Internals

The Scheduler

Overview of the Scheduler

About the Scheduler

A Symbolics machine has a single processor, multiplexed among several different processes. A *process*, loosely defined, is the state of a computation: a thread of control and an environment.

At any time a set of *active processes* exists; these are all the processes that are not stopped. Each active process is either running, trying to run, or waiting for some condition to become true. Many processes can run on a single processor only by sharing it. The processor performs work on behalf of each process, in turn, for a short period of time.

If all the processes are simply trying to compute, the machine time-slices among them. This is not a particularly efficient mode of operation, since dividing the finite memory and finite processor power of the machine among several processes does not increase the available power and in fact wastes some of it in overhead.

The processor could run one process at a time (until completion) and then start on the next process. This is the way that older computers often worked, batch-processing jobs. However, there are (at least) two reasons why having multiple active processes is desirable.

1. *Efficiency.* There are many times during the course of a single computation when the process is not using the processor. It is waiting for an external event (perhaps for a disk operation to complete, real time to pass, or for a user to type a character). Alternatively, it can be waiting for an internal event (for another process on the machine to change a value in a shared data-structure, or for some relationship between several objects in memory to become true). During that waiting period we can run other processes with no loss of performance.

2. *Convenience.* People seem to prefer a style of interaction in which the computer operates on their behalf in the background while they type characters at it (or provide input in other ways) in the foreground. Programmers, too, often find it easier to express an algorithm as if the activity is divided among several processes, each executing at the same time. The illusion that several processes are running concurrently allows the programmer to avoid dealing with issues of explicitly yielding the processor, or explicitly controlling the flow of two independent computations that might need to be interleaved.

The illusion that several processes are running concurrently comes at a price, though. First, there is some overhead associated with switching from one process

to another — the processor has to save and restore the context. Secondly, *synchronization* and *communication* become important. Synchronization and communication primitives are intimately connected to the scheduler. Programmers need to synchronize between two processes that are running "simultaneously" that share resources. Some examples of synchronization mechanisms used in other computer systems are locks, monitors, semaphores, and atomic-actions. Two cooperating processes that are running simultaneously presumably need to communicate. Since all Genera processes run in a single address space, interprocess communication is straightforward. Two or more processes can share arbitrary objects in memory.

If all processes waited (went *blocked*) often enough, and long enough, to allow other processes to get their fair share of the machine, the system would have no problems in managing the processor. When a process yielded the processor, the scheduler would simply switch to another process, eventually satisfying them all. Unfortunately, the processor is a scarce resource on most computer systems. Many processes want as much time on the processor as they can get.

The scheduler is the part of the system that manages the processor and implements processes. It must allocate the processor "fairly" between competing processes. Therefore it cannot always wait for a process to voluntarily yield the processor, occasionally it must preempt the currently running process. The decision of when to preempt a process, and what process to switch to, is the province of the scheduler.

How the Scheduler Works

The scheduler consists of three parts:

- A *context switcher*
- A *dispatcher*
- A *priority manager*

The context switcher switches from one process to another. It takes care of the mechanics of transferring control from one process to another, and making certain that each process executes in the proper execution context.

The dispatcher runs every time a process yields the processor, and every time the current process is preempted. Its duty is to determine the highest priority *runnable* process on the machine, and call the context-switcher to switch to it.

The priority manager translates priorities explicitly assigned by a program or a programmer to internal scheduler priorities.

In the Genera system, the context switcher is simple. All it needs to do is execute a stack-group switch. All of the context and the state of the computation for a process is captured in the stack group. See the section "Stack Groups". Since all Genera processes execute in a single address space there is no need to change virtual memory, switch page tables, or do any of the assorted things that other computers

might do. The context switcher does have to restore the binding environment and set up the processor to begin executing the next instruction in the new process.

The dispatcher makes sure that the currently running process (stored in the variable ***current-process***) is always the highest priority runnable process.

Two runnable processes never have equal priority. The priorities used by the dispatcher to order processes on the RUNNABLE queue are internal priorities used only by the system. If two priorities are equal, the process that is closer to the head of the RUNNABLE queue is defined to have higher priority. The reason that this arbitrary ordering is not unfair is because the order changes over time.

It is the job of the *priority manager* to translate from programmer priorities to internal dispatcher priorities. The priority manager makes sure that at any given time there is an internal ordering of all processes. This order can change over time as the priority manager decides that one process is getting too much of the processor, or another process is getting too little.

The priority manager also has an internal process that runs periodically and performs various tasks necessary for its operation — for example, processing statistics about process utilization.

Any time a program sets the priority of a process, the priority manager translates that priority to an internal priority. The criteria it uses to do this is called its *policy*. The default policy combines the program specified priority with the recent history of the process in order to compute the internal dispatcher priority. To learn more about the default policy, see "Priorities and Scheduler Policy". Additionally, see the function **process:allow-preemption**.

The priority manager can use any criteria — static priorities, recent history, past performance, or a round robin scheme. You can write your own scheduler policy; see the section "Extensible Scheduler". The priority manager also runs periodically, interrupting compute bound processes, to recompute the priorities of processes that do not voluntarily give the priority manager a chance to adjust their internal priority.

Together, the context switcher, dispatcher, and priority manager constitute the scheduler. This section discusses the scheduler.

There are other facilities that are tightly connected to the scheduler: the timer facility, synchronization primitives, managing the control-abort gesture, and warm boot actions. They are documented here, too.

We also briefly discuss ways to customize the scheduler.

Scheduler Concepts

Using Processes for Computations

To start a computation in another process, you must first create a process and specify the computation you want to occur in that process. The computation to be executed by a process is specified as an *initial function* for the process and a list

of arguments to the initial function. When the process starts up it applies the function to the arguments. In some cases the initial function is written so that it never returns, for example, a Lisp Listener that has a **read-eval-print** loop that just keeps going, while in other cases it performs a certain computation and then returns, which stops the process, for example a **compile-file** operation.

When you create a process, it is active, assuming you have provided the necessary initial function and, for a simple process, a *verify function*. You can override this default by use of the **:run-reasons** keyword to **process:make-process**.

To *reset* a process means to "throw out" of its entire computation, then force it to call its initial function again. (See the special form **throw**.) Resetting a process clears any waiting condition, and so if it is active it becomes runnable. To *preset* a process is to set up its initial function (and arguments), and then reset it. This is one way to start up or change a computation in a process.

To *interrupt* a process means to tell it to execute some function on your behalf in its environment. If the function returns normally the interrupted computation is resumed. You can interrupt a process in order to throw out of its computation, though, in which case you have aborted the interrupted computation.

To *abort* a process means to throw out of a process by signalling **sys:abort**, but respecting processes that are in the dynamic extent of a **sys:without-aborts** form. (See the function **sys:without-aborts**.).

For functions that allow you to *create, preset, reset, interrupt*, or *abort* a process, see the sections "Creating and Enabling Processes" and "Resetting, Interrupting, and Aborting Processes".

Process States

A process can be in one of several states.

When a process is created it is *alive*. When it is killed it is *dead*. A dead process is inaccessible to functions such as **process:map-over-all-processes** and **process:map-over-active-processes**. A dead process consumes very few resources — it is *frozen*. A dead process can be restored to life by *resetting* and *enabling* it. (For more information about resetting a process, see the section "Resetting, Interrupting, and Aborting Processes". For more information about enabling a process, see the section "Creating and Enabling Processes".)

Live processes can be further subdivided into *active* and *stopped* processes.

A process has two sets of Lisp objects associated with it, called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically, keyword symbols and active objects such as windows and other processes are found. A process is considered *active* when it has at least one run reason and no arrest reasons. A process that is not active is *stopped*.

Stopped processes are not considered by the scheduler. They cannot run until they first become active. Stopped processes can either be *suspended* or *frozen*. A suspended process is "ready to run" — if it were active, it could become runnable. A

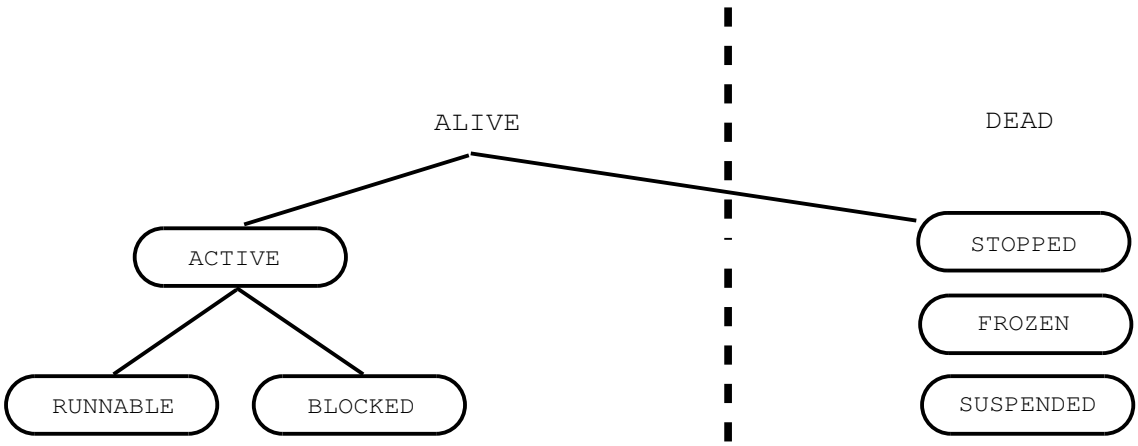


Figure 28. Process States

frozen process has released all of its resources, and needs to reacquire a stack-group (for example) in order to become active. The internal state of a frozen process is the same as the internal state of a dead process except that the frozen process is still accessible to the process mapping functions (it shows up in the Peek display, and in the output of the Show Processes command.) Stopped processes existed in the old scheduler, and therefore functions and documentation refer to them. They are interchangeably referred to as *stopped*, *arrested*, or *disabled* processes. *Stopped* process is the preferred term.

An active process is either *runnable* or *blocked*. The term *active process* is synonymous with *enabled process* and with *unarrested process*. These terms are used interchangeably throughout the existing documentation. *Active* process is the preferred term.

A process is *runnable* when it can use the processor immediately. A process is *blocked* when it has yielded the processor and has no need to use it.

If a process is *blocked* you can try to make it *runnable* by calling **process:wakeup** on it. If the verify function returns **nil**, however, it does not become runnable. See the section "Blocking, Waiting, and Waking Processes". If a process is *runnable* it can go *blocked* by yielding the processor. The simplest ways to go blocked are to call **process:block-process**, **process:block-and-poll-wait-function**, or **process-wait**.

For more information about functions to add and delete objects to the set of run and arrest reasons for a process, see the section "Creating and Enabling Processes".

See the section "About the Scheduler".

Blocking Vs. Waiting

When a process yields the processor it specifies a verify-function. The verify-function is a predicate that evaluates to a non-**nil** value when the process is ready to run again.

The new scheduler is a hybrid event-driven and polling scheduler which means that the programmer has a choice of whether to *block* or *wait*. To *block* means that the programmer believes that some other process (or interrupt handler) has accepted the responsibility to *wakeup* this process when its verify function becomes true. To wait means that the programmer request the scheduler to periodically poll the verify function and, when it becomes true, wakeup the process.

A process can block by calling **process:block-process**. If no wakeup is forthcoming, or if the wakeup occurs before the verify-function becomes true, the process might never be awakened. This is a "lost wakeup" and is one of the dangers of an event-driven scheduler.

A process can wait by calling **process:block-and-poll-wait-function**. When it does this, the scheduler consumes system resources in order to poll the process.

See the section "Event-Driven Scheduler".

When a process goes blocked and provides a verify-function, the verify-function is tested before the process goes blocked. If it returns a non-**nil** value the process does not go blocked.

process:block-process does not return unless the verify-function has been true at least once after the process went blocked.

A common idiom is to want to block, or wait, for an event, but to give up after a certain interval has passed. **process:block-with-timeout**, **process:block-and-poll-with-timeout**, and **process:wait-with-timeout** allow the programmer to specify a timeout period (in seconds), after which the function should just return.

(**process:wakeup** *process*) evaluates the verify-function of *process*. It does not wake up a process unless the verify-function returns a non-**nil** value. It is not always convenient, legal, or possible to execute an arbitrary piece of code from another process. **process:wakeup-without-test** wakes up the process without testing its verify-function. The verify function is tested, instead, in the target process by **process:block-process**. It does not return to its caller unless the verify function is true. If the verify function returns **nil**, the process goes blocked again.

