offset 0))))))

```

The remaining examples use direct transfers to perform this same operation, and therefore work for D32 slaves only.

```

;;; Map the VMEbus addresses into Lisp arrays, then use a Common
;;; Lisp sequence operator to do the copying. The bytes are
;;; shuffled by the interface hardware as each word is read from
;;; the source. Simple and reasonably efficient for large
;;; transfers, although the setup overhead is fairly high.

(defun copy-VME-memory-shuffling (source-bus-address
                                destination-bus-address words)
  ;; with-bus-mode must be wrapped around all uses of
  ;; direct-transfer addresses.
  (sys:with-bus-mode ()
    (stack-let ((s (sys:make-bus-array source-bus-address words
                                       :shuffle :byte))
               (d (sys:make-bus-array destination-bus-address words)))
      (replace d s))))

```

```

;;; Map the VMEbus addresses into physical addresses and use
;;; simple memory subprimitives to do the copying. Efficient for
;;; short transfers because of the low setup overhead, but low
;;; level and error prone.
(defun copy-VME-memory-shuffling (source-bus-address
                                destination-bus-address words)
  ;; with-bus-mode must be wrapped around all uses of
  ;; direct-transfer addresses.
  (sys:with-bus-mode ()
    (loop repeat words
      for s first (sys:make-bus-address source-bus-address words
                                       :shuffle :byte)
      then (sys:%pointer-plus s 1)
      for d first (sys:make-bus-address destination-bus-address words)
      then (sys:%pointer-plus d 1)
      do
        (sys:%memory-write d (sys:%memory-read s))))))

;;; Map the VMEbus addresses into physical addresses and use
;;; block memory operations to do the copying. This is the most
;;; efficient way to do bulk transfers.
(defun copy-VME-memory-shuffling (source-bus-address
                                destination-bus-address words)
  ;; with-bus-mode must be wrapped around all uses of
  ;; direct-transfer addresses. Direct transfers will work for
  ;; A24 and A16 slaves, using the :address-modifier option to
  ;; with-bus-modes as follows. If we're really trying to be fast
  ;; and don't mind being nasty, we can do the entire transfer
  ;; without ever relinquishing the bus to another master, using
  ;; the :ownership option.
  (sys:with-bus-mode (:address-modifier #x39 :ownership :bus-hog)
    ;; with-block-registers must be wrapped around all uses of
    ;; block registers.
    (sys:with-block-registers (1 2)
      ;; Use block register 1 to address the source
      (setf (sys:%block-register 1)
            (sys:make-bus-address source-bus-address words
                                 :shuffle :byte))
      ;; Use block register 2 to address the destination
      (setf (sys:%block-register 2)
            (sys:make-bus-address destination-bus-address words))
      ;; Use an unrolled loop to copy the words, which makes the
      ;; memory pipeline operate most efficiently.
      (sys:unroll-block-forms (words 4)
        (sys:%block-write 2 (sys:%block-read 1))))))

```

Dictionary of VMEbus Functions

sys:allocate-slave-buffer-memory *name words &key :from-end* *Function*

Returns a starting and ending address in words. There is no enforcement, but this a simple check-out scheme for slave buffer memory so you do not accidentally use memory allocated by the system (as on the UX-family machine or by another application (like FrameThrower). Specific allocations can be added to an initialization list. System allocations take place on the system initialization list.

sys:bus-error *Flavor*

This condition is signalled if there is a VMEbus error such as a request timeout. Errors are signalled only on read operations; the XL400 processor stores errors that occur on write operations to be signalled by a future read operation.

sys:bus-read *bus-address &rest options* *Function*

Reads the location specified by *bus-address* using a polled transfer. All options default to those specified by the ambient bus mode.

See the section "Summary of VMEbus Transfer Options" for a description of the applicable bus options.

sys:bus-store-conditional *bus-address old new &rest options* *Function*

Checks to see whether the specified bus location contains *old*, and, if so, stores *new* in that location. The test and set are done as a single atomic operation; no other bus operations are allowed between the two. Both the read and the write are performed using the specified bus options, if any, which default to those specified by the ambient bus mode, if any. **sys:bus-store-conditional** returns **t** if the test succeeded and **nil** if the test failed. See the function **store-conditional**.

See the section "Summary of VMEbus Transfer Options" for a description of the applicable bus options.

sys:bus-write *bus-address value &rest options* *Function*

Stores the specified *value* into the location specified by *bus-address* using a polled transfer. All options default to those specified by the ambient bus mode.

See the section "Summary of VMEbus Transfer Options" for a description of the applicable bus options.

sys:deallocate-slave-buffer-memory *name* *Function*

name represents the portion of the slave buffer that was allocated by **sys:allocate-slave-buffer-memory**.

sys:disable-bus-interrupt *level* *Function*

Disables VMEbus interrupts at *level*. *level* can be between 1 and 7.

sys:enable-bus-interrupt *level* *Function*

Enables VMEbus interrupts at *level*. *level* can be between 1 and 7.

sys:install-bus-interrupt-handler *function status-id* *Function*

Installs *function* as the interrupt handler for interrupts within the specified *status-id*. When an interrupt is detected at that interrupt level, and that interrupt level is enabled in the interrupt mask, *function* will be called with one argument, the status/id byte supplied by the interrupter. The handler will be called in a simple process, and therefore must not depend on the dynamic environment (special variable bindings, catch tags, and so on).

Note that if *function* is redefined, the handler must be installed again for the new definition to take effect.

sys:logand-bus-interrupt-mask *mask* *Function*

Atomically reads the VMEbus interrupt enable mask register, logands it with the *mask* argument, and stores the result back in the register. This function is useful for disabling particular interrupts. It returns the new value, so the current state of the interrupt mask can be read as follows:

