Upgrading to Genera 8.0

Preface

Upgrading to Genera 8.0 presents an overview of porting applications running in Genera 7.2 or Genera 7.4 Ivory to Genera 8.0. Information here will be useful to you if you are:

- Porting applications running in Genera 7.2 on 3600-series machines to Genera 8.0.
- Porting applications running in Genera 7.4 Ivory on Ivory-based machines to Genera 8.0.
- Porting applications running on the 3600 series to the Ivory series.

In addition, information is included on converting Lisp code from Zetalisp to Symbolics Common Lisp, and on converting from Flavors to CLOS. Note that flavors are fully supported in Genera 8.0; conversion tools are provided primarily for reasons of portability.

Porting Your Application to Genera 8.0

Genera 8.0 is the first Genera release that runs uniformly across all Symbolics machines: the 3600 series and the XL400 , Symbolics UX-family, and MacIvory.

Major features of this release include:

- Source compatibility. The same source code is supported for all types of Symbolics machines in Genera 8.0. In most cases, code that ran in Genera 7.2 or Genera 7.4 Ivory can be run in Genera 8.0 after recompilation.
- Portability. Genera 8.0 supports the Common Lisp Object Standard (CLOS) for portability of Lisp programs that use this standard. (A tool for converting code that uses Flavors to CLOS is provided. Such a conversion may be useful for porting programs to other Common Lisp systems. However, it is important to note that such a conversion is not required for running code in Genera 8.0.)
- Networking and I/O. Many areas of the system provide more robust and faster performance. Genera 8.0 includes support for the Sun Remote Procedure Call (RPC) protocol. In addition, handling of character I/O streams, support of Postscript hardcopy, Symbolics Network File System (NFS) and the X Window System client function are improved. (Symbolics NFS and Symbolics X Window System are Genera layered products).
- New Scheduler. Users of the 3600 series can now use a new Scheduler facility, first released as part of the Ivory-only Genera 7.3 release.

- Many bug fixes and improvements. The compiler produces faster code, and the Common Lisp Developer is part of Genera 8.0.

The *Genera 8.0 Release Notes* describe changes made since Genera 7.4 Ivory. In addition, many chapters of the release notes contain a section, "Changes to ... Since Genera 7.2", for users of 3600-family machines migrating from Genera 7.2. These sections list changes made to Genera in Genera 7.3 Ivory and Genera 7.4 Ivory.

One important addition in 8.0 is Symbolics CLOS, our implementation of the Common Lisp Object System. For more information, see the section "CLOS is Available in Genera 8.0". For information on converting from Flavors to CLOS, see the section "Flavors to CLOS Conversion".

Code Recompilation

Genera 8.0 provides source code compatibility with Genera 7.2 for the 3600 series and Genera 7.4 Ivory for the Ivory series. Most users will need only to recompile their programs to run them under Genera 8.0.

For Genera 7.2 users, recompilation is required if you are porting programs from Genera 7.2 on the 3600 series. Some code may seem to run properly without recompilation, but running such code in Genera 8.0 will introduce subtle bugs and unexpected behavior.

For Genera 7.4 Ivory users, if you are porting code from Genera 7.4 Ivory on a Symbolics UX-family, XL400, or MacIvory, we recommend that you recompile your code to take advantage of bug fixes and performance improvements, but it is not a requirement.

Applications using Symbolics layered products will also benefit from recompilation.

For both layered-language and Symbolics Common Lisp applications, object code produced for a 3600-series machine (in binary files or world loads) may not be loaded into an Ivory-based machine, or vice versa. In such cases, recompilation is a requirement.

The following table lists common situations of porting programs to Genera 8.0 and summarizes what you need to do in each case:

Situation	*Actions*
3600 series to 3600 series (Genera 7.2 to Genera 8.0)	Recompile your code in Genera 8.0 on a 3600-series machine.
Ivory series to Ivory series (Genera 7.4 Ivory to Genera 8.0)	Recompile your code in Genera 8.0 on an Ivory-based machine.
Layered products applications* (Genera 7.2 or 7.4 Ivory to Genera 8.0)	Recompile your code in Genera 8.0 following instructions for the machine types you are using.
3600 series to Ivory series (Genera 8.0 to Genera 8.0)	Recompile your code in Genera 8.0 on a 3600-series machine.

See the section "Porting Genera Applications to Ivory Machines".

	Recompile your code in Genera 8.0 on an Ivory-based machine.

*Including Symbolics C, FORTRAN, Pascal, Joshua, Statice, and Symbolics Concordia.

Using Ivory-based Machines

If you are porting applications from the 3600 series to an Ivory-based machine, first convert your code to Genera 8.0 on a 3600-series machine.

If you are running an existing application on an Ivory-based machine (MacIvory, XL400, or Symbolics UX-family) for the first time, see the section "Porting Genera Applications to Ivory Machines". This tells you how to convert applications that run on Genera 7.2 on the 3600 series to Genera 8.0 on Ivory-based machines (the XL400, MacIvory, and the Symbolics UX-family), as well as presenting some general issues you need to be aware of when porting such code. Additional sections describe issues in maintaining parallel Ivory and 3600-series systems and present some variables and macros you may find useful in converting your programs.

Incompatible Changes to Undocumented Interfaces

Programs being moved from Genera 7.2 on the 3600 series to Genera 8.0 on the 3600 series will encounter some incompatible changes to undocumented facilities, mainly in the window system. Some points to be aware of:

- The use of undocumented names in the **system**, **system-internals**, and **common-lisp-internals** packages is suspect.
- The polling scheduler has been replaced with an event-driven version.
- The window system has been changed, in particular the locking strategy, to allow access to the display, mouse, and keyboard hardware of MacIvory to go through the Macintosh host. The window system no longer uses **without-interrupts**.

The vast majority of the Genera system software remains essentially unchanged, particularly higher levels of the system such as Dynamic Windows and Zwei. Many programs that use undocumented interfaces will continue to work, for the time being. See the document *Genera 8.0 Release Notes*, for further information on changes to undocumented interfaces.

Porting Genera Applications to Ivory Machines

This section presents an overview of the work involved in porting applications running on the 3600 series of Symbolics machines to machines based on the Ivory processor (XL400, Symbolics UX-family, and MacIvory).

The first several sections discuss general issues you need to be aware of when porting code to Ivory-based systems. This information is primarily useful in determining the scope and cost of a porting effort. (In general, this effort should be a fairly simple one, involving little more than recompiling your code.) Additional sections describe issues in maintaining parallel Ivory and 3600-series systems and present some variables and macros you may find useful in porting your programs.