Verify-Functions and Wait-Functions

When a process goes blocked, it supplies a *verify-function* to the system. The first purpose of the verify-function is to avoid unnecessary process switches. Since stack-group switching is expensive, we first evaluate the verify-function to verify that the process is in fact runnable. A call to **process:block-process**, or a related function, returns only if the verify-function evaluated to a non-**nil** value at least once since the call to go blocked.

The second purpose of a verify-function is to allow more flexibility in setting the priority of a process. The value returned by a verify-function is treated as a process priority. If it is **nil** or **t**, the priority is not changed. If it is any other value, the priority of the process is set to the returned value.

This behavior is useful because if a process goes blocked, waiting for more than one event, it does not know which path it will take after it awakes. It is conceivable that some events are higher priority than others. If we required the process to set the priority itself, after it is awakened one of the two following situations may occur.

If a process, P, waits with the highest priority it may take after awakening, P can needlessly preempt another process before P lowers its own priority. If, on the other hand, P waits with the lowest priority, and another process is runnable when P wakes up, that other process may hold P off for a long time before P gets a chance to raise its priority. Finally, it is convenient to be able to set **:deadline** priorities from the time a process is awakened, rather than from the time it was blocked.

When a process waits instead of blocking (see the section "Blocking Vs. Waiting") the verify-function has one more purpose. It is periodically polled by the wait-function poller, and if it returns a non-**nil** value the process is awakened. The verify function of a waiting process is called the *wait-function*.

Verify-functions can be executed in dynamic environment — if another process calls (**process:wakeup** *p*), then *p*'s verify-function will be evaluated in the dynamic environment of the waking process.

Thus, extreme care must be taken in designing wait-functions.

- A verify-function cannot depend on any special bindings. It cannot depend on the binding environment being empty, either. The verify- function is not executed in the global environment — it is executed in some processes dynamic environment. This means that you cannot "hide" a special from a verify-function by binding it in your process. This restriction has some non-obvious corollaries — for example, you must call (**si:follow-syn-stream** stream) on any stream that you pass as an argument to a verify-function, since a synonym-stream might depend on the current binding of ***terminal-io***.

- A verify-function must return either **t**, **nil**, or a process priority created by **process:make-process-priority**.

- A verify-function cannot go blocked or wait.

- There is no guarantee that a verify-function will only be executed once after it becomes true. For example, you must be wary of code that uses **store-conditional**, or any atomic operations.

- It is desirable in the new scheduler, but not necessary, that a verify-function does not start returning **nil** after it becomes true, until after the process becomes runnable. This condition was critical in the old scheduler.

Verify Function Compatibility Note

In the old scheduler all verify functions were called wait functions. They *were* evaluated in the global environment, so it *was* possible to hide values from a wait-function by binding the variables in your own process.

The value returned by a wait-function in the old scheduler was not interpreted as a priority — the only consideration was whether or not it returned **nil**. Therefore, for compatibility, all wait-functions specified by calling an old scheduler entry-point have an implicit **(not (null** ... *body* ...**))** wrapped around them.

All of the other constraints also held in the old scheduler.

Priorities

At any given instant in time, all of the runnable processes on the machine are strictly ordered. The *dispatcher* makes sure that the highest priority runnable process is the ***current-process***.

The *priority manager* converts process priorities to *instantaneous* priorities to facilitate this ordering. These instantaneous priorities are internal dispatcher priorities. Programmers or users need never concern themselves with them. *Process priorities* do not impose a strict ordering on the processes. They are interpreted by the priority manager, and converted to dispatcher priorities. The rules that govern these conversions are the implementation of the scheduler policy. See the section "Priorities and Scheduler Policy". The text that follows is specific to the current implementation of scheduler policy.

There are three types of priorities: **:deadline**, **:foreground**, and **:background**.

You can specify a :deadline priority by using **process:with-process-priority** with a **:deadline** priority. The deadline is specified in microseconds, and refers to the maximum desired elapsed time spent in the body of **process:with-process-priority**. All processes executing with **:deadline** priorities are higher priority than any process with a **:foreground** or **:background** priorities. Although you specify deadlines in *relative* microseconds, the scheduler converts the deadline to an absolute completion time. The processes are ordered by earliest completion deadline first.

process:with-process-priority defines a priority *region* — the priority will only be set inside that form, and will revert to the previous priority on exit. Because of the dangers involved with careless use, **:deadline** priorities can only be set within regions and not by calls to **process:set-process-priority** or equivalent functions. This behavior is controlled by **process:*policy-hook-region-priority*** and **process:*policy-hook-set-priority***.

If you are already within a priority region (within a call to **process:with-process-priority** or an equivalent region defining form), you can set the priority with calls to **process:set-process-priority** and the set of legal priorities is the same as the set for a region. The priority change has limited extent. Upon exiting the innermost priority region, the old priority is restored, and all changes made by **process:set-process-priority** are lost. The priority regions nest, and form a stack of process priorities. The bottom of the stack is called the *base* priority. The function **process:set-process-base-priority** can be used to set the base priority.

:foreground priorities have integer values. They represent different priority levels. The larger (more positive) the number, the higher priority. Foreground priorities are analogous to old scheduler priorities. Unlike old scheduler priorities, however, a priority 1 process will not always take precedence over a priority 0 process.

This behavior is controlled by the scheduler parameters. Currently the scheduler parameters are global — there is one set for the entire system. At some time in the near future, scheduler parameters will be settable on a per-process basis, as part of the specification of a foreground priority.

The scheduler parameters are: **:record**, **:hysteresis**, **:resolution**, **:spread**, **:boost**, **:promotion-boost**, and **:wakeup**.

In the normal course of events priority 1 processes run ahead of priority 0 processes. If a foreground process holds the processor for N% of the time, its priority is degraded by **:spread** times N%.

The computation of N% is controlled by **:record**, **:hysteresis**, and **:resolution**. In order for the priority manager to recompute the priority of a compute bound process that does not explicitly call it, the priority manager must wakeup every so often and recompute the dispatcher priorities. The interval between wakeups is measured in seconds and controlled by **:wakeup**.

Every few seconds the priority manager captures the state of all runnable processes. This interval is controlled by **:record**.

:hysteresis controls the number of recording intervals used to compute N%.

N% is rounded to within the value of **:resolution**.

If a process is runnable, but is denied the processor by higher priority processes for S seconds, its priority increases by S times **:boost**. (Its dispatcher priority is incremented by **:boost** each second.) For more information about the scheduler parameters, see the section "Priorities and Scheduler Policy".

If a process has a **:preemptive** priority, it immediately preempts the current process upon becoming runnable, if the current process has a lower priority. If a process is not preemptive, then it does not preempt a process of a lower priority unless the lower process explicitly yields the processor or another preemptive process preempts the currently running process. A process that does not have a preemptive priority does not initiate a process switch, but if one occurs, then that process is considered in the pool of runnable processes.

Choosing Process Priority Levels

The following are some guidelines about what values to use when you modify a process's priority.

Normal processes run with a default **:foreground** priority of 0 (controlled by **process:*default-process-priority***) when computing and a **:foreground** priority of 1 (controlled by **process:*process-interactive-priority***) when they are interacting with a user. If the priority number is higher, the process receives higher priority. You should avoid using priority values higher than 9, since some critical system

processes use priorities of 10 to 30; setting up competing processes could lead to degraded performance or system failure. You can also use negative values to get processes to run in the with even lower priority. Values of -5 or -10 for unimportant processes and 2 or 5 for urgent processes are reasonable.

Only the relative values of these numbers are important. Once these relative priority values are set, be advised that the process priorities are interpreted consistently.

Although processes with priorities of -5 consume very little of the machines resources, they still take a toll. If you really do not care when the process runs you can specify a **:background** priority, which effectively submits the process as a background job that runs whenever the machine is idle. All background processes have equal priority. All foreground processes (even those with priorities of -100) have priority over background processes.

If there is a critical piece of code with a real time deadline (for example, a deadline between loading a register and reading it, or a deadline in responding to an interrupt from a device) you can use **process:with-process-priority** with a **:deadline** priority. Setting a deadline priority provides no guarantee that the deadline will be met, just a promise that the best attempt will be made.

A process with a deadline priority has a higher priority than all foreground processes, and is therefore as dangerous as executing code within a **without-interrupts** form. Deadline priorities should be used sparingly, if at all.

Use the Command Processor command Show Processes to see the priorities used by existing processes.

Promotion

Problem: If there are three processes P1, P2, and P3, that have priorities PR1, PR2, and PR3 respectively, and PR1 > PR2 > PR3. (where priority-1 > priority-2 implies that a process with priority-1 has precedence over processes with priority-2).

If P3 holds some resource that P1 is waiting for, and if P2 is compute bound, effectively there is a subversion of the priority system, since P2 now has effective precedence over P1.

There are many variations on this theme.

Solution: If a resource obeys the promotion protocol, when P1 waits on a resource held by P3, P3 is *promoted* to the priority of P1.

This problem and its solution comes up in many guises in the scheduler. The most common case is locking. Promotion is built into the locking substrate, so if there is contention for locks, the scheduler will promote the process holding the lock to the priority of the highest priority waiter. (In the case where the lock's owner is waiting for another lock, in turn, promotion recurses.)

In order to make promotion available as a solution to similar problems elsewhere in the system we have separated the promotion protocol out of locking, and documented it.

For the specification of this protocol, and the functions involved, see the section "Promotion Protocol".

Promotion occurs "under the covers". When a process is promoted, its visible priority remains the same. In scheduler terms promotion occurs at the level of *dispatcher priorities*, not *process priorities*.

Comparison of New and Old Scheduler Functionality

There are two important changes in the way the scheduler works:

- The new scheduler is event-driven; the old scheduler polled.
- The new scheduler does not have a separate stack-group; the old scheduler did.

This means that you can write programs that use processes more efficiently than in the old scheduler.

The new scheduler two new features that the old one did not have: a more efficient timer-facility, and a slightly wider range of priority types (**:deadline** and **:background**).

Event-Driven Scheduler

The old scheduler was a polling scheduler. That means that when a process gave up the processor it provided a function (called the *wait-function*) for the scheduler to test. Every time the scheduler ran, it evaluated this function for each process. If the function for process P returned a non-null value, the scheduler ran process P (assuming it was the highest priority runnable process).

This has the disadvantage that the scheduler spends a lot of time running these functions. It has the advantage that you can wait for complicated events to happen, and you do not require the co-operation of the program causing these events.

The new scheduler is event-driven. This means that the new scheduler evaluates the wait-function (called the *verify-function* in the new scheduler) significantly less frequently than the old scheduler. It does *not* evaluate the verify function *every* time it runs. Some other process has to explicitly *wakeup* process P when you want it to run again. (When an attempt is made to awaken it, the verify-function is run to verify the wakeup, to make sure we are not making a useless stack-group switch.) If no other process wakes up a blocked process, it will never run again A process that waits for another process to wake it up is in the BLOCKED state. The new scheduler also provides a polling option. Processes that wait for their verify functions to become true, without expecting an explicit wakeup, are in the WAITING state.

Waiting processes are treated as in the old scheduler. A process runs every so often and polls the verify-functions of the waiting processes. This is almost, but not exactly, indistinguishable from the old scheduler. There are two ways a process can *wait*:

- Explicitly call **process:block-and-poll-wait-function**, which allows you to set the interval that you want your wait function to be polled.

- Simply call the old **process-wait**, which will poll it every **process:*process-wait-interval*** seconds. (The new scheduler uses a default **process:*process-wait-interval*** of 1/6 second)

Waiting processes are actually polled more frequently than their interval specifies. The interval is only used to determine the polling rate when the machine is busy. When the machine is (relatively) idle the wait functions are polled every **process:*background-wait-function-polling-interval*** seconds. Additionally, the null process polls all wait functions for a short interval after a process starts waiting. This is done for **process:*idle-time-wait-function-polling-interval*** microseconds after any process starts waiting.

The difference between waiting processes in the new scheduler and processes in the old scheduler is that waiting processes are only polled at least every interval. In the old scheduler they were polled every process switch. The tradeoff here is: the less frequently you poll these processes, the less overhead the wait-function-poller uses, but the less responsive these processes are.