```
(sys:logand-bus-interrupt-mask -1)
```

sys:logior-bus-interrupt-mask *mask* *Function*

Atomically reads the VMEbus interrupt enable mask register, logiors it with the *mask* argument, and stores the result back in the register. This function is useful for enabling particular interrupts. It returns the new value, so the current state of the interrupt mask can be read as follows:

```
(sys:logior-bus-interrupt-mask 0)
```

sys:make-bus-address *bus-address size &rest options* *Function*

Returns an Ivory physical address usable to access the specified location on the VMEbus. This address is usable within only the ambient **sys:with-bus-mode**. All options default to those specified by the ambient bus mode. An error is signalled if there are any conflicts between the specified options and the hardware configuration specified by the ambient bus mode, or if the desired address range is not supported by the hardware. The first call to **sys:make-bus-address** or **sys:make-bus-array** within a **sys:with-bus-mode** will set up any necessary address window; if a subsequent call specifies an address range outside that window an error will be signalled.

See the section "Summary of VMEbus Transfer Options" for a description of the applicable bus options.

sys:make-bus-array *bus-address dimensions &rest options* *Function*

Returns an indirect array pointing to the specified VMEbus address, using the direct transfer facility. This array is usable only within the ambient **sys:with-bus-mode**. The options include all the **make-array** options, but note that the array cannot contain arbitrary Lisp objects, only integers and single-precision floating point numbers; see the section "Keyword Options for **make-array**". The options may also include any applicable bus options, which default to those specified by the ambient bus mode. An error is signalled if there are any conflicts between the specified options and the hardware configuration specified by the ambient bus mode, or if the desired address range is not supported by the hardware. The first call to **sys:make-bus-address** or **sys:make-bus-array** within a **sys:with-bus-mode** will set up any necessary address window; if a subsequent call specifies an address range outside that window an error will be signalled.

See the section "Summary of VMEbus Transfer Options" for a description of the applicable bus options.

sys:post-bus-interrupt *&optional (level 0) (status 0)* *Function*

Issues an interrupt request for the specified level on the bus, waits for the interrupt acknowledge cycle, then transmits the specified status/id byte to the interrupt handler.

sys:*vme-slave-buffer-base* *Variable*

The starting address for the slave buffer, in words.

sys:*vme-slave-buffer-end* *Variable*

The ending address for the slave buffer, in words.

sys:with-bus-mode *(&rest options) &body body* *Macro*

Establishes a context within which VMEbus addresses may be generated and used; it is illegal to use a VMEbus address returned by **sys:make-bus-address** or **sys:make-bus-array** outside the dynamic scope of the **sys:with-bus-mode** in which it was created. **sys:with-bus-mode** programs the VMEbus interface according to the specified options, and guarantees that those parameters will be maintained throughout the dynamic extent of the macro, even if some other process is trying to use the VMEbus simultaneously in a completely different manner.

See the section "Summary of VMEbus Transfer Options" for a description of the applicable bus options.