See the following sections:

"General Considerations in Porting Programs to Ivory-based Machines"
"Software Management Differences for Ivory-based Machines"
"Changes to Undocumented Interfaces for Genera on Ivory-based Machines"
"Changes to Subprimitives and System Constants for Genera on Ivory-based Machines"
"Changes to Genera I/O for Ivory-based Machines"
"Compiling a System for Multiple Machine Types"
"Variables and Macros for Converting to Ivory"

MacIvory Users

If you are porting your application to MacIvory, portions of the following documents may be of interest:

- *MacIvory User's Guide*
- *Genera 8.0 Release Notes*

XL400 Users

If you are porting your application to an XL400, portions of these documents may be of interest:

- *XL User's Guide*
- *Genera 8.0 Release Notes*

Symbolics UX-family Users

If you are porting your application to a Symbolics UX-family machine, portions of these documents may be of interest:

- *User's Guide to the Symbolics UX*
- *Genera 8.0 Release Notes*

We recommend you read through these books before converting any programs.

General Considerations

A key aspect of Genera's design philosophy is that different architectural levels are carefully separated, and may be changed independently. The instruction set architecture may be changed, as it was when the 3600 series was introduced, with-

out affecting the overall system architecture. The processor architecture may be freely changed, as it was in the 3675, 3620, and 3650, with no effect on the software. Or, the overall software system architecture may be changed, as it was in Genera 7.0, independent of the underlying instruction set and processor architecture.

Ivory-based systems use a different instruction set and processor architecture than the 3600 series, but maintain the Genera system architecture and provide full source-code compatibility for Genera applications. The new architecture enables a single-chip processor implementation, with accompanying benefits in system cost, performance, reliability, and the opportunity to configure systems that were previously infeasible.

Source code compatibility means that programs using Common Lisp, Symbolics Common Lisp extensions, or the Genera substrate in the documented manner need only be recompiled in order to run on the new machines. Symbolics' layered languages, and the code they produce, are compatible to this same degree. Object code produced for a 3600-series machine (in binary files or world loads) may not be loaded into an Ivory-based machine, or vice versa.

Therefore, the porting task primarily consists of recompiling software, and maintaining and distributing consistent object code for both families of machines. Occasionally, an application will require source changes to make it portable to the Ivory-based series; this can result from the use of undocumented internal functions of the system, or more rarely from the use of low-level "subprimitive" operations, as discussed in the section "Changes to Subprimitives and System Constants for Genera on Ivory-based Machines".

Once an application is ported to Genera Ivory and the Ivory processor, you might want to change its user interface, for instance to fit into the Macintosh user interface in a MacIvory system. This is best done after the basic porting is done. You might also want to tune the application's performance (see the section "Metering Interface").

In addition, you may want to use some features of the new scheduler. For further information, see the section "Converting to the New Scheduler".

Software Management Differences

Binary File Formats for Ivory-based Systems

Since the Ivory processor implements a different instruction set from the 3600 series, all programs must be recompiled to run on it. Compiling a file on an Ivory-based system produces a binary file in a file format identified by the file type `ibin`. This binary file format is distinct from the type of binary file produced by compiling the same file on a 3600-series machine (which is identified by the file type `.bin`). The System Construction Tool (SCT) helps manage this, ensuring that the correct binary files for a given software system are loaded and compiled, depending on the target processor (see the section "Compiling a System for Multiple Machine Types").

Data Files

The differences between the 3600-series `bin` file format and the Ivory `ibin` file format are limited to the representation of compiled code; data stored in binary files is compatible between the two processors. So binary files containing only data and no compiled code, such as those produced by **sys:dump-forms-to-file**, can be used freely on both processors.

World-load Files

3600-series and Ivory-based machines require different world loads. The `.ilod` file extension indicates world-load files for Ivory-based machines, and the `.load` file extension indicates world-load files for the 3600-series. Files with the `.ilod` extension can be copied only between Symbolics Ivory-based machines. Files with the `.load` extension can be copied only between Symbolics 3600-series machines. For further information, see the section "Copy World Command".

Read-time Conditionals

Many incompatibilities in different vendors' software are handled using read-time conditionals. If you currently distinguish Symbolics software with **#+3600**, change your sources to say **#+Symbolics**. Use **#+3600** to distinguish any code that is different on the 3600 series and Ivory. If you want to distinguish Symbolics Genera from Symbolics CLOE, change your sources from **#+3600** to **#+Genera.** In addition, the following read-time conditionals are valid: **#-Symbolics**, **#+IMach**, **#-3600**, and **#-IMach**.

Changes to Undocumented Interfaces

Adapting Genera to run on Ivory-based systems required changing portions of the underlying system software to various degrees. Although this software is undocumented, the source code is either included with Symbolics' standard software license or is available for an additional fee. If you have application software that uses undocumented features gleaned from the 3600-series system source code, be aware that such code may not work on Ivory-based systems without modification.

The following is a summary of the major areas of change. Machine dependencies are most likely to be in one of these areas, although any use of undocumented names in the **system**, **system-internals**, or **common-lisp-internals** packages is suspect.

- Internal system functions have been rewritten to accommodate Ivory's differing object representations, including arrays, compiled functions, various types of numbers, logic variables, and stack groups.

- Internal functions with knowledge of the control stack format, including the implementation of **throw** and similar functions, stack-group manipulation, and the Debugger, have been rewritten to accommodate the Ivory stack format.

- Primitive graphics operations that were implemented in 3600-series microcode have been rewritten in (highly tuned) Lisp.

- The garbage collector has been extensively modified to accommodate the streamlined GC support hardware used in Ivory.

- The virtual memory system has been modified to accommodate the memory mapping techniques used by Ivory, and to support more efficiently the larger virtual and physical address space provided by Ivory.

- The polling scheduler has been replaced with an event-driven version.

- Internal functions of the scheduler and the I/O system have been rewritten to accommodate Ivory's improved interrupt architecture.

- The window system has been changed, in particular the locking strategy, to allow access to the display, mouse, and keyboard hardware of MacIvory to go through the Macintosh host. The window system no longer uses **without-interrupts**.

Note that most of these areas of the system also received significant improvements in performance or functionality during the porting process. Some of these improvements are transparent, and others will require some (optional) source changes to gain their full benefit.

The vast majority of the Genera system software remains essentially unchanged, particularly higher levels of the system such as Dynamic Windows and Zwei. Many programs that use undocumented interfaces will continue to work, for the time being.

Changes to Subprimitives and System Constants

All subprimitives documented in the the Genera Documentation Set work exactly the same on Ivory as on the 3600 series. However, since some object representations have been changed (for example, the internal structure of arrays has been streamlined) and some fundamental implementation parameters are different (for example, the virtual address space size has been increased), code that uses these subprimitives may not be machine independent.

There have been a few changes to subprimitive constants documented in *Internals*. The constants, and their equivalents on Ivory, are listed in the following table. All symbols are in the **system** package unless otherwise noted.

3600 Series	*Ivory Equivalents*
dtp-fix *(16 values)*	**dtp-fixnum**
dtp-float *(16 values)*	**dtp-single-float**
dtp-closure	**dtp-dynamic-closure**
dtp-extended-number	**dtp-small-ratio**
	dtp-double-float
	dtp-bignum
	dtp-big-ratio
	dtp-complex
	dtp-spare-number
dtp-list	**dtp-list**
	dtp-list-instance
dtp-array	**dtp-array**
	dtp-array-instance
	dtp-string
	dtp-string-instance
stack-area	**stack-area**
	si:control-stack-area
	si:structure-stack-area
	si:binding-stack-area

Additionally, the values of many documented system constants have changed. These changes do not affect code that uses these constants by name; only code that depends on their values is affected. Note the following:

- When using **sys:%start-function-call**, you must finish with a **sys:%finish-function-call**; it is illegal to abort out of the started call.