Also, evaluating these verify functions consumes system resources. The fewer processes are "waiting" (as opposed to being "blocked") the more frequently we can poll the verify-functions. So, it is important to convert as much of the system as possible to use block/wakeup when the new scheduler is running. If we do this, our code will be faster, and the wait-function-poller will use up less of the system *and* still be responsive.

Processes that block and are explicitly woken up are both more responsive and more efficient.

Processes that process-wait, but are heavily involved in interprocess communication, and expect rapid response can be less responsive than in the old scheduler when the machine has compute-bound processes.

Example of Blocking, Waiting, and Process-switching in the Old and New Schedulers

Let's look at an example of how wakeups can improve performance of cooperating processes.

- Process C waits for some arbitrary condition, runs for *n* milliseconds and sets the value cell of Y to **t**

- Process B waits for Y to be true and then runs for *n* milliseconds and sets X to **t**.

- Process A waits for X to be true and then runs for *n* milliseconds.

All processes have the same priority.

(The numbers for stack-group switch and evaluating wait-functions are not accurate, but are chosen to illustrate a point.)

Let's say *n* is 1 (millisecond). Here is the exact sequence of events:

In the old scheduler:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Switch to scheduler stack group
500 microseconds
   evaluate wait-functions and find that B is now RUNNABLE.
8 milliseconds
   switch to B
500 microseconds
   B starts running
1 millisecond
   B sets X to 'T
   Switch to scheduler stack-group
500 microseconds
   evaluate wait-functions and find that A is RUNNABLE.
8 milliseconds
   switch to A
500 microseconds
   A starts running
1 millisecond.
```

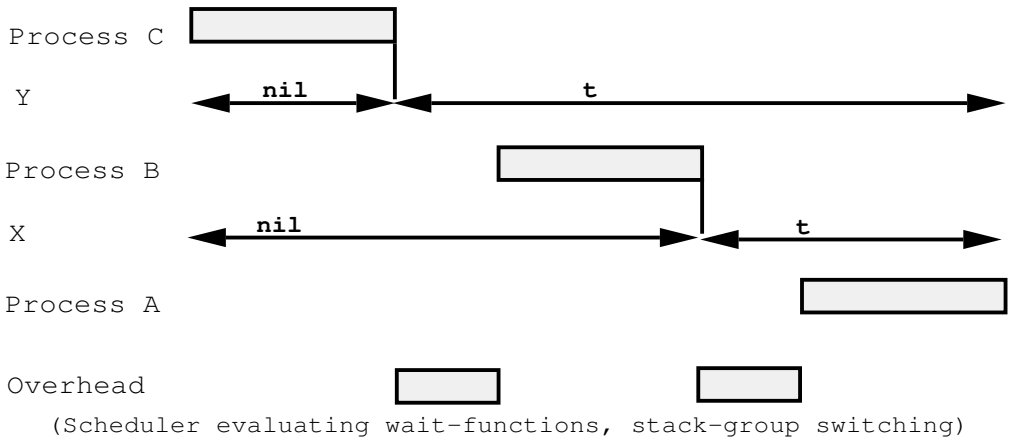


Figure 29. Stack Group Switching in the Old Scheduler

So, elapsed time is around 21 milliseconds.

If we convert it to block and wakeup by adding a **process:wakeup** after setting Y or X, though, in the new scheduler:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C wakes up B
   Switch to B
```

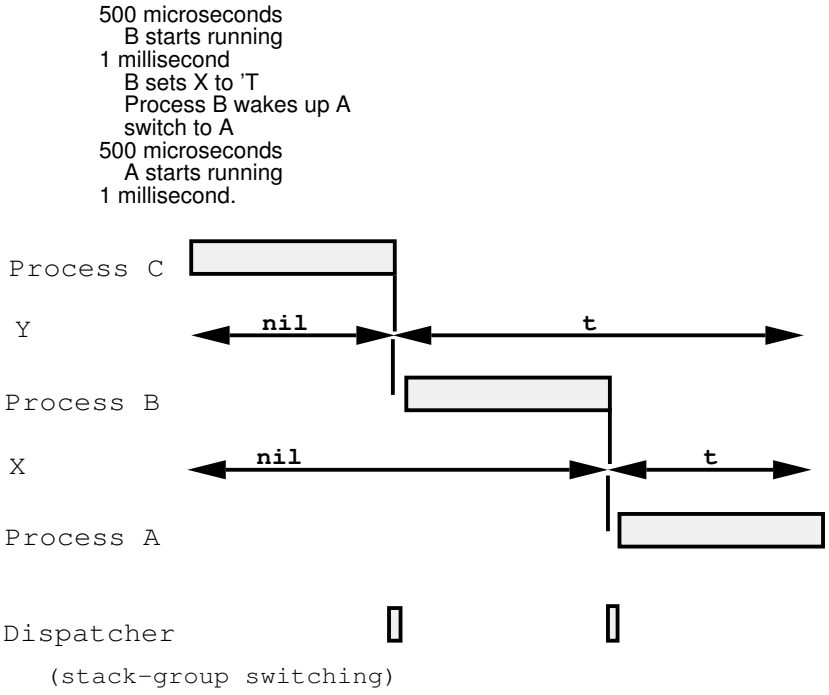


Figure 30. Blocking and Waking with the New Scheduler

Elapsed time is about 4 milliseconds.

So converting to the new scheduler, and converting your code to using block and wakeup yields a significant improvement (4, rather than 21, milliseconds).

If we do not convert our code but still run in the new scheduler, this is what happens:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Switch to Null Process
500 microseconds
   Null process is evaluating wait-functions since C just started waiting.
   evaluate wait-functions and find that B is now RUNNABLE.
10 milliseconds
   switch to B
500 microseconds
   B starts running
1 millisecond
   B sets X to 'T
   Switch to Null Process
500 microseconds
   Null process is evaluating wait-functions since B just started waiting.
   evaluate wait-functions and find that A is RUNNABLE.
10 milliseconds
   switch to A
```

500 microseconds
A starts running
1 millisecond.

So the elapsed time is about 25 milliseconds, which is comparable to the old scheduler.

On the other hand, let's look at an extreme and unrepresentative example, designed to show worst-case performance of the compatibility stubs for the new scheduler. In a case like this, you sometimes need to convert your code in order to get performance comparable to the old scheduler.

Let's look at the three processes again, but add a slightly more complicated condition.

- Process C waits for some arbitrary condition, runs for N milliseconds and sets the value cell of Y to 'T, and Z to the value of the microsecond-clock.
- Process B waits for Y to be true and then runs for N milliseconds and sets X to 'T.
- Process A waits for X to be true and 20 milliseconds to have elapsed since Z was set, and then runs for N milliseconds.

All processes have the same priority.

(The numbers for stack-group switch and evaluating wait-functions are not accurate, but are chosen just for reasons of this example.)

We should examine six cases.

1. These three processes running under the old scheduler when the machine is idle.
2. These three processes running under the old scheduler when there is a compute-bound process on the machine.
3. These three processes running under the new scheduler when they are event-driven.
4. These three processes running under the new scheduler when they are event-driven and there is a compute-bound process on the machine.
5. These three processes running under the new scheduler without wakeups, when the machine is basically idle, simply using the compatibility stubs.
6. These three processes running under the new scheduler without wakeups, when there is a compute-bound process on the machine, simply using the compatibility stubs.

Let's say N is 1 (millisecond). Here is the exact sequence of events in each of the cases:

In the old scheduler, when the machine is idle:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C sets Z to the value of the microsecond-clock.
   Switch to scheduler stack group
500 microseconds
   evaluate wait-functions and find that B is now RUNNABLE.
8 milliseconds
   switch to B
500 microseconds
   B starts running
1 millisecond
   B sets X to 'T
   Switch to scheduler stack-group
500 microseconds
   evaluate wait-functions and find that A is not RUNNABLE. (only 18 and 1/2 milliseconds
     have passed)
8 milliseconds
   continue evaluating wait-functions until you get around to A's again, this time
     find that it is RUNNABLE.
10 milliseconds
   switch to A
500 microseconds
   A starts running
1 millisecond.
```

So the elapsed time in the old scheduler case is about 31 milliseconds.

In the old scheduler, with a compute-bound process D (at a lower priority than the other 3):

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C sets Z to the value of the microsecond-clock.
   Switch to scheduler stack group
500 microseconds
   evaluate wait-functions and find that B is now RUNNABLE, so choose it.
8 milliseconds
   switch to B
500 microseconds
   B starts running
1 millisecond
   B sets X to 'T
   Switch to scheduler stack-group
500 microseconds
   evaluate wait-functions and find that A is not RUNNABLE. (only 18 and 1/2 milliseconds
     have passed)
8 milliseconds
   continue evaluating wait-functions and find that D is runnable.
1 millisecond
   switch to D
500 microseconds
   D starts running
100 millisecond (the value of si:sequence-break-interval)
   SEQUENCE-BREAK forces a preemption
   Switch to scheduler stack-group
500 microseconds
   evaluate wait-functions and find that A is now runnable, so choose it
8 milliseconds
   switch to A
500 microseconds
   A starts running
1 millisecond.
```

So the elapsed time in the compute-bound old scheduler case is about 132 milliseconds.

If we convert it to block and wakeup, though:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C sets Z to the value of the microsecond-clock.
   Process C sets a timer to go off in 20 milliseconds and wakeup A
   Process C wakes up B
   Switch to B
1 millisecond
   B starts running
1 millisecond
   B sets X to 'T
   Process B wakes up A which is not runnable.
   switch to Null Process
500 microseconds
   Null Process starts running.
18 and 1/2 milliseconds
   Depending on the timer resolution, either the timer goes off, or the null-process
      evaluates A's wait-function, and sends a wakeup to A
   Switch to A
1 millisecond
   A starts running
1 millisecond.
```

Elapsed time is about 24 milliseconds.

If we convert it to block and wakeup, this is what happens when there is a background computation:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C sets Z to the value of the microsecond-clock.
   Process C sets a timer to go off in 20 milliseconds and wakeup A
   Process C wakes up B
   Switch to B
1 millisecond
   B starts running
1 millisecond
   B sets X to 'T
   Process B wakes up A which is not runnable.
   switch to D, which is runnable
500 microseconds
   D starts running.
between 8 and 24 milliseconds,  depending on the timer resolution
   Sequence-break (interrupt)
   Timer goes off
   Switch to timer process
1 millisecond
   find timer and execute it
      wakeup A
      switch to A
3 milliseconds
   A starts running
1 millisecond.
```

Elapsed time is between 17 and 33 milliseconds, so the background computation does not have much effect (other than in making us more sensitive to the timer resolution).

If we just run the old scheduler code in the new scheduler, leaving it as waiting processes rather than converting it to use block and wakeup, this is what happens:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C sets Z to the value of the microsecond-clock.
   Process C starts to wait.
   switch to Null Process
1 millisecond
   evaluate verify-functions and find that B is now runnable.
10 milliseconds
   switch to B
500 microseconds
   B starts running
1 millisecond
   B sets X to 'T
   Process B starts to wait.
   switch to Null Process
1 millisecond
   evaluate wait-functions and find that A is not runnable. (20 milliseconds have
      passed but we ignore that)
10 milliseconds
   evaluate verify-functions and find that A is now runnable.
10 milliseconds
   switch to A
500 microseconds
   A starts running
1 millisecond.
```

So the elapsed time in the new scheduler, where we naively wait, is around 36 milliseconds, which is not so bad.

However, if there are background computations, performance is much worse:

```
   Process C wakes up.
1 millisecond
   Process C sets Y to 'T
   Process C sets Z to the value of the microsecond-clock.
   Process C starts to wait.
   switch to Process D
1 millisecond
   Process D starts to run
70  milliseconds
   timer goes off, wait-function-poller starts to run, and determines that B is runnable
10 milliseconds
   switch to B
1 microseconds
   B starts running
1 millisecond
   B sets X to 'T
   Process B starts to wait.
   switch to D
1 millisecond
   Process D starts to run
70 milliseconds
   timer goes off, wait-function-poller starts to run, and determines that A is runnable
10 milliseconds
   switch to A
1 millisecond
   A starts running
1 millisecond.
```

Elapsed time was 167 milliseconds, which is much longer than 24, the elapsed time when you convert your code.

No Scheduler Stack Group

The process switching overhead in the old scheduler can be split into two parts: context switching and dispatching.

The switching overhead consisted of a stack-group switch to the scheduler stack group, and a stack group switch to the next process's stack group. A process switch can take anywhere from several hundred microseconds to milliseconds. Typical times were about 1/2 a millisecond for each of the two stack-group switches on the 3600.

The dispatching overhead consisted of evaluating the wait functions of processes until the scheduler found one that met all the criteria for running (that is, it was runnable and no higher priority processes were runnable). This overhead varied considerably depending on how many active processes were on your machine, and on the wait functions they had.

The new scheduler has very little dispatcher overhead at process switch time. It does not have any decisions to make at that point.

Additionally, getting rid of the scheduler stack group means that we can further reduce the context-switching cost of a process switch by a factor of two.

Since there is no scheduler stack group, simple-processes must run in their own (temporarily allocated) stack-group. (Note that one of the reasons for the existence of simple processes in the old scheduler was performance. Since simple processes were executed in the scheduler's stack group only one stack group switch was necessary when switching between simple and non-simple processes. No stack group switches were necessary when switching between two simple processes. In the new scheduler, even though we must allocate a stack-group for a simple process it still requires only one stack group switch to switch between simple and non-simple processes. It still requires no stack-group switches to switch between two simple processes that have run to completion.)

This means that simple-processes are allowed to call **process:process-wait**, **process:block-process**, and so on. This also means that they are not required to use **without-interrupts** as their sole means of synchronization. (See the section "Locks and Synchronization".)

Creating and Enabling Processes

The value of ***current-process*** is the process that is currently executing, or **nil** while the scheduler is running. When the scheduler calls a process's wait-function, it binds ***current-process*** to the process so that the wait-function can access its process.

There are two ways of creating a process. The primary way is to say simply, "call this function on these arguments in another process, and don't bother waiting for the result." In this case you never actually use the process as an object. The other way is to create a "permanent" process that you instantiate and manipulate as desired. In this latter case you use the **process:make-process** function. See the section "Using Processes for Computations".

Normally the function to be run should not do any input or output to the terminal. For a discussion of the issues: See the section "Input/Output in Stack Groups".

process:process-run-function implements the first way of creating a process. It allows you to call a function and have its execution happen asynchronously in another process. This can be used either as a simple way to start up a process that will run "forever", or as a way to make something happen without having to wait for it to complete.

process:process-run-function *name-or-kwds function* &rest *args* *Function*

Creates a process, presets it so it will apply *function* to *args*, and starts it running. *name-or-kwds* can be a string that becomes the process's name, or it can be a list of alternating keywords and values to which the corresponding process attributes are set.

The keywords are:

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

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

:restart-after-boot Applies to both warm and cold booting, if the cold boot occurs after a disk save with the process in it. If this is **t**, booting the machine restarts the process. If this is **nil**, booting the machine kills the process. The default is **nil**.

:warm-boot-action If this option is provided, its value controls what happens when the machine is warm booted. It overrides **:restart-after-boot**. If it is **nil** or not provided, the value of the **:restart-after-boot** option takes effect. For a description of the value of the warm-boot action:

See the section "Warm Boot Actions for Processes".

All other keywords are treated as arguments to **process:make-process**.

The function, **process:make-process** implements another way of creating a process. **process:make-process** creates a permanent Lisp object that can be manipulated by calling various functions.

process:make-process *name* &rest *init-args* &key *(:priority* **process:*default-process-priority***) *:initial-function :initial-function-arguments :verify-function :verify-function-arguments :flavor :run-reasons :area :simple-p :interrupt-handler :system-process :flags :top-level-whostate* &allow-other-keys *Function*

Creates and returns a process named *name* (a string). If the process is capable of running (it has an **:initial-function**, and, if it is **:simple-p**, a **:verify-function**)

then it is immediately RUNNABLE. You can override this default behavior by explicitly providing **:run-reasons nil**.

The *init-args* are alternating keywords and values that allow you to specify things about the process; however, no options are necessary. The following keywords are allowed:

:initial-function The computation to be executed by the process.

:initial-function-arguments
The arguments to the **:initial-function**.

:simple-p States that the top-level-whostate function of the process is of the form

```
(loop doing
      (apply process:block-process
         top-level-whostate verify-function verify-function-arguments)
      (apply initial-function initial-function-arguments)
```

Various optimizations are possible when a process is **:simple-p**. Unlike the old scheduler, there are no restrictions on code running inside a process that is **:simple-p**. The **:simple-p** optimization only pays off if **:initial-function** usually returns quickly. If you specify **:simple-p t**, you can then specify the following additional keywords:

:top-level-whostate Allows you to specify how the process is identified in the status line.

:verify-function The verify function to be used.

:verify-function-arguments
The arguments to the verify-function.

:flavor Specifies the flavor of process to be created. The **:flavor** keyword overrides the **:simple-p** keyword.

:stack-group The stack group the process is to use. If this option is not specified a stack group will be created according to the relevant options below.

:warm-boot-action What to do with the process when the machine is booted. See the section "Warm Boot Actions for Processes".

:priority The priority of the process. The default is :foreground 0. See the section "Priorities and Scheduler Policy".

:run-reasons Lets you supply an initial list of run reasons. The default is for a process to be runnable. You can override this by explicitly giving **:run-reasons nil**. For a regular process only **:initial-function** is necessary. For a simple-process both **:initial-function** and **:verify-function** are necessary.

:system-process **t** implies that this process is used to implement part of the operating system, and is not of interest to the user. See the section "Show Processes Command".

:interrupt-handler **t** if this process is referenced by an interrupt handler or sequence break. Necessary to set things up so that a reference to this process will not cause a transport trap or a page fault under any circumstances. This keyword should be used with care.

In addition, the options of **make-stack-group** are accepted. See the function **make-stack-group**.

If you specify **:flavor**, there can be additional options implemented by that flavor.

Other functions relating to creating and enabling processes:

process:preset *process function* &rest *args*
Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that it throws out of any current computation and start itself up by applying *function* to *args*. A **process-preset** call to a stopped process returns immediately, but does not activate the process, hence the process does not really apply *function* to *args* until it is activated later.

process:preset-simple-process *simple-process initial-function initial-args predicate predicate-args*
Handles the arguments required by a simple process so that it can be preset.

process:*default-process-priority*
The priority of a process if no priority is explicitly assigned.

process:process-p *thing*
Returns **t** if *thing* is a process.

process-simple-p *process*
Returns **t** if *thing* is a process.

process:map-over-all-processes *function* &rest *args*
Maps *function* over the set of all processes, including stopped ones.

process:map-over-active-processes *function* &rest *args*
Maps *function* over the set of active processes.

process:process-arrest-reasons *process*
Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

process:process-run-reasons *process*
Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

process:disable *process*
Stops *process* by revoking all its run reasons. Also revokes all its arrest reasons.

process:disable-arrest-reason *process* &optional *(reason* **':user**)
Removes *reason* from the process's arrest reasons. This can activate the process.

process:disable-run-reason *process* &optional *(reason* **':user**)
Removes *reason* from the process's run reasons. This can stop the process.

process:enable *process*
Activates *process* by revoking all its run and arrest reasons, then giving it a run reason of **:enable**.

process:enable-arrest-reason *process* &optional *(reason* **':user**)
Adds *reason* to the process's arrest reasons. This can stop the process.

process:enable-run-reason *process* &optional *(reason* **':user**)
Adds *reason* to the process's run reasons. This can activate the process.

For a discussion of choosing priority levels, see the section "Choosing Process Priority Levels".

Warm Boot Actions for Processes

The **:warm-boot-action** keyword to **process:make-process** sets the warm boot action for a process. It can also be set by the **setf** method for **process-warm-boot-action** of a **process:process**.

Whenever the system is booted, the warm-boot-action for each process is applied to the process. (Contrary to its name, the warm-boot-action applies to both cold or warm booting.) A warm boot action must either be a function or **nil**. If nil, **process:flush** is used, in which case the process will remain "flushed" until it is reset.

The default is **process:process-warm-boot-delayed-restart** which resets the process after initializations have been completed, causing it to start over at its initial function. You can also use **process:process-warm-boot-reset** which throws out of the process's computation and kills the process.

If you choose to write a function of your own, use the following guidelines:

- You must either reset or kill the process. The state of a stack group is indeterminate after a warm boot, and all processes must clear out their computations.

- If you intend to restart the process, you should wait until after initializations are complete. You do this by calling **process:process-warm-boot-delayed-restart** after performing whatever actions you want. **process:process-warm-boot-delayed-restart** sets up the process to be reset after initializations are complete.

process:process-warm-boot-delayed-restart *process*
Resets the process after initializations have been completed, causing it to start over at its initial function.

process:process-warm-boot-reset *process*
Throws out of the process's computation and kills the process.

process:process-warm-boot-restart *process*
Resets the process, causing it to start over at its initial function.

To find out the warm-boot-action of a process, use **process-warm-boot-action**.

process-warm-boot-action *process*
Returns *process*'s warm-boot-action, which controls what happens to this process if the machine is booted. (Note: Contrary to the name, this applies to both cold and warm booting.)

Getting Information About a Process

These functions return information about a process:

process:active-p *process*
Returns **t** if *process* is active.

process:runnable-p *process*
Returns **t** if *process* is runnable.

process:process-p *thing*
Returns **t** if *thing* is a process.

process-simple-p *process*
Returns **t** if *thing* is a process.

process-name *process*
Returns the name of *process*, which was the first argument to **process:make-process** or **process:process-run-function** when the process was created.

process-initial-form *process*
Returns the initial "form" of *process*. To change the initial form, call the **process:preset** function.

process-warm-boot-action *process*
Returns *process*'s warm-boot-action, which controls what happens to this process if the machine is booted. (Note: Contrary to the name, this applies to both cold and warm booting.)

process:process-run-reasons *process*
Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

process:process-arrest-reasons *process*
Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

process:process-base-process-priority *process*

process:process-process-priority *process*

process:process-run-time *process*
Returns the amount of run time the process has accumulated, in microseconds. It is accurate for the current process.

process:process-run-time-low *process*
Returns the low 32 bits of **process:process-run-time**, in microseconds. It is not up to date for the current process.

process:process-disk-wait-time *process*
Returns the amount of time the process has spent waiting for the disk (that is, paging), in microseconds. It is accurate to 10E24 microseconds, quantizable.

process:process-disk-wait-time-milliseconds *process*
Returns the amount of time the process has spent waiting for the disk (that is, paging), in milliseconds, accurate to 10E24 microseconds. It is quantizable.

process:process-cpu-time *process*
Returns the amount of cpu time the process has received, in microseconds. It is accurate to 10E24 microseconds, quantizable.

process:process-idle-time *process*
Returns the amount of time since the process ran last, in sixtieths of a second.

process:process-last-time-run *process*
Returns the last time the process ran, as a universal time.

process:process-page-fault-count *process*
Returns the number of page faults the process has taken.

process:process-runnable-time *process*
Returns the amount of time the process has been runnable, in microseconds.

Blocking, Waiting, and Waking Processes

For a discussion of the difference between blocking and waiting, see the section "Blocking Vs. Waiting".

sleep *n-seconds* &key *(:sleep-reason* **"sleep"***)*
Waits for *n-seconds* and then returns.

process:block-process *whostate verify-function* &rest *args*
Causes the process to block. This assumes that another process (or interrupt handler) is going to wakeup the process when its verify function becomes true.

process:block-and-poll-wait-function *whostate interval verify-function* &rest *args*
Causes the process to wait. The scheduler polls it periodically to see if its verify function has become true.

process:allow-preemption
Allows the scheduler to recompute the internal dispatcher priority of this process. If there are processes that now have higher priority because of this priority reduction, they will preempt the current process and run. This allows preemption even when preemption is disabled.

process:wakeup *process*
Evaluates the verify function of *process*.

process:force-wakeup *process*
Wakes up *process*.

process:wakeup-without-test *process*
Wakes up *process* without testing its verify function. **process:block-process** then tests the verify function and returns to its caller only if the verify function returns **t**. If it returns **nil**, *process* goes blocked again.

process:block-with-timeout *timeout whostate verify-function* &rest *args*
Specifies a time interval (in seconds) after which the function just returns.

process:block-and-poll-with-timeout *n-seconds whostate interval function* &rest *arguments*
Specifies a time interval (in seconds) after which *function* just returns.

process:process-wait *whostate function* &rest *arguments*
Waits until the application of *function* to *arguments* returns non-**nil** (at which time **process:process-wait** returns).

process:process-wait-with-timeout *whostate time function* &rest *args*
Specifies a time period (in seconds) after which *function* should just return.

process:safe-to-process-wait-p *process*
Returns **t** if it is safe to call **process:process-wait** on *process*.

process:*process-wait-interval*
The length of time to wait, in fractions of a second.

process:with-process-block-timeout *(timeout* &optional *timer-name)* &body *body*
Specifies a timer and a time interval (in seconds) after which the function just returns.

process:wait-forever &optional *(whostate* **"wait forever"***)*
Causes the current process to wait forever.

process:with-wait-function-polling *(interval function* &rest *args)* &body *body*

process::poll-simple-process *simple-process* &optional *(interval* **process:*process-wait-interval****)** *force*
Polls a simple-process if it is not running.