- The textual output of various system-describing functions will change somewhat, reflecting a different implementation. Examples of affected functions are **disassemble**, **describe-area**, **zl:gc-status,** and **room**.

- **sys:%unsynchronized-device-read** is no longer valid.

- In contrast to the 3600 series' 28-bit address space and 36-bit word size, Ivory provides a 32-bit address space and a 40-bit word. Additionally, the address space in Ivory is partitioned differently from that on the 3600 series; numerical addresses should not be built into programs. A minor software implication is that addresses represented as two's-complement fixnums are negative for half of the address space. Software that converts object locations to fixnum addresses and then assumes that they will be positive will not work properly. Numeric area numbers should not be used, either.

- The larger word size results in a larger page size (in terms of bits per page). Therefore, the sector size for disks on Ivory-based systems can be expected to differ from the sector size for disks on the 3600 series. On Ivory-based systems, disks use a page size of 1280 bytes per page. I/O software that depends on the

3600-series page size (1152 bytes per page) may have to be modified. This affects programs that call such interfaces as **sys:disk-read** or that use the disk interface documented in *Internals*.

Changes to Genera I/O

Ivory is used in a number of different system configurations, all of which have I/O hardware that differs from that used in the 3600 series. Some of these configurations are stand-alone workstations that provide the same sort of I/O environment provided with 3600-series systems: dedicated disks formatted for optimal performance, a high-resolution, directly addressable console display, a Symbolics keyboard, and a three-button mouse. In such systems, the internal workings of the I/O-related system software may be substantially different (due to differences in I/O hardware and Ivory's interrupt architecture), but the documented interfaces provided to application programs remain the same.

One exception to this is that the 3600-series facilities for audio output (with the exception of **beep**) are not supported. Also, note that Ivory machines do not use the LBUS used in the 3600 series, and any code that depends on a specific LBUS device, such as the UNIBUS interface, is not portable to Ivory-based systems.

Ivory is also used in *embedded systems*, configurations in which an Ivory processor is embedded in another computer, and relies on that computer to provide some or all of its I/O services. In these systems, the documented interfaces perform the same basic functions, but there may be qualitative differences in the services provided. For example, the screen may be smaller, or hold less information, the keyboard may not have all the same keys, or the mouse may not have three buttons. The exact limitations of each system configuration can be expected to vary.

To isolate application programs from these system differences, Symbolics recommends using the highest levels of abstraction possible. User interfaces built on Dynamic Windows, for example, readily adapt to differing system configurations with little or no effort, and may in fact gain interesting new functionality on some machines. Programs that directly use I/O devices such as disks or serial lines should use the devices through the documented stream interfaces. Network access has been highly abstracted for some time now, and is readily portable across system configurations.

Compiling a System for Multiple Machine Types

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

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

You can use **:version** *N* to compile systems for different machine types in this way: First, compile the system on one machine type. From the second type of ma-

chine, recompile the system specifying the version number of the system you have just compiled. Specifying the **:version** option makes SCT compile the same version of each source file that was compiled when the system was compiled before, instead of compiling the newest version of each source file. It causes SCT to look at the journals and compile the same version of the system from the same sources for the new machine type.

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

```
Compile System Sample
```

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

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

or

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

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

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

Maintaining Parallel Systems for Multiple Machine Types

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

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

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

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

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

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

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

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

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

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

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

Variables and Macros for Converting to Ivory

si:*default-binary-file-type* *Variable*

A variable that takes the canonical type of compiled files as its value: :BIN for the 3600-series and :IBIN for Ivory-based machines.

sys:system-case *(keylist form...)...* *Macro*

Selectively executes forms depending on what system it is executing on. This macro exists only on Ivory-based machines, not the 3600 series, and is used only in machine-dependent code. It works in both wired and pageable code. Since all Ivory-based systems are binary compatible, for both .ibin files and world loads (.ilod files), run-time dispatching with **sys:system-case** is used when code must vary for different Ivory-based systems.

Each *keylist* can be a system-type name, a list of system-type names, the symbol **otherwise** or the symbol **t**. System-type names are compared without regard for packages. If no **otherwise** clause is present, a default one that signals an error is provided (this is like **ecase**).

The currently recognized system-types are:

System-Type	*System*
Native	A standalone Symbolics computer. Examples are XL400, XL1200, and NXP1000 systems.
MacIvory	A Symbolics processor embedded in a Macintosh modular computer. Examples are MacIvory, 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 embeddings.
Domino	A Symbolics processor with a proprietary I/O architecture. Domino systems normally do not have a hardware console. An example is the NXP1000.

A clause of the form **(***keylist* **(error))** signals an error for any of the systems identified by *keylist*. Clauses of this form may be used only if no **otherwise** clause is specified.

The clause **(otherwise (error))** is 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 clause of the form **(never** *keylist***)** is equivalent to the clause **(***keylist* **(error)).** Clauses of this form may be used only if no otherwise clause is specified.

graphics:with-scan-conversion-mode *(stream* &key *:round-coordinates :center-circles :host-allowed :sketch)* &body *body* *Special Form*

Binds the local environment so that the figures produced within it are drawn using the specified scan conversion mode. Use this form in place of **graphics:with-coordinate-mode**. Note that this scheme permits the future inclusion of other controls, such as anti-aliasing.

stream The output stream.

:round-coordinates The coordinates specified to the drawing function are rounded to integer values and special, faster integer drawing methods are used. Use this mode when speed is important in drawing a filled figure or one with thick lines and exactness is of little importance. The default is yes.

:center-circles Figures are drawn so that they are centered around a whole pixel. The center of a circle is offset by half a device unit. For example, a circle with specified radius *r* and center <*x*, *y*> would be drawn with actual center <*x*+1,*y*+1> and radius *r*+1/2. Use this mode when you want small circles that need not align with other shapes to appear symmetrical around a single pixel. This option corresponds to **graphics:with-coordinate-mode :center**. The default is no.

:host-allowed Permits the use of the embedding host's drawing routines. For example, on the MacIvory, QuickDraw graphics are used. Use of this option overrides the effect of most other options to this function. On embedded systems, like the MacIvory, this can improve the performance of graphics, but slightly alter the shape of the graphic. The default is yes.

:sketch Equivalent to **:round-coordinates** and **:host-allowed**. (with a value of **t** or **nil** for both). Use this to declare unspecifically that speed is preferred over accuracy. The default is **t**.

See the section "Scan Conversion" for related information.

Maximum Stack Levels for Ivory-based Machines

Maximum stack level values differ for Ivory-based machines and 3600-series machines as shown in this table:

Form	*3600-series Value*	*Ivory-series Value*
call-arguments-limit	128	50
lambda-parameters-limit	128	50
multiple-value-limit	128	50

For further information:

See the constant **call-arguments-limit**.
See the constant **lambda-parameters-limit**.
See the constant **multiple-values-limit**.

Converting to the New Scheduler

New Scheduler facilities became available to users of Ivory-based machines in Genera 7.3 Ivory. Genera 8.0 makes this facility available for 3600-series machines as well. For a description of new Scheduler facilities, see the section "Overview of the Scheduler".

Converting Code for the New Scheduler

No conversion is required for most user code. Several relatively minor incompatibilities that might require you to modify your code are documented in the section "Scheduler Incompatibilities That Might Require Code Changes".

There can be some performance differences when you run "old scheduler" code in the new scheduler. Even though most code does not *need* to be changed, you may want to take advantage of some features of the new scheduler.

In the new scheduler, processes that wait for wait-functions are at a disadvantage. Processes that *wakeup* can run immediately; processes that *wait* have to wait for the wait-function poller to poll the wait-functions again, which increases delay and increases system overhead. In general, it pays to convert your code to use block and wakeup, rather than polling.

For more information about the differences between blocking and waiting (blocking with polling), see the section "Blocking Vs. Waiting".

For commands to control the rate and style of polling, see the section "Blocking, Waiting, and Waking Processes".

Scheduler Incompatibilities That Might Require Code Changes

Tracking Active Processes

The internal data structure **sys:active-processes** is no longer used, and is always **nil**. It is included in the world so that old code does not break, but it is not useful.

See the function **process:map-over-active-processes**. See the function **process:map-over-all-processes**.

The Scheduler Stack Group

There is no longer a scheduler stack group, so **sys:scheduler-stack-group** has no meaning. It is included in the world, for compatibility with existing code. It should not be used in new code. Its value is **nil**.

If a piece of code referenced this variable in order to determine if it was safe to call **process-wait**, it should call **process:safe-to-process-wait-p** instead.

Clock Functions

Clock functions are considered obsolete, although they are supported for compatibility reasons. They should be replaced by timers or periodic actions.

The old scheduler attempted to run each clock function at most 60 times per second. Typically, they ran 10 times per second.

In the new scheduler, a clock function runs every **process:*clock-queue-interval*** by default. The priority of the clock-function queue is high, so the clock-functions run this frequently regardless of how busy the machine is. **process:*clock-queue-interval*** is 1/6 seconds by default.

This is slightly less frequently than in the old scheduler on a machine with compute-bound processes. If you decrease the interval in order to increase the frequency, the clock functions will preempt running processes, increasing the system overhead.

If a clock function returns a value, the clock-function queue interprets that number as the number of sixtieths of a second in which this clock function wants to run again. You can use this technique to vary the rate of clock-function polling.

Process Stack Groups

In the new scheduler, **(process-stack-group** *process***)** can return different types of objects than in the old scheduler. In the old scheduler, **process-stack-group** of a simple process would return the initial function, and **process-stack-group** of any other type of process would return the stack group.

In the new scheduler, simple processes temporarily acquire stack groups when they are runnable. Additionally, non-simple processes get rid of their stack group when they are not needed (typically, after a call to **process:reset-and-release-resources**, or **process:kill**).

So, in the new scheduler **process-stack-group** can return either **nil** or a stack group.

Warm Boot Actions

In the new scheduler at warm boot time, the scheduler initialization goes over every process, not just active processes, and executes the warm-boot-action. If a process is arrested, the warm-boot-action might not actually take effect until the process is unarrested, later, (this is true of RESETting an arrested process), but all processes will have their warm-boot-actions run on them at warm-boot time.

The old scheduler ran only the warm-boot-actions of unarrested processes.

Scheduler Incompatibilities That Do Not Require Code Changes

current-process is Not nil Inside the Scheduler

In the old scheduler the value of ***current-process*** was the process that is currently executing, or **nil** while the scheduler is running. Since the scheduler just runs in your process it is, effectively, never running. ***current-process*** is never **nil** during the normal course of operations except for a short period during initialization during booting.

Quantum

The process quantum is not used by the new scheduler, with one exception. The **:quantum-boost** keyword argument to **si:with-process-non-interactive-priority** *is* used. It is interpreted as follows: the quantum boost is assumed to be a timer interval, measured in 60ths of a second (for compatibility). If the non-interactive-priority is nested within a **si:with-process-interactive-priority**, the quantum-boost specifies how long the priority should remain interactive, before actually reducing the priority to the non-interactive level.

Priority

All old priorities are interpreted to be **:foreground** priorities in the new scheduler. Any entry points in the scheduler compatibility package, or any functions that existed in previous releases (for example **make-process** or **process-run-function**), take old scheduler priorities as their arguments, and return old scheduler priorities as their values.

To explicitly convert an old scheduler priority to a new priority, you can use (**process:make-process-priority :foreground** old-pri).

The compatibility function **process::process-priority-to-old-priority** takes a scheduler priority and converts it to an integer representing an old scheduler priority. **:foreground** priorities are converted to their priority level. **:background** priorities

are arbitrarily converted to -256, and **:deadline** priorities are arbitrarily converted to 110.

Non-integer priorities are no longer supported. The **floor** of that priority is used instead.

Sequence Breaks and Break Intervals

si:*default-sequence-break-interval* and **si:sequence-break-interval** are not operational in the new scheduler. A preemption will occur whenever the highest priority runnable process is not the ***current-process***, and preemption is enabled.

If a high priority process is awakened by something that happens during a sequence break, and preemption is enabled, the current process is preempted. **si:sequence-break-interval** will not delay it. Nor will **si:sequence-break-interval** cause periodic preemptions. In the new scheduler there is no reason to arbitrarily preempt the current process. In the old scheduler periodic preemption was necessary to evaluate wait-functions, force the current process to yield the processor, and so on.

In the new scheduler each of these functions is performed by different agents. Wait functions are either evaluated periodically by the wait-function poller, or at wakeup time. The minimum polling rate is set by the caller of **block-and-poll-wait-function**, and by the variable **process:*process-wait-interval***. The current process is periodically preempted by the priority manager. This allows multiprocessing even when the current process does not voluntarily yield the processor. The interval that the scheduler waits between these interrupts is controlled by the **:wakeup** keyword to **process:set-scheduler-parameters**. Finally, many activities are driven by timers. The parameter **process:*timer-resolution*** controls the resolution at which timers go off. The timer facility is free to delay a timer execution for ***timer-resolution*** timer units. Increasing this makes timers less responsive, but reduces the system overhead.

Using the Conversion Tools with Genera 8.0

About the Conversion Tools

The Conversion Tools are a series of special-purpose Zmacs commands that save you time and effort in editing large pieces of code in ways that can be done semi-automatically. This is particularly useful when converting from one Lisp dialect to another.

The available Conversion Tools are:

- Flavors to CLOS, for converting object-oriented programs from using Flavors to using the Common Lisp Object System (CLOS). See the section "Flavors to CLOS Conversion".