Resetting, Interrupting, and Aborting Processes

process:abort &rest *args*
Exits a process by signalling **sys:abort**, but respecting processes that are in the dynamic extent of a **sys:without-aborts** form.

process:interrupt *process function* &rest *args*
Tells *process* stop its computation and to execute some function on your behalf in its environment. If the function returns normally the interrupted computation is resumed. This should be used with care, it can be dangerous.

process:reset *process* &key *(:if-current-process* **t***) (:if-without-aborts* **:ask***)* *Function*

Exits the entire computation of *process* and forces it to call its initial function again. Resetting a process clears any waiting condition, and so if it is active it becomes runnable.

:if-without-aborts indicates what to do if the process is not currently resettable. See the function **sys:without-aborts**. It takes the values **:force**, **:ask**, and **nil**.

:force	Process is reset anyway.
:ask	Queries the user as to whether to force the process to reset or not.
nil	Return a list of reasons why the process cannot be reset.

process:reset-and-release-resources *process* &key *(:if-current-process* **t***) (:if-without-aborts* **:ask***)*
Resets *process*, but does not start it up again. It releases most resources used by the process, arrests the process, and makes it FROZEN. If the process is re-enabled, a new set of resources are allocated for it.

process:kill *process* &key *(:if-current-process* **t***) (:if-without-aborts* **:ask***)*
Resets *process* and releases its resources, making the process DEAD, that is inaccessible to **process:map-over-all-processes**. Reset makes the process ALIVE again. It is not recommended.

process:flush *process*

Forces a process to be blocked. The state of its computation is not changed and it continues to hold on to all of its resources. The process will not proceed until it is reset.

Scheduler CP Commands

Debug Process Command

Debug Process *process*

Enters the Debugger to look at *process*.

process — A process. You can press HELP for a list of all the processes currently running in your environment. See the section "Show Processes Command".

Restart Process Command

Restart Process *process*

Causes the process to start over in its initialized state. This is one way to get out of stuck states when other commands do not work.

process — A process. You can press HELP for a list of all the processes currently running in your environment. See the section "Show Processes Command".

Start Process Command

Start Process *process*

Starts a process that has been halted with Halt Process.

process — A process. You can press HELP or use the Show Processes command for a list of all the processes currently running in your environment.

Halt Process Command

Halt Process *process*

Causes *process* to stop immediately. This is the same as [Arrest] in the Peek processes menu.

process A process. You can press HELP for a list of all the processes currently running in your environment. (See the section "Show Processes Command".)

Kill Process Command

Kill Process *process*

Causes *process* to go away completely.

process A process. You can press HELP for a list of all the processes currently running in your environment. See the section "Show Processes Command".

Show Processes Command

Show Processes *keywords*

Displays all the processes currently in your environment. See Figure ! .

Keywords :Active, :Idle, :More Processing, :Name, :Order, :Output Destination, :Priority Above, :Priority Below, :Recent, :State, :System, :Unarrested

:Active {*time-interval*} Shows only processes that have been active within *time-interval*. The mentioned default is "1 minute". (Obsolete, use :Recent.)

:Idle {*time-interval*} Shows only processes that have been idle for at least *time-interval*. The mentioned default is "1 minute".

: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").

:Name {*process-names-or-substrings*} Enables 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.

:Order {Idle, Name, None, Percent} Sorting method for the processes display. The default is None.

: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***.

:Priority Above {*integer*} Shows only processes of priority higher than *integer*.

:Priority Below {*integer*} Shows only processes of priority lower than *integer*.

:Recent {*time-interval*} Shows only processes that have run within *time-interval*.

:State {*string*} Shows only processes whose state contain *string*.

:System {Yes, No} Shows the system processes also. The default is No, the mentioned default is Yes.

:Unarrested {Yes, No} Shows only processes that are not arrested. The default is No, the mentioned default is Yes.

```
Command: Show Processes (keywords) :Priority Above (an integer) 5
Process Name           State                Priority  Idle      % utilization
Chaos Background       Chaos Background      F:8      1 sec     0.2%
Mouse                  Mouse                PF:31               0.5%
Timer                  Timer Process        PD:500    forever   0.0%
Timer                  Timer Process        PD:500    2 hr      0.0%
Timer                  Timer Process        PD:500    forever   0.0%
Timer                  Timer Process        PD:500    forever   0.0%
TCP Background         TCP Background        F:8      57 sec    0.0%
Timer                  Timer Process        PD:500    forever   0.0%
Timer                  Timer Process        PD:500    1 hr      0.0%
Process Scheduler      Scheduler Wait       PD:100000           1.2%
Wait function poller   Wait function poller  F:110              1.0%
Keyboard               Keyboard             PF:30     3 sec     0.6%
3600 Ethernet Receiver Ethernet Packet      PF:8                1.8%
Timer                  Timer Process        PD:500              2.3%
Update Status Line     SIMPLE-PROCESS wait   D:500000           1.9%
Timer                  Timer Process        PD:500    9 min     0.0%
Timer                  Timer Process        PD:500    37 sec    0.0%
Timer                  Timer Process        PD:500              1.9%
```

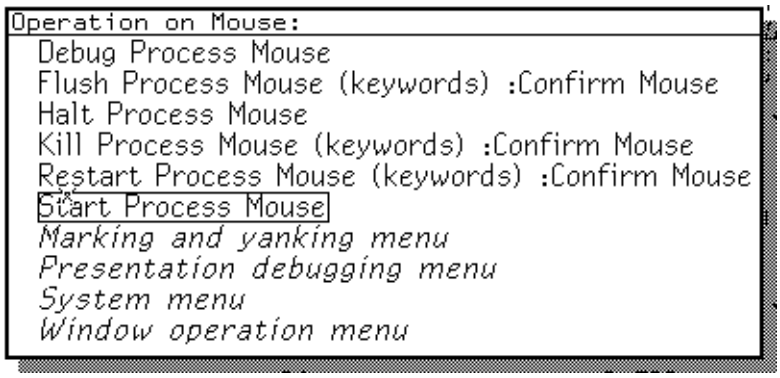
Figure 31. Show Processes

The priorities are:

F Foreground

B	Background
D	Deadline
P	Preemptive

The process names in the display are mouse sensitive. Clicking Middle on one of them puts that process in the debugger. Clicking Right pops up a menu of operations:



Set Process Priority Command

Set Process Priority *process value*

Adjusts the priority of *process*.

process	A process. You can press HELP or use the Show Processes command for a list of all the processes currently running in your environment.
value	{*number*} For Foreground processes only. The priority value to give the process. Usually values range from -1 to 30, with most normal processes having priority 0. The mouse and keyboard process usually have priority 30 and some network processes have priority 10. The garbage collector and notifications have priority 5. Use Show Processes to see all your current processes with their priorities. It is the relationship among the priorities that is important. The mouse and keyboard have a high priority so that user input is recognized and handled rapidly. Notifications should not take priority over keyboard or mouse input, but they should happen in a timely fashion. See the section "Choosing Process Priority Levels".

Timers

About Timers

The Timer Facility implements *timer*s, a mechanism that allows arbitrary code to execute, asynchronously, at some time in the future.

In an event-driven scheduler timers are of critical importance. In a polling scheduler "timers" are often implicit in the wait-function. In an event-driven scheduler a blocked process has nothing periodically polling its wait-function, (which would just beg the question, of course, of how we implemented the periodic polling without timers) and therefore, if it needs to perform a time dependent action (from simply timing out, to executing arbitrary code) some external timer facility needs to wake it up.

The timer facility manages the external timer(s).

If a timer is waiting to go off, it is considered *pending*. When it goes off it is *executing*. Each timer executes in its own process at the priority of the timer.

Each pending timer has a function, a priority, and an expiration time.

You can create, reset (set the expiration time), and clear timers.

The principal function for dealing with timers is **process:reset-timer-relative**, which takes its arguments in seconds. **process:reset-timer-relative-timer-units** takes its arguments in **process:*timer-units***, which is generally microseconds.

Timers are accurate to within **process:*timer-resolution***. This means that if you request a timer to go off at time N, you can expect it to go off at any time within plus or minus **process:*timer-resolution*** **process:*timer-units*** of N.

This resolution is not a guarantee, but a statement of probability. You can change **process:*timer-resolution*** by using the **:timer-resolution** keyword to **process:set-scheduler-parameters**. You cannot set **process:*timer-resolution*** to be less than the value of **process:*minimum-timer-resolution*** on any given machine architecture.

The third unit that timers take are universal times. If you set a timer to go off at an absolute time by **process:reset-timer-absolute**, or if you expect it to survive a warm-boot, you must specify the time in universal time. Relative times cannot be expected to be meaningful across boots.

Creating Timers

There are two functions for creating timers.

process:create-timer-call *function args* &key *(:name* **"timer"***) (:priority* **process:*default-timer-entry-priority***)

process:create-timer-wakeup &optional *(process* ***current-process****)* &key *(:name* **"timer wakeup"***) :force-p*

A common use of **process:create-timer-call** is to wake up a process, using **process:wakeup**. The timer facility can optimize this case if you use **process::create-timer-wakup**.

Setting and Clearing Timers

process:clear-timer *timer*
Clears a timer, so that it does not go off. If it has not gone off yet, **process:timer-expired-p** continues to return **nil**.

process:reset-timer-relative *timer delta-t*
Resets the timer relative to its initial setting, using seconds.

process:reset-timer-relative-timer-units *timer delta-t*
Resets the timer relative to its initial setting, using timer-units.

process:reset-timer-absolute *timer universal-time*
Resets timer to an absolute time, using universal-time.

process:reset-timer-absolute-timer-units *timer clock-time*
Resets the timer to an absolute time.

Getting Information About Timers

process:timer-expiration-time *timer*
Returns the time the timer should expire.

process:timer-function *timer*

process:timer-name *timer*
Returns the name of the timer as a string.

process:timer-priority *timer*

process:timer-universal-time *timer*

process:timer-universal-time-of-expiration *timer*

process:*timer-resolution*
Determines the accuracy of timers.

process:*minimum-timer-resolution*

process:*timer-units*
The number of seconds per timer unit.

Timer Predicates

process:timer-expired-p *timer*
Returns **t** if the timer has actually fired and has not been reset.

process:timer-expires-by *timer relative-time*
Returns the amount of time until the timer expires, in seconds.

process:timer-expires-by-relative-timer-units *timer relative-time*

Returns the amount of time until the timer expires, in timer units.

process:timer-pending-p *timer*

Returns **t** if the timer is still set to go off, but has not done so yet.

process:timer-should-have-expired-p *timer*

Returns **t** if the timer should have expired, regardless of its state. **process:clear-timer** has no effect on this predicate. When you are simply using a timer as an argument to a predicate, and not as a trigger for some asynchronous action, the correct predicate to use is **process:timer-should-have-expired-p**.

Timer Warm Boot Actions

process:set-timer-warm-boot-action *timer warm-boot-action*

process:timer-warm-boot-action *timer*

Locks and Synchronization

When two or more processes share a data structure some form of synchronization is necessary.

The general topic of synchronization is covered in many books on operating systems, and is beyond the scope of this documentation.

There are four main synchronization mechanisms provided by Genera: atomic operations, locks, disabling preemption, and, on Ivory based systems only, raising the trap mode.

Atomic Operations

Genera provides a set of Lisp operations that are atomic with respect to multiple processes and interrupts. These functions are all built out of the more primitive operator, store-conditional.

store-conditional *pointer old new*

Checks to see whether the cell contains *old*, and, if so, it stores *new* into the cell. The test and the set are done as a single atomic operation.

These operations are all atomic and therefore guaranteed race free. Even with many processes performing the same operations, no operation is lost or duplicated.

process:atomic-incf *reference* &optional *(delta* **1***)*

Atomically increments (**incf**) *reference* by *delta*, side effecting *reference*, and returning the new value of *reference*. This works in the presence of multiple processes all trying to increment and decrement *reference*.

process:atomic-decf *reference* &optional *(delta* **1***)*

Atomically decrements (**decf**) *reference* by *delta*, side effecting *reference*, and returning the new value of *reference*. This works in the presence of multiple processes all trying to increment and decrement *reference*.

process:atomic-push *item reference* &key *:area*

Atomically pushs *item* onto *reference*. This operation works in the presence of multiple processes all trying to do **push** and **pop** of values on and off *reference*.

process:atomic-pop *reference*

Atomically pops the first item off of *reference*, side-effecting *reference* and returning the new value. This operation works in the presence of multiple processes all trying to do **push** and **pop** of values on and off *reference*.

process:atomic-replacef *reference new-value*

Atomically sets *reference* to *new-value*. This works in the presence of multiple processes all trying to modify *reference*.

process:atomic-updatef *variable function*

Atomically updates *variable* by the combinational function *function*. This works in the presence of multiple processes all trying to modify *variable*. This guarantees that the transition of *variable* from *value* to **(funcall** *function value***)** is atomic.

Locking

A *lock* is a software construct used for synchronization of two processes. On Symbolics computers, the software construct for a lock is a Lisp object. A lock protects some resource or data structure so that only one process at a time can use it. A lock is either held by some process, or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing some other process to seize it. If a process, *P1*, holds a lock, and another process, *P2*, tries to acquire that lock, then P2 *promotes* P1. This means that if P2's priority is higher than P1's, P1 has its priority boosted to that of P2. See the section "Promotion".

Creating Locks

Locks are created with the function **process:make-lock**.

process:make-lock *name* &key *(:type* **:simple***) :recursive :area :flavor* *Function*

Creates a lock.

:name	A string, the name of the lock.
:type	The kind of lock. The possibilities are **:simple** (the default), **:multiple-reader-single-writer**, and **:null**.
:recursive	If **:recursive** is **t**, then the lock can be locked recursively by the process holding the lock. The default is **nil**.
:area	The area in which to put the lock object. This is primarily useful if you are concerned with keeping your working set small. The value specifies the area. It should be either an area number (an integer), or **nil** to mean the default area. This argument defaults to **nil**.
:flavor	The flavor of lock to create. Normally, the value is computed automatically. This keyword is needed only if you want to use some special flavor.

There are three types of locks, **:simple**, **:multiple-reader-single-writer**, and **:null**. A **:simple** lock is a mutual exclusion lock; only one process can hold it at a time and all other processes must wait until it is released. A **:multiple-reader-single-writer** permits any number of processes to access its resource as readers but only one process can hold the lock for writing. When a process attempts to seize a **:multiple-reader-single-writer** lock, it must specify if it wants to be a reader or a writer. A **:null** lock is one that does not actually lock anything, it is intended primarily for debugging. Any of these can be **:recursive**, meaning that the holder of the lock can relock it any number of times. The default lock created by **process:make-lock** is a simple, non-recursive lock.

```
(process:make-lock "my-lock")
#<PROCESS::SIMPLE-NONRECURSIVE-NORMAL-LOCK my-lock 47107526>
```

process:reset-lock *lock* *Function*

Takes a lock object and resets the lock to its initial state.

Locking and Unlocking Locks

The usual way to use a lock is with **process:with-lock**.

process:with-lock *(lock* &key *:mode* &allow-other-keys*)* &body *body* *Macro*

Seizes *lock* and executes *body* while it holds the lock.

:mode is only used for reader-writer locks. It can be **:read**, **:write**, or **:exclusive-read**. **:exclusive-read** is for **:multiple-reader-single-writer** locks and locks the lock for a read but does not allow any other readers.

process:with-lock uses the lock and the **:mode** argument to construct a call to **process:make-lock-argument**, then calls **process:lock** with the lock argument, runs *body*, and finally calls **process:unlock** to release the lock.

If you need greater control over the execution of *body*, you can call **process:make-lock-argument**, **process:lock**, and **process:unlock** explicitly.

process:make-lock-argument *lock* &rest *keys*
Constructs an argument list for **process:lock** and **process:unlock**, including the lock object to seize and the mode.

process:lock *lock lock-argument*
Locks a lock.

process:unlock *lock lock-argument*
Releases a lock.

If you need to do something explicitly without a lock, while you are holding a lock, you can use **process:without-lock**.

process:without-lock *(lock)* &body *body* *Macro*

Unlocks any lockings that were done by the current process, runs *body*, and then relocks the lock.

Getting Information About a Lock

process:lock-idle-p *lock*
Returns **t** if no one is holding *lock* in any way.

process:lock-lockable-p *lock* &rest *keys*
Returns **t** if the current process could lock the lock in the specified mode without waiting.

process:lock-name *lock*
Takes a lock object and returns the name of the lock as a string.

process:lockp *lock* Returns **t** if its argument is a lock object.

Locks on 3600-Family Computers

The material in this section is obsolete and is retained only for compatibility. New code should use **process:make-lock** and **process:with-lock**.

On 3600-family machines, a lock is a locative pointer to a cell. If the lock is free, the cell contains **nil**; otherwise it contains the process that holds the lock. The **process-lock** and **process-unlock** functions are written in such a way as to guarantee that two processes can never both think that they hold a certain lock; only one process can ever hold a lock at a time.

process-lock *locative-pointer* &optional *lock-value* (*whostate* "**Lock**") *interlock-function* *Function*

This function is obsolete, use **process:lock** in new code.

Seizes the lock to which *locative-pointer* points. If necessary, **process-lock** waits until the lock becomes free. When **process-lock** returns, the lock has been seized. *lock-value* is the object to store into the cell specified by *locative-pointer*, and *whostate* is passed on to **process-wait** if the process must wait. If *lock-value* is **nil** or unsupplied, the value of **sys:current-process** is used.

The argument *interlock-function* must be **nil** or a function of zero arguments. **process-lock** guarantees to call *interlock-function* if the lock is successfully changed to be locked, and not otherwise. It can therefore be used to implement atomic unwind-protect of locking.

This is atomic and protected from unlocking locks held at a higher level. In this case, "atomic" means that the operation cannot be decomposed into smaller operations. If an operation is atomic, then `c-ABORT` and other interrupts cannot occur in the middle of it.

Here is an example.

```
(let ((locked nil))
    (unwind-protect
      (progn
        (process-lock lock si:current-process "Lock"
          #'(lambda () (setq locked t)))
        ... body of locked code ...
        )
      (when locked (process-unlock lock si:current-process))))
```

Note that this example protects against aborting while your process waits for the lock.

In the example, a program is designed to lock a lock. It wants to use **unwind-protect**, when it exits, if and only if the lock was locked. Therefore, the program needs to maintain a flag that indicates if the lock has indeed been locked. Changing the state of the lock and the flag must happen together. If they occur asynchronously, errors ensue. Thus the sequence of changing the lock and setting the flag must be atomic.

For simple sets of operations, **without-interrupts** gives atomicity. However, you cannot call **process-wait** while in a **without-interrupts**, and locking a lock calls **process-wait**. This is why **process-lock** supplies this argument and guarantees that calling the function and setting the lock to be locked will be atomic.

process-unlock *locative-pointer* &optional *lock-value error-p* *Function*

This function is obsolete. Use **process:unlock** in new code.

Unlocks the lock to which *locative-pointer* points. If the lock is free or was locked by some other process, an error is signalled if *error-p* is **t**. Otherwise the lock is

unlocked. If *lock-value* is **nil** or unsupplied, the value of **sys:current-process** is used.

It is a good idea to use **unwind-protect** to make sure that you unlock any lock that you seize. For example, if you write:

```
(unwind-protect
   (progn (process-lock lock-3)
          (function-1)
          (function-2))
   (process-unlock lock-3))
```

then even if **function-1** or **function-2** does a **throw**, **lock-3** is unlocked correctly. Note that if you use this example, your system enters the debugger when the cleanup handler attempts to unlock the lock, claiming it is not locked. Particular programs that use locks often define special forms that package up this **unwind-protect** into a convenient stylistic device.

process-lock and **process-unlock** are written in terms of a subprimitive function called **store-conditional**, which is sometimes useful in its own right.

You can also use **si:make-process-queue** and related functions to set up a queue for processes waiting to seize a lock. Each process on the queue is given a chance to seize the lock in the order in which it requests the lock.

si:make-process-queue *name size* *Function*

Makes and returns a queue for processes requesting a lock. *name* is an external name for the queue and is used only in printing the queue. *size* is the size of the queue. This is the maximum number of processes that will be guaranteed to lock the queue in exact requesting order.

si:process-enqueue *queue* &optional *queue-value* (*whostate* "**Lock**") *Function*

Locks *queue*. *queue-value* is an object to enter on the queue; if *queue-value* is **nil** or unsupplied, the object is the current process. If *queue* is empty, **si:process-enqueue** seizes the lock immediately by inserting *queue-value* on the queue and returning. If *queue* is not full but other processes are on the queue waiting for the lock to be free, it inserts *queue-value* at the end of the queue, waits for the lock to be free, and then seizes the lock by returning. If *queue* is full, it waits until *queue* is not full and tries again to seize the lock. *whostate* is displayed in the status line while waiting to seize the lock. **si:process-enqueue** signals an error if *queue-value* has already seized the lock.

si:process-dequeue *queue* &optional *queue-value* (*error-p* **t**) *Function*

Unlocks *queue*. *queue-value* is an object on the queue. If *queue-value* is **nil** or unsupplied, it is the current process; if not **nil**, it should be the same as the *queue-value* given to the matching call to **si:process-enqueue**. If *queue-value* has the lock, unlocks the lock by removing *queue-value* from *queue* and giving the next pro-

cess on the queue a chance to seize the lock. If *queue-value* does not have the lock and *error-p* is not **nil**, signals an error.

si:process-queue-locker *queue* *Function*

Returns the *queue-value* for the process that holds the lock on *queue*, or **nil** if the lock is free.

si:reset-process-queue *queue* *Function*

Unlocks *queue* and removes all processes on the queue.

Disabling Preemption

One way of synchronizing a data structure that is accessed by multiple-processes is to inhibit scheduling when you manipulate that structure.

This has the advantage that readers (clients that do not modify the data structure) do not have to pay any price for synchronization, since all the work is done by the writers. It is also simpler — there are no deadlocks.

The disadvantage is that it shuts out all other processes. Usually it is a more drastic measure than is necessary. Disabling preemption is discouraged as a technique for synchronizing data structures. Use an alternative method if possible.

See the section "Atomic Operations". See the section "Locking".

However, there will still occasionally be some need for inhibiting preemption, or stopping other processes from running. Also, for compatibility, we support old code that still uses **without-interrupts**.

The name *without-interrupts* is misleading, since interrupts are not disabled within the body of that form. It might be more accurately called *without-preemption*, since interrupts occur, scheduling is legal (if you voluntarily yield the processor), and only preemption is inhibited.

Because of these problems with the name, and because we wish to encourage conversion of old code that used **without-interrupts** to use locks or atomic operations, **without-interrupts** will be declared obsolete in a future release, and generate style warnings.

without-interrupts &body *forms*
: Evaluates *forms* with **sys:inhibit-scheduling-flag** bound to **t**. In other words, the body is an *atomic operation* with respect to process scheduling.

If it is inappropriate to convert to locks or atomic operations, four forms are provided that disable preemption in subtly different ways.

process:with-no-other-processes &body *body*
: Evaluates its *body* forms without being preempted by any other

processes. On a multiprocessor, all processes running on any other processors are stopped.

process:with-preemption-disabled &body *forms*
Evaluates its *body* forms without being preempted by any other processes. On a multiprocessor, all processes running on any other processors continue running, but they cannot yield the processor or be preempted.

process:without-preemption &body *forms*
Evaluates its *body* forms without being preempted by any other processes. On a multiprocessor this has no effect on any processor other than the processor it is executed on.