- Zetalisp to Common Lisp, for converting Zetalisp programs to Symbolics Common Lisp. See the section "Zetalisp to Common Lisp Conversion".

- Symbolics Common Lisp to portable Common Lisp, for assistance in converting Symbolics Common Lisp programs to more portable programs that will run in a variety of Common Lisp implementations, including Genera and Cloe as well as other vendors' Common Lisps. See the section "Symbolics Common Lisp to Portable Common Lisp Conversion".

- Package Conversion, for moving a program to a different package. See the section "Package Conversion".

- Tools you write yourself, to do any source-to-source conversions you may require. See the section "Creating Your Own Conversion Set".

Though the Conversion Tools are simple, the conversion task is not. For this reason you must use the Conversion Tools with knowledge and care. For example, when converting Zetalisp to Common Lisp, you should be sufficiently comfortable with both Zetalisp and Common Lisp to be aware of function equivalences and to understand the possible effects of a conversion (such as a changed order of argument evaluation for functions whose calling sequence is different in Common Lisp).

It is also very desirable that you be familiar with the details and the intent of the code you are converting. As a trivial example, in order to get the correct conversion of a Zetalisp division operation, you must know what numeric data types the operation is intended for, whether or not truncated division is needed, and so on; this knowledge will let you select quickly among the three Common Lisp alternatives offered you by the Conversion Tools.

Obviously, you also need familiarity with the basic workings of the Zmacs editor.

Bear in mind that the conversion commands are intended as an *aid* to conversion, not as a fully automatic conversion tool. Used properly, they will save you time and effort, but you must monitor the results carefully after each step and be aware that you might have to do some manual work after conversion: for instance, comments might not end up exactly in the right place in the rearranged program, indentation might change, converted functions might need some additions to the code, and so on.

Loading the Conversion Tools

The Conversion Tools reside on the source tape you received with Genera. Once you have restored the distribution tape as described in your *Software Installation Guide*, type this from the Lisp Listener:

```
Load System Conversion-Tools
```

When the system files are loaded, you are ready to use the Conversion Tools.

For your convenience, the distribution tape also includes two sample programs you can use to try out the Conversion Tools: SYS:CONVERSION-TOOLS;CONVERSION-TEST-

PROGRAM.LISP (the main example), and SYS:CONVERSION-TOOLS;CONVERSION-OCTAL-TEST-PROGRAM.LISP (for radix conversion).

If you need further information, see the section "Getting Started with Conversion".

Converting From Flavors to CLOS

When to Convert From Flavors to CLOS

CLOS is the Common Lisp Object System, an object-oriented programming language similar to Flavors that is part of the Common Lisp standard being produced by X3J13, The ANSI Common Lisp working group.

You should convert from Flavors to CLOS if you want to to port your application to other Common Lisp implementations. **You do not need to convert code to CLOS to run your programs in Genera 8.0;** Flavors are fully supported in this release. Convert existing programs using conversion tools provided by Symbolics.

Note that you can begin using CLOS by developing new programs using Common Lisp Developer. The Common Lisp Developer adds a new dimension to the Genera development environment, by enabling you to define certain programs in an environment that replaces Genera's fault-tolerant interpretation of Common Lisp with a stricter interpretation, which is more predictive of other Common Lisp implementations.

See the section "Developing Portable Common Lisp Programs".

Complete integration between Flavors and CLOS is not currently provided. This means that some Flavors programs cannot be converted to CLOS at this time. For example, a Flavors program that makes use of Flavors defined by Genera (such as window or stream flavors) cannot be converted to CLOS until the integration between CLOS and Flavors is completely supported.