It is easiest to describe these in the context of a multiprocessor, since some primitives behave identically on a single processor, yet have different intents. Behavioral differences show up only in the multiprocessor case, so we use the multiple processor case to illustrate the differences.

without-interrupts assumes that no other processes will run anywhere in shared memory. This restriction includes processes currently running on other processors that reference our memory. **process:with-no-other-processes** has this meaning and should be used instead of **without-interrupts**. Explicitly calling the scheduler (with **process:block-process** or **process:process-allow-preemption**, for example) is not allowed. It is important to note that this does not disable interrupts (sequence-breaks).

process:with-preemption-disabled does not allow preemption on any processor. It allows processes that are currently running to continue running, but, if they try to enter the scheduler, they must wait until preemption is enabled again. If the process on the machine that entered the **process:with-preemption-disabled** enters the scheduler, it does switch processes, and after the switch is over the new current-process can re-enable preemption. You can wakeup processes during a **process:with-preemption-disabled**, as long as the scheduler data-structures are not locked.

process:with-preemption-disabled is useful, for example, when you are holding a lock on a data structure, and you wake up a higher priority process that is waiting for the same lock. You want to atomically wake up this process, and simultaneously release the lock. If you just wake it up, without preemption disabled, it preempts you, and goes blocked again. Or, if you release the lock, and then wake up the process, another process can sneak in, in between.

process:without-preemption is mainly a performance optimization. It means "no preemption is allowed on this processor". It has no effect on the scheduler or processes on other processors. It just locks the current-process down to its current-processor.

Trap Mode and Synchronization

For Ivory-based systems only:

Preemption only occurs in emulator mode. Therefore, no extra synchronization is needed between code running in trap mode greater than emulator and code that might execute in other processes.

Please see the documentation on trap-mode.

Promotion Protocol

If a process, p1, is forced to wait for another process, p2 (perhaps because p2 holds a resource that is critical to p1), p1 might want to promote p2's priority. This promotion is done so that p1 can retrieve the resource from p2 in a timely fashion.

The locking facility provided by Genera automatically does promotion.

A process can promote another process by following the promotion protocol. This facility is rarely necessary and should be used with caution.

process:add-promotion *p1 uid1 p2 uid2*
Promotes the priority of a process. Process *p1* with *uid1* is promoting process *p2* with *uid2*. When *p2* finishes *uid2*, or *p1* quits *uid1*, this promotion is removed (by **process:finish-promotions** or **process:remove-promotions** respectively.)

If *p2* had a lower priority than *p1*, and *p2* were not promoted, a compute-bound process (named, say, p3) with a priority between that of *p1* and that of *p2* could conceivably starve out *p1*. Therefore it is appropriate that *p2* adopt the priority of *p1* until it allows *p1* to proceed.

Even if *p1* has a lower priority than *p2* it must call **process:add-promotion**.

Priorities change. For example, if *p1* is itself promoted, we would want the promotion to recurse to *p2*.

process:promotion-block *scheduler-queue whostate verify-function* &rest *args*

process:remove-promotions *p1 uid1*
Notes that *p1* has finished, or quit, the activity identified by *uid1*. Removes all promotions done for *p1* at *uid1*.

process:finish-promotions *p2 uid2*
Notes that *p2* has finished the activity identified by *uid2*. Removes all promotions done to *p2* at *uid2*.

Priorities and Scheduler Policy

There are three classes of scheduler priorities used by the dispatcher:

DEADLINE	The earliest deadline in microseconds is the highest priority.
FOREGROUND	A strict ordering by positive numbers. Highest number is highest priority.
BACKGROUND	Round robin.

DEADLINE are higher priority than foreground priorities, which, in turn, are higher than background.

If there are no runnable processes, the dispatcher switches to the NULL process.

Priorities are created by **process:make-process-priority**.

process:make-process-priority *class priority* &rest *extra-args* *Function*

Creates a process priority. The priority argument depends on *class*.

Class	*Priority*
:deadline	A relative deadline in microseconds. It is the deadline by which the process must exit the most deeply nested **process:with-process-priority**.
:foreground	An integer. The higher the integer, the higher the priority.
:background	Priority is ignored.

Both **:deadline** and **:foreground** priorities take an *extra-arg* of **:preemptive**

A programmer can specify priorities for a process by using **process:set-process-priority**, or **process:with-process-priority**.

process:with-process-priorities *(priority* &rest *priorities)* &body *body* *Function*

Defines a priority *region* — *body* will be executed with priority *new-priority*. Since the extent of *new-priority* is bounded, scheduler policy is often more lenient with priority regions than with simply setting the priority. The policy is controlled by the policy hook **process:*policy-hook-region-priority***. See the section "Extensible Scheduler".

Additionally, **process:with-process-priorities** takes a sequence of *time priority* pairs that define how the priority of the process will change over time while executing this form.

The times are expressed in seconds, and are relative to the last time priority pair.

If it were important to execute a certain form within 1 second, but you didn't want to otherwise interfere with other processes, you could write code like this:

```
(process:with-process-priorities
  ((process:make-process-priority :foreground -2)
   .5 (process:make-process-priority :foreground -1)
   .1 (process:make-process-priority :foreground 0)
   .1 (process:make-process-priority :foreground 1)
   .1 (process:make-process-priority :foreground 2)
   .1 (process:make-process-priority :deadline 100000.))
  form)
```

This says: start executing at priority -2. If, after half a second has elapsed, you haven't completed executing *form* then raise your priority to -1. If, after .6 seconds you still haven't completed executing *form* then raise your priority to 0. Continue to do this until .9 seconds have elapsed, and then panic and use a deadline.

Or, alternatively, if there were a form that you expect to be very quick, you might want to play a different game with the priorities. At first, it is reasonable to give it a high priority. If it takes too long to complete, you don't want to degrade the performance of the reset of the machine, so you gradually decrease its priority.

```
(process:with-process-priorities
  ((process:make-process-priority :deadline 1000000.)
   .1 (process:make-process-priority :foreground 2)
   .1 (process:make-process-priority :foreground 0))
  form)
```

For the first 10th of a second the form has a deadline of 1 second. If we haven't finished by a 10th of a second, we are tying up the machine and responsiveness isn't as important as we thought at first, so we lower our priority to 2. If after two tenths of a second we haven't finished, chalk this execution up as a loss, and let it complete computing at normal priority, so we don't selfishly use up most of the machine.

This is the mechanism that **si:with-process-non-interactive-priority** uses to allow short functions to execute at interactive priority, but longer ones to revert to the default compute-bound priority.

```
(defun test-process-priority-changes (time)
  (let ((end-time (sys:%32-bit-plus
                   (sys:%microsecond-clock) time))
        (priority
          (process:process-process-priority *current-process*))
        (results nil))
    (flet ((record-priority ()
             (push (cons (sys:%microsecond-clock) priority) results)))
      (record-priority)
      (process:with-process-priorities
        ((process:make-process-priority :foreground 2)
         .166666 (process:make-process-priority :foreground 3)
         .333333 (process:make-process-priority :foreground 5)
         .500000 (process:make-process-priority :deadline 500000.))
        (loop while (time-lessp (sys:%microsecond-clock) end-time) do
          (let ((new-priority
                  (process:process-process-priority *current-process*)))
            (when (neq priority new-priority)
              (setf priority new-priority)
              (record-priority)))))
      (setf priority
            (process:process-process-priority *current-process*))
      (record-priority))
    results))
```

```
(defun show-delayed-priorities (results)
  (let* ((results (reverse results))
         (initial-time (caar results)))
    (flet
      ((print-priority (s class priority)
         (select class
           (process:*process-priority-class-idle-time*
            (format s "NA"))
           (process::*process-priority-class-background*
            (format s "Background"))
           ((process:*process-priority-class-interactive*
              process:*process-priority-class-foreground*)
            (format s "~A:~D"
                    (if
                      (eql class process:*process-priority-class-interactive*)
                      "I" "F")
                    (process::back-convert-foreground-priority priority)))
           (process::*process-priority-class-deadline*
            (format s "D:~D" priority))
           (otherwise
            (format s "~D:~D" class priority)))))
      (loop for (time . priority) in results do
        (format t "~&~6D: Priority: "
                (sys:%32-bit-difference time initial-time))
        (print-priority *standard-output*
                        (process:scheduler-priority-class priority)
                        (process:scheduler-priority-priority priority))))))
```

```
Command: (test-process-priority-changes 2000000.)
((-989859200 . -822083584) (-990850110 . -2146983648) (-991367439 . -832569344)
 (-991700866 . -828375040) (-991891470 . -826277888) (-991892886 . -824180736))
Command: (show-delayed-priorities *)
     0: Priority: F:1
  1416: Priority: F:2
192020: Priority: F:3
525447: Priority: F:5
1042776: Priority: D:500000
2033686: Priority: F:0
NIL
```

process:get-instantaneous-priority translates between the programmer priorities set up by **process:set-process-priority** and **process:with-process-priority** and the priorities the dispatcher uses.

process:set-process-priority *process new-priority*

process:set-process-base-priority *process new-priority*

The *Priority manager* is a process that periodically re-orders the scheduler priorities of all processes whose programmer priorities translate into scheduler priorities of class **:foreground**.

The Priority manager must arrange this order so that, when viewed over a sufficiently long interval, the processes behave as specified by their programmer priority.

Other priority functions:

process:set-scheduler-parameters &key *(:wakeup* **process::*scheduler-wakeup-interval***)** *(:record* **process::*scheduler-record-interval***)** *(:hysteresis* **process::*scheduler-hysteresis-interval***)** *(:peek* **process::*scheduler-peek-interval***)** *(:resolution* **process::*scheduler-resolution***)** *(:spread* **process::*scheduler-spread***)** *(:boost* **process::*scheduler-boost***)** *(:timer-resolution* **process:*timer-resolution***)**

process:show-scheduler-parameters &optional *(stream* ***standard-output****)*

process:process-process-priority *process*

process:set-process-base-priority *process new-priority*

process:process-base-process-priority *process*

process:process-priority-lessp *proc1 proc2*
Returns **t** if the dispatcher/instantaneous priority of *proc1* is higher than the priority of *proc2*.

process:scheduler-priority-lessp *pri1 pri2*
Returns **t** if *pri1* is higher priority than *pri2*.

process:with-delayed-process-priorities *(priority* &rest *priorities)* &body *body*
Like **process:with-process-priority** except the priority does not start until the indicated time interval has passed. The delay is in seconds.

process:with-delayed-process-priorities-in-timer-units *(priority* &rest *priorities)* &body *body*
Like **process:with-delayed-process-priorities** but the units are timer units, not seconds.

si:with-process-non-interactive-priority *(*&key *(:quantum-boost* **si:*process-command-initial-quantum***))* &body *body* *Function*

:quantum-boost Instead of immediately lowering the priority of this process, the high priority is extended for the length of quantum-boost. Quantum is measured in units of 60ths of a second. The desired effect of *:quantum-boost* is to allow this process to continue to have high priority for a short period of time after you

using non-interactive priority. This allows short commands to execute quickly and makes the machine feel more responsive. The default is **si:*process-command-initial-quantum***

In order to allow you to type without interference while other processes are computing in the background, Genera raises the priority of your process when you are typing. This is called *interactive priority*. Often, you can see a process executing (or waiting) with an interactive priority by looking at the Processes display in Peek. A priority of 0+1 (base priority of 0, current priority of 1) almost always means that the process is currently interactive.

The process only executes with interactive priority while you edit (for example, when you edit text, or when you use the input editor). When you execute code, Genera lowers the priority to *non-interactive priority*, or the normal priority of the process.

The system uses interactive priority to allow your keystrokes to be processed in preference to computations — this makes typing seem smoother and more responsive. This also speeds up the response of mouse-sensitivity highlighting as you move the mouse. The system lowers your priority to non-interactive priority when executing commands or forms so that the computation does not interfere with other computations (or with your typing to another window), so that, in principle, each activity gets a fair portion of the machine.

The system assumes that any input editor command that has an input-editor accelerator will be used to edit text, and therefore, executes it at the interactive priority. The system makes the same assumption about presentation actions.

This can be a problem if you have a command that will execute for a long time, and will be executed at interactive-priority. It can use up a lot of the machine and interfere with other processes.

To solve this problem, you need to wrap a **si:with-process-non-interactive-priority** form around the body of your form.

```
(si:with-process-non-interactive-priority () body)
```

Extensible Scheduler

The scheduler is extensible in the following senses:

1. The representation of, and operations on, programmer or process priorities can be changed, although changing them might involve a change to the exported interface of the scheduler. This is the sort of thing that should only be changed across major releases.

2. The *meaning* of the programmer priority can be changed at any time by changing a small number of procedures and/or parameters.

3. The module(s) that determine what effect the process priority has on the instantaneous priority are replaceable. This collection of modules represents the *scheduler policy*.

Higher-Level Functions

Not all control of processes falls under the heading of scheduler policy. There is certain behavior that a programmer might want, independent of policy. For example, a timer driven process. Primitives to control this kind of behavior should exist, independent of which policy the scheduler is currently using.

process:process-run-function is an example of an already implemented primitive. *Events* and *Periodic-Actions* are examples of primitives that might be implemented in the future.

process:process-run-function allows you to call a function and have its execution happen asynchronously in another process. This can be used either as a simple way to start up a process that will run "forever", or as a way to make something happen without having to wait for it to complete.

It is sometimes syntactically more convenient than **process:make-process**. It also provides a keyword, **:restart-after-reset**, which can allow all **process:reset**s to kill the process. This is often useful behavior for temporary, anonymous, computations. These are the only two reasons to use **process:process-run-function** in place of **process:make-process**.

See the section "Using Processes for Computations".

process:process-run-function *name-or-kwds function* &rest *args* *Function*

Creates a process, presets it so it will apply *function* to *args*, and starts it running. *name-or-kwds* can be a string that becomes the process's name, or it can be a list of alternating keywords and values to which the corresponding process attributes are set.

The keywords are:

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

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

:restart-after-boot Applies to both warm and cold booting, if the cold boot occurs after a disk save with the process in it. If this is **t**, booting the machine restarts the process. If this is **nil**, booting the machine kills the process. The default is **nil**.

:warm-boot-action If this option is provided, its value controls what happens when the machine is warm booted. It overrides **:restart-after-boot**. If it is **nil** or not provided, the value of the **:restart-after-boot** option takes effect. For a description of the value of the warm-boot action:

See the section "Warm Boot Actions for Processes".

All other keywords are treated as arguments to **process:make-process**.

The Scheduler Compatibility Package

Introduction to Processes

Symbolics computers support *multiprocessing*; several computations can be executed "concurrently" by placing each in a separate *process*. A process is like a processor, simulated by software. Each process has its own "program counter", its own stack of function calls and its own special-variable binding environment in which to execute its computation. (This is implemented with stack groups: See the section "Stack Groups".) A process is a Lisp object, an instance of one of several flavors of process. See the section "Process Flavors".)

If all the processes are simply trying to compute, the machine time-slices among them. This is not a particularly efficient mode of operation, since dividing the finite memory and processor power of the machine among several processes certainly cannot increase the available power and in fact wastes some of it in overhead. The way processes are normally used is different: there can be several ongoing computations, but at a given moment only one or two processes are trying to run. The rest are either *waiting* for some event to occur, or *stopped*, that is, not allowed to compete for resources.

A process waits for an event by means of the **process-wait** primitive, which is given a predicate function that defines the event being waited for. A module of the system called the *process scheduler* periodically calls that function. If it returns **nil** the process continues to wait; if it returns **t** the process is made runnable and its call to **process-wait** returns, allowing the computation to proceed.

A process can be *active* or *stopped*. Stopped processes are never allowed to run; they are not considered by the scheduler, and so never become the current process until they are made active again. The scheduler continually tests the waiting functions of all the active processes, and those that return non-**nil** values are allowed to run. When you first create a process with **make-process**, it is stopped.

A process has two sets of Lisp objects associated with it, called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically, keyword symbols and active objects such as windows and other processes are found. A process is considered *active* when it has at least one run reason and no arrest reasons. A process that is not active is *stopped*, is not referenced by the process scheduler, and does not compete for machine resources.

To start a computation in another process, you must first create a process, and then specify the computation you want to occur in that process. The computation to be executed by a process is specified as an *initial function* for the process and a list of arguments to the initial function. When the process starts up it applies the function to the arguments. In some cases the initial function is written so that it

never returns, while in other cases it performs a certain computation and then returns, which stops the process.

To *reset* a process means to "throw out" of its entire computation, then force it to call its initial function again. (See the special form **throw**.) Resetting a process clears its waiting condition, and so if it is active it becomes runnable. To *preset* a process is to set up its initial function (and arguments), and then reset it. This is one way to start up a computation in a process.

All processes in a Symbolics computer run in the same virtual address space, sharing the same set of Lisp objects. Unlike other systems, which have special restricted mechanisms for interprocess communication, Symbolics computers allow processes to communicate in arbitrary ways through shared Lisp objects. One process can inform another of an event simply by changing the value of a global variable. Buffers containing messages from one process to another can be implemented as lists or arrays. The usual mechanisms of atomic operations, critical sections, and interlocks are provided. For more information:

See the function **store-conditional**.
See the special form **without-interrupts**.
See the function **process-lock**.

Obsolete Process Functions

The functions in this section are obsolete and form part of the Scheduler Compatibility Package. Their documentation is retained to assist in reading old code. New code should use the new functions in the **process** package.

Process Attribute Functions

These functions let you find out the attributes of a process.

process-name *process* *Function*

Returns the name of *process*, which was the first argument to **process:make-process** or **process:process-run-function** when the process was created. The name is a string that appears in the printed representation of the process, stands for the process in the status line and the Peek display, and so on.

process-stack-group *process* *Function*

Returns the stack group currently executing on behalf of *process*. This can be different from the initial-stack-group if the process contains several stack groups that coroutine among themselves. If the process is *simple*, or it has released its resources, then it might not have a stack group. In that case **process-stack-group** returns **nil**.

process-initial-stack-group *process* *Function*

Returns the stack group the initial-function is called in when *process* starts up or is reset.

Note that the initial stack group of a *simple* process is not a stack group at all, but a list.

process-initial-form *process* *Function*

Returns the initial "form" of *process*. This is not really a Lisp form; it is a cons whose car is the initial-function and whose cdr is the list of arguments to which that function is applied when the process starts up or is reset.

To change the initial form, call the **process:preset** function.

process-wait-function *process* *Function*

Returns *process*'s current wait-function, which is the predicate used by the scheduler to determine if the process is runnable. This is **#'true** if the process is running, and **#'false** if the process has no current computation (just created, initial function has returned, or "flushed").

process-wait-argument-list *process* *Function*

Returns the arguments to *process*'s current wait-function. This is frequently the **&rest** argument to **process-wait** in the process's stack, rather than a true list. The system always uses it in a safe manner, that is, it forgets about it before **process-wait** returns.

process-whostate *process* *Function*

Returns a string that is the state of the process to go in the status line at the bottom of the screen. This is the value of **process::*process-run-whostate*** (defaulting to "Run") if the process is running or trying to run, otherwise the reason why the process is waiting. If the process is stopped, then this whostate string is ignored and the status line displays "Arrested" if the process is arrested or "Stopped" if the process has no run reasons.

si:default-quantum *Variable*

The number of 60ths of a second a process is allowed to run without waiting before rescheduling. It is used when a new process is created. The default is 6 (0.1 seconds).

process-quantum *process* *Function*

Returns the number of 60ths of a second *process* is allowed to run without waiting before the scheduler runs something else. The quantum default is governed by the variable **si:default-quantum**. **zl:setf** can be used to change its value.

process-quantum-remaining *process* *Function*

Returns the amount of time remaining for *process* to run before rescheduling, in 60ths of a second.

process-priority *process* *Function*

Returns the priority of *process*. Use **zl:setf** to change its value. The default priority is 0.

process-priority converts the processes priority into values that are compatible with the old scheduler. **zl:setf** of **process-priority** takes old scheduler priorities as arguments, and **process-priority** returns old scheduler priorities as values.

You can convert between old and new scheduler priorities by using the functions **process::process-priority-to-old-priority** and **process:make-process-priority** with a class of **:foreground**.

process-warm-boot-action *process* *Function*

Returns *process*'s warm-boot-action, which controls what happens to this process if the machine is booted. Use **zl:setf** to change its value. (Note: Contrary to the name, this applies to both cold and warm booting.) This can be a function to call or **nil**, which means to "flush" the process. The default is **process:process-warm-boot-delayed-restart**, which resets the process after initializations have been completed, causing it to start over at its initial function. You can also use **process:process-warm-boot-reset**, which throws out of the process's computation and kills the process.

process-simple-p *process* *Function*

Returns **t** for a simple process, **nil** for a normal process.

Run and Arrest Reason Functions

This section describes the functions used to specify the run reasons and arrest reasons for processes.

si:process-run-reasons *process* *Function*

Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

si:process-arrest-reasons *process* *Function*

Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

process-enable *process* *Function*

Activates *process* by revoking all its run and arrest reasons, then giving it a run reason of **:enable**.

process-disable *process* *Function*

Stops *process* by revoking all its run reasons. Also revokes all its arrest reasons.

process-enable-run-reason *process* &optional *(reason* **:user***)* *Function*

Adds *reason* to the process's run reasons. This can activate the process.

process-disable-run-reason *process* &optional *(reason* **:user***)* *Function*

Removes *reason* from the process's run reasons. This can stop the process.

process-enable-arrest-reason *process* &optional *(reason* **:user***)* *Function*

Adds *reason* to the process's arrest reasons. This can stop the process.

process-disable-arrest-reason *process* &optional *(reason* **:user***)* *Function*

Removes *reason* from the process's arrest reasons. This can activate the process.

process-active-p *Function*

Returns **t** if the process is active, that is, it can run if its wait-function allows. Returns **nil** if the process is stopped.

Functions for Starting and Stopping Processes

This section describes the functions used to start and stop processes.

process-preset *process function* &rest *args* *Function*

Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that it throws out of any current computation and starts itself up by applying *function* to *args*. A **process-preset** call to a stopped process returns immediately, but does not activate the process, hence the process does not really apply *function* to *args* until it is activated later.

process-reset *process* &optional *unwind-option kill (without-aborts* **:ask***)*

Function

Forces *process* to reset, that is, to throw out of its present computation and apply its initial function to its initial arguments, when it next runs. The throwing out is skipped if the process has no present computation (for example, it was just created), or if **:unwind-option** option so specifies. The possible values for **:unwind-option** are:

:unless-current or **nil**
: Unwind unless the stack group to be unwound is the one we are currently executing in, or belongs to the current process.

:always
: Unwind in all cases. This can cause the **process-reset** to throw through its caller instead of returning.

t
: Never unwind.

If **:kill** is **t**, the process is to be killed after unwinding it. This is for internal use by the **process-kill** function only.

:without-aborts indicates what to do if the process is not currently resettable. See the function **sys:without-aborts**. It takes the values **:force**, **:ask**, and **nil**.

:force
: Process is reset anyway.

:ask
: Queries the user as to whether to force the process to reset or not.

nil
: Return a list of reasons why the process cannot be reset.

If the process is arrested, and is inside a **process::without-aborts** or a **unwind-protect** cleanup, in the old scheduler **process-reset** would behave as if **:without-aborts** had been **nil**. In the new scheduler (even with the old scheduler entry-points) **:without-aborts** continues to be respected. In particular, if **:without-aborts** has the default value, **:ask**, the user is prompted even if the process is arrested.

A **process-reset** call to a stopped process returns immediately, but does not activate the process, hence the process does not really get reset until it is activated later.

process-reset-and-enable *process* *Function*

Resets *process*, then enables it.

process-abort *process* &key *message all* (*query* **t**) (*time-out* **si:*default-process-abort-timeout***) *stream* *Function*

Used by the ABORT key. It aborts *process* but respects processes that are executing code in or are called in the body of a **sys:without-aborts** macro (see the function **sys:without-aborts**). **sys:without-aborts** notifies users when they attempt to abort

a process that is executing code in or is called in the body of the macro. Most code that currently uses the form **(signal 'abort)** should instead use **(process-abort *current-process*)**.

process-abort waits until the process is abortable or asks the user what to do if it is not abortable