Flavors to CLOS Conversion

Commands:

```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

To translate a Flavors program to CLOS, issue one of the above commands. When prompted "Conversion to use", select Flavors to CLOS. This semiautomatic conversion will do most of the work required to convert a program from Flavors to CLOS. Additional manual effort will be required if the program uses Flavors features that do not have any direct counterpart in CLOS.

Each form in the program that can be converted to CLOS results in an interactive query showing the old form in context in the editor buffer and one or more suggested replacement functions. You can then enter one of a variety of single-character commands. Press HELP for a list of options. For further information, see the section "Getting Help with Conversion".

This is not a conversion from Symbolics Common Lisp to portable Common Lisp. The Flavors to CLOS conversion converts only Flavors functions and macros; it does not touch the rest of the program.

If your Flavors program is in Zetalisp, it is recommended, but not required, that you first convert it to Common Lisp before converting it to CLOS. See the section "Zetalisp to Common Lisp Conversion".

The general idea is to convert a flavor to a class with the same name, to convert messages to generic functions, and to convert Flavors generic function and method definitions to CLOS generic function and method definitions with the same generic function name. Flavors names and syntax are replaced with CLOS names and syntax wherever a correspondence exists. Flavors features that do not exist in CLOS are left in the program for you to convert by hand.

CLOS uses symbols that are not accessible in packages such as **cl-user** and **scl**. Unless you first move your Flavors program into a package that uses **clos** or **future-common-lisp**, the CLOS symbols will be inserted into your program with package prefixes. Thus **defmethod** will be converted to **clos:defmethod**. If you later move your program to a package or an implementation where CLOS is directly accessible, you can use m-X Query Replace to remove the package prefixes. For information about moving your Flavors program to another package, see the section "Package Conversion".

The conversion tool extracts information about your program, such as what method-combination type a generic function uses or what instance variables and **defun-in-flavor** functions are defined for a flavor, by looking in three places:

1. Forms that have already been converted during the same conversion operation.

2. Forms that have been read into editor buffers.

3. Flavor and generic function definitions that have been loaded into the world.

Consequently, you will get better results if the entire program is read into the editor or loaded into the world before converting any of it.

The new version of the program can't be used at the same time as the old version, because it uses the same names with different meanings (a class is different from a flavor, and a CLOS generic function is different from a Flavors generic function). However, if you move the new version into a new package before converting it you can avoid this problem. For information about moving your Flavors program to another package, see the section "Package Conversion".

CLOS does not have any equivalent of Flavors' functions whose scope is limited to methods for a particular flavor. Therefore **defun-in-flavor** is converted to **clos:defmethod** (with the addition of an argument, since **self** is no longer conveyed automatically) and **defmacro-in-flavor** is converted to ordinary **defmacro**. These conversions can result in name conflicts, if the same name had been used in the scope of two different flavors. These conflicts will probably show up as method lambda-list congruency errors and must be resolved manually.

Options for **defflavor** or **defgeneric** that do not have counterparts in CLOS are converted into an **:unconverted-flavor-options** or **:unconverted-defgeneric-options** option to remind you to convert these options manually. CLOS does not accept **:unconverted-flavor-options** or **:unconverted-defgeneric-options**, so if you compile the program without completing the manual conversion, an error will be reported.

Some **defflavor** options, such as **:required-flavors** or **:abstract-flavor**, can simply be deleted without harming the operation of a working program. Other options, like **:mixture** or **:special-instance-variables**, have no simple conversion to CLOS. If you use them you might have to restructure your program. The following options have fairly straightforward conversions that are a little too complex to be done automatically:

:constructor	Use **defun** to define a function that calls **clos:make-instance**.
:default-handler	Use **clos:defmethod** to define a default method for each generic function that needs default handling.
:init-keywords	Use **&key** in a **clos:initialize-instance** method.

Symbolics CLOS has an extension to allow slots to be located with **locf**. If you use **:locatable-instance-variables** in Flavors, it is converted to the Symbolics CLOS **:locator** slot-option. This will not work in other CLOS implementations.

Some Flavors efficiency features with no counterpart in CLOS are simply discarded. For example, **defwhopper-subst** is simply converted into an **:around** method, and **defsubst-in-flavor** is treated the same as **defun-in-flavor**.

Messages are converted to generic functions. This applies both to receivers (**defmethod**, **:gettable-instance-variables**, and so on.) and to senders (**send**, **lexpr-send**). Messages sent with **funcall** or **apply** are not recognized as messages and are not converted. The first time a particular message is encountered during a conversion operation, you will be prompted for the name of the replacement generic function. You can press RETURN to accept the offered default. You can press SPACE to complete to "None", which means to leave this message unconverted. You can later convert the message by hand, or leave it as a message if your program runs partially in CLOS and partially in Flavors. A generic function name that is already in use as a special form, macro, or nongeneric function will not be accepted.

The following Flavors constructs are not automatically converted. They are left in the program and must be converted by hand. Some of these do not have any simple conversion to CLOS; if you use them, you might have to restructure your program. Others are used infrequently enough that automatic conversion did not seem worthwhile.

compile-flavor-methods
define-method-combination
defwrapper
:fasd-form

get-handler-for
lexpr-send-if-handles
operation-handled-p
recompile-flavor
send-if-handles
:unclaimed-message
:which-operations

In addition, none of the documented Flavors constructs in the **flavor**, **system**, and **zl** packages is converted. These are generally too internal to have exact correspondences in CLOS.

Mixed use of Flavors and CLOS is not supported at present. That is, a class cannot inherit from a flavor, a flavor cannot inherit from a class, CLOS generic functions cannot be used with Flavors methods, and Flavors generic functions cannot be used with CLOS methods nor with CLOS instances. The only supported mixed use is that CLOS generic functions can be used with Flavors instances and CLOS methods can be specialized to a flavor as if it was a class. Therefore if you convert a program to CLOS, you must convert all uses of a given flavor to use a class instead, and all Flavors methods for a given generic function to be CLOS methods instead.

Zetalisp to Common Lisp Conversion

The Zetalisp to Common Lisp Conversion Tools convert programs from Symbolics' Zetalisp dialect to Symbolics' extended version of Common Lisp. The kinds of changes needed to convert your code from the Zetalisp to the Common Lisp package structure and syntax are summarized in the six-step breakdown of the Zetalisp portion of the Conversion Tools system as follows:

1. Syntax Conversion
2. Radix Conversion (optional)
3. Name Conflict Resolution
4. Package Prefix Conversion
4a. CL Prefix Removal (optional)
5. Function Conversion
5a. Remaining ZL Prefix Removal (optional)
6. Structure Conversion

These six steps are executed entirely via Zmacs commands that run over files read into the editor buffer. The conversion commands are simple and easy to use; some are largely automated; others depend on your input for correct results. A Help facility, invoked by pressing the HELP key, is also available.

Each of the six steps produces a program that is complete, can be recompiled, and that functions equivalently to the source program. Each step is dependent on its predecessor and must, therefore, occur in the enumerated order, including the three optional steps, 2, 4a and 5a, if you choose to execute them.

Note

The Zetalisp to Common Lisp Conversion Tools do not necessarily produce portable Common Lisp code. Using the Common Lisp package is helpful, since after conversion Symbolics Common Lisp symbols will be identifiable by their **scl:** prefixes; but the Zetalisp to Common Lisp Conversion Tools do not especially flag conversions that might involve the use of functions that have Symbolics Common Lisp extensions to Common Lisp.

The Conversion Tools specifically don't do the following function conversions:

- Zetalisp functions for which there is no Common Lisp analog
- Zetalisp functions used inside a macro
- Complex Zetalisp functions, or those requiring rearrangement of the code in their vicinity

Unconverted functions will, of course, remain prefixed by **zl:** as a result of the Package Prefix conversion; they'll be easy to find either visually or with Zmacs Search commands.

We now present a step-by-step description of the Zetalisp to Common Lisp Conversion Tools, listing the Zmacs commands used to perform each step.

Step One: Syntax Conversion

Commands:

```
m-X Convert Lisp Syntax of Region
m-X Convert Lisp Syntax of Buffer
m-X Convert Lisp Syntax of Tag Table
```

Zetalisp uses "/" as the syntax-quoting (escape) character. Common Lisp uses "\" for that purpose. Since the syntax is reliable in both cases, the conversion is automated.

The syntax conversion done in this first step results in the following changes:

ZL character	*Becomes CL character*
//	/
#/	#\
#\	#\
/	\
\	\\
#^	**#\control-**
#Q	**#+lispm**
#M	**#+Maclisp**
#N	**#+NIL**

(The last four character changes are listed only for the sake of completeness; they are unlikely to appear in any files.)

The buffer syntax conversion tool changes the syntax in the file attribute line to Common Lisp. The tag table syntax conversion tools asks whether to change the syntax in the file attribute line; normally you answer yes.

Syntax Conversion of a Buffer

Type:

```
m-X Convert Lisp Syntax of Buffer
```

The system verifies the name of the input buffer, as usual for file and buffer related commands. (Use HELP to see a display of all your buffer names, if you don't want to convert the current buffer.)

The syntax conversion then proceeds automatically. The file attribute line is changed to Common Lisp from Zetalisp, and all uses of the Zetalisp quoting character "/" have changed to the Common Lisp quoting character "\".

Syntax Conversion of a Region

Command: `m-X Convert Lisp Syntax of Region`

To apply the syntax conversion to a region, first mark the region, then issue the conversion command. Only the region you marked is converted, and the rest of your code remains unchanged.

Syntax Conversion of a Tag Table

Command: `m-X` Convert Lisp Syntax of Tag Table

You can use the Tag Table option of the conversion command if you have previously issued either `m-X` Select System As Tag Table or `m-X` Select Some Files As Tag Table. When you type the conversion command, the system first reads in all the files in the Tag Table that are in Zetalisp. It then displays a list, "Files to be converted" and asks whether to convert all, none, or some of these files. If you opt for conversion, the system asks whether it should set the file attribute lists also. The syntax conversion then proceeds automatically; the system displays the name of each file as it is converted, ending with the message "Done."

Be sure to test your program after the syntax conversion, and to save the buffer before proceeding to the next step.

Step Two: Radix Conversion (Optional)

If any of your files have a base of 8 (octal), you may wish to convert them to base 10 (decimal); if so, you must do it now. This step is completely optional, since Common Lisp programs can have nonstandard radices.

Commands:
```
m-X Convert Base of Region
m-X Convert Base of Buffer
m-X Convert Base of Tag Table
```

After verifying the buffer (or files, in the case of Tag Tables) to convert, the conversion commands give you the option of having large octal numbers (numbers larger than 10 octal) converted to decimal automatically or selectively. If you opt for selective conversion, you are queried separately for each "large" number in your program. The range of responses is:

Action	*Character or Key*
Do it	Y, SPACE
Skip it	N, RUBOUT
Manual Edit	c-R
Do it, then Allow Edit	, (comma)
Redisplay screen	c-L, REFRESH

If you edit your program manually during conversion, press END to signal completion of your edit and return to the current conversion step.

Step Three: Name Conflict Resolution

Commands:
```
m-X Find Conflicting Symbols in Buffer
m-X Find Conflicting Symbols in Tag Table
```

In your Zetalisp program you may have defined some functions or variables whose names will conflict with the names of Common Lisp functions after conversion. For instance, you might have defined a function named **search** that conflicts with the Common Lisp function **search**.

Before your program can be moved to a new package, all function and variable names that conflict with the names of functions or variables inherited by the new package must be changed. This happens in two stages. First the Conversion Tools generate a buffer that lists all conflicting symbol names. You then edit this buffer, typing a new name after each of these symbols. During this edit you can also delete lines whose symbols are not truly conflicting, such as symbols used only for naming local variables. Then use m-X Multiple Query Replace command to substitute the new name for the old one in the source program.

Name Conflict Resolution of a Buffer

The output buffer produced by the Syntax Conversion step is our input buffer for this next step. Type:

```
m-X Find Conflicting Symbols in Buffer
```

After verifying the filename, the system asks how the buffer's package will be converted. See the section "Step Four: Package Prefix Conversion".

Select your choice. Now the system creates a buffer:

```
[Creating ZMACS Buffer "Conflicting symbols".]
```

and places you in it. In our example, the Conflicting Symbols buffer might contain only the name

```
SEARCH
```

Now edit the buffer to supply a new name. The new name must appear on the same line as the old name, immediately following it. After editing, our sample buffer of conflicting symbols looks like this:

```
SEARCH  MY-SEARCH
```

Move back to your input buffer and type:

```
m-1 m-X Multiple Query Replace from Buffer
```

(`m-1` specifies that only whole-word occurrences of the symbol are picked for substitution.) The Query Replace command works in the usual fashion, letting you confirm or bypass the substitution in each individual case.

Name Conflict Resolution of a Tag Table

Command: `m-X` Find Conflicting Symbols in Tag Table

If you are working with a Tag Table, the conversion command asks how the packages used by files in the Tag Table will be converted. It then creates the buffer of conflicting symbols, which you edit as described earlier. Now type:

```
m-1 m-X Tags Multiple Query Replace from Buffer
```

Zmacs displays each occurrence of a symbol to be replaced, and you respond with an instruction to replace or skip, as appropriate, then press `c-.` to continue. See the section "Performing Operations with Tag Tables". The system notifies you when it has finished all substitutions.

Step Four: Package Prefix Conversion

Commands:

```
m-X Convert Package of Region
m-X Convert Package of Buffer
m-X Convert Package of Tag Table
```

This step modifies a program so that it can be read in a Common Lisp package instead of a Zetalisp package, but still get the same symbols. Each time a symbol in-

herited from Zetalisp is referenced, if the same symbol does not occur in Common Lisp a **zl:** package prefix is inserted in front of the symbol. No symbol substitution is done; the file is just changed to read the same global symbols into the new package. For instance, **memq** is converted to read **zl:memq**, not to the corresponding Common Lisp function name, **member** (that translation occurs later).

For a description of this step, see the section "Package Conversion".

A number of Zetalisp global symbols are much more often used as the names of local variables than as functions; examples are **zl:args** and **zl:array**. The Conversion Tools display a list of such symbols and offer you the option of suppressing their conversion. This will later save you the trouble of removing the unnecessary **zl:** prefixes generated by the conversion. You can edit the list, adding or removing symbols as required.

Step Four-a: CL Prefix Removal (Optional)

If your program was already using functions in the Common Lisp package by means of an explicit **cl:** package prefix, you may want to remove these prefixes now. Use the standard Zmacs Search and Replace commands for this purpose.

Step Five: Function Conversion

Commands:

```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

Many Zetalisp functions and variables readily translate into their corresponding Common Lisp functions. Three such types of translation are performed in this step.

The simplest translation is the direct substitution of one symbol for another; you can opt to have the system do these without querying you about each case. Other translations may involve some changes such as the addition of a keyword, the changing of an expression to a list, a changed order of arguments, and so on. In such cases the conversion command displays the proposed change, and asks you to confirm it. The most complicated renamings involve cases where there may be more than one option for translating a given Zetalisp function. Here, the conversion command displays the available options, along with some explanation about each, and asks you to select the most appropriate among them. This is where maximum familiarity with the code and the conversion set comes into play.

Before doing the function conversion, the system also prompts you for a conversion set; typically this is Zetalisp to Common Lisp. It can also be any of several other predefined conversion sets, such as Flavors to CLOS, or a conversion set that you have defined yourself.

As already stated, Zetalisp functions without a Common Lisp analogue, Zetalisp symbols not appearing their usual syntactic context, and complex Zetalisp functions are not translated in this step. However, all such untranslated functions remain

prefixed by **zl:** as a result of Package Prefix Conversion, so you can find them and deal with them yourself later.

The substituting mechanism is careful not to lose comments; they may not, however, end up exactly in the right place in the rearranged program. Indentation might also have changed. You should look the program over as the conversion progresses.

Function Conversion of a Buffer

The output buffer produced by the Package Prefix Conversion step is our input buffer for this next step. Type:

```
m-X Convert Functions of Buffer
```

After verifying the file name as usual, the Conversion Tools prompt for a conversion set; the options can be displayed by pressing HELP.

The **defstruct** option is for Structure conversion, the final conversion step. Here, select Zetalisp to Common Lisp as the conversion set.

Next indicate whether or not you want to be queried for straightforward renamings. Since these are simply direct symbol substitutions, the favored approach is to let the system do them automatically.

After doing the simple conversions, the system uses a typeout window to query you on all other substitutions. Before each query, the system displays the affected line(s) in bold along with the immediate context so you can identify the code. For a table of your possible actions at this point, see the section "Getting Help with Conversion".

Function Conversion of a Region

Command: m-X Convert Functions of Region

This works analogously to the Buffer version of the command, except that it operates only on the region you have marked.

Function Conversion of a Tag Table

Command: m-X Convert Functions of Tag Table

The sequence of events is as for the Buffer version: the conversion command prompts you for a conversion set, asks if you want to be queried for simple renamings, and displays proposed changes or options, requesting a response. It displays each file name as it finishes its conversion; its last message is "Now no more sets of possibilities."

Step Five-a: Remaining ZL Prefix Removal

Some Zetalisp functions may have been too complicated to convert automatically because the change requires rearranging the code in the vicinity of the function.

Structure definitions and constructors are still unconverted, and should be left with their **zl:** prefixes until the Structure Conversion step.

Some Zetalisp functions might remain, however, because a global symbol is really being used in an innocent way and is not suppressed explicitly during the initial package conversion. Such **zl:** prefixes can be removed altogether, since the new local symbols serve better. Use the standard Zmacs Search and Replace commands to do this.

Step Six: Structure Conversion

Commands:

```
m-X Convert Functions of Region
m-X Convert Functions of Buffer
m-X Convert Functions of Tag Table
```

To convert structures, you use the same command as for Function Conversion, except that this time you specify **defstruct** as the conversion set instead of Zetalisp to Common Lisp. As before, you have the option of doing straightforward renamings without a query. Keep in mind that since the system being converted is not guaranteed to be loaded, you will be asked about all forms beginning with **make-** that have an even number of arguments and no keywords other than **:make-array**. This may include a number of legitimate function calls.

For the most part, the **defstruct** macro is compatible with its **zl:defstruct** counterpart. However, some of the options accepted by both have a slightly different behavior when given to **cl:defstruct** than when given to **zl:defstruct**. In some cases, default behavior when no options are given also differs. There are, as well, differences in the format of the constructors generated by each **defstruct**; these are discussed in more detail below. For these reasons, we need a special conversion phase just to convert structure definitions and constructors. This phase deals with all **defstruct** forms, and all forms beginning with **make-** that appear to be structure constructor macros. You will probably need to do some editing of the conversion results, as we explain later.

In Zetalisp, you give the structure component names to the constructor macro as arguments for initializing these components, or slots. Common Lisp constructor functions take instead keyword arguments with the same name as the structure components. Further, Common Lisp constructor functions don't accept keyword arguments to the **:make-array** keyword. This Zetalisp form of a constructor macro, for example, is not acceptable to Common Lisp:

```
:make-array (:length 20)
```

The Conversion Tools let you get around this restriction by offering a Symbolics Common Lisp extension to **defstruct** that adds the option **:constructor-make-array-keywords** to a **cl:defstruct** macro. This option is followed by a list of all the **:make-array** arguments destined for use by the constructor function; these ar-

guments then become top-level arguments to the constructor instead of appearing as keyword arguments to the **:make-array** keyword argument in the constructor. Only arguments included in the **:constructor-make-array-keywords** option to **defstruct** can be used in the constructor function.

If you opt for this approach, and if during conversion the system finds that the **zl:** constructor contains **:make-array** keyword arguments that were not explicitly specified in the **zl:defstruct :make-array** option, it gives you an explicit message, telling you to add the missing keywords. The message is also recorded in the compiler warnings database to let you do your editing all at once after the conversion.

Warning: Currently, the constructor function always expands to a call to **zl:make-array**. This works in most cases, but could create problems if your Zetalisp structure definition specifies a type for the new structure (or any other keyword argument to **zl:make-array** that differs from a **make-array** keyword).

Structure Conversion of a Buffer

All previous phases of Common Lisp conversion must have already been executed on the buffer before performing Structure Conversion.

This section uses examples in the file

SYS:CONVERSION-TOOLS;CONVERSION-STRUCTURES-TEST-PROGRAM.LISP

Type:

```
m-X Convert Functions of Buffer
```

After verifying the filename to convert as usual, the conversion command asks for a conversion set to use; this time, type **defstruct** . As for function conversion, you are asked if the conversion should do straightforward renamings without query; answer Yes. Straightforward renamings include converting **zl:defstruct** to **defstruct**, changing the names of structure components in constructor functions to keywords, and changes involving different default behavior in Zetalisp and Common Lisp **defstruct**, as in the use of **:conc-name** in our sample.

After querying you for each conversion (press `HELP` for the full list of possible responses), the system produces the new code. Major changes are as follows:

All **zl:defstruct** macros have become Common Lisp **defstruct**.

In all the converted constructor functions, the names of the structure components have become keyword arguments.

Example differences in zl: and cl: defstruct default behavior

The treatment of the **:conc-name** option to **defstruct** points to differences in the default behavior of the Zetalisp and Common Lisp forms. The **zl:defstruct** for the structure **my-structure** contains a **:conc-name** option without an argument. This form of the option is the default for **defstruct**. Therefore, **:conc-name** becomes redundant and is removed during the conversion. In contrast, the **zl:defstruct** for the structure **header-structure** in the example does not contain a **:conc-name** option, because we wanted to use the default version of the **zl:defstruct**, which is

the **:conc-name nil** option. Since this is not the default behavior for **defstruct**, the conversion adds an explicit **:conc-name nil** option.

Treatment of :make-array keywords

The **zl:defstruct** for the structure **header-structure** in our example specifies that the structure should be implemented as an array, and uses a **:make-array** option to define the array length.

The Zetalisp constructor macro, **make-header-structure**, also uses the **:make-array** keyword, followed by two keyword arguments to it, namely **:length** and **:displaced-to**. To allow use of the latter two in the CL constructor function, which does not accept keyword arguments to **:make-array**, the conversion did the following:

It replaced the **:make-array** option in the **zl:defstruct** with the Symbolics Common Lisp option **:constructor-make-array-keywords**, following it with the former **:length** argument to **:make-array** which now serves as an argument rather than a keyword.

It also replaced the **:make-array** option in the Zetalisp constructor macro by the top-level keyword arguments, **:length** and **:displaced-to**.

While converting the constructor macro, the system finds that the constructor **:displaced-to** argument is missing from the argument list of the **:constructor-make-array-keywords** option in the **defstruct**. This is because it was not specified in the **zl:defstruct**, appearing only in the constructor. The system therefore displays a message, telling you to add this argument to the list of arguments following the **:constructor-make-array-keywords**, in the **defstruct**, so that the constructor function can subsequently use it. To deal with all such messages together at the end of the conversion, execute m-X Edit Compiler Warnings, which displays the Compiler Warnings database.

If your zl:defstruct used the :type option:

Suppose your original code specifies that the structure be implemented as a string array, as follows:

```
(zl:defstruct (header-structure :array-leader
                                (:make-array (:length 20
                                              :type 'zl:art-string)))
  slot-1)

(defun make-one ()
  (make-header-structure slot-1 'one :make-array (:length 40)))
```

This is then converted to the following Common Lisp form:

```
(defstruct (header-structure (:conc-name nil) :array-leader
                             (:constructor-make-array-keywords
                               (length 20) (element-type 'string-char)))
  slot-1)
```

```
(defun make-one ()
  (make-header-structure :slot-1 'one :length 40))
```

Since the constructor function currently expands to a call to **zl:make-array**, the form **(zl:make-array 20 ... :element-type 'string-char)** would result. This is not legal, since Zetalisp recognizes only **type** as an array type specifier.

This problem will be fixed in a future release. Currently, you can get around it by editing the converted code of the **defstruct** to restore the original Zetalisp type declaration:

```
(defstruct (header-structure (:conc-name nil) :array-leader
                             (:constructor-make-array-keywords (length 20)
                                                               (type 'zl:art-string)))
  slot-1)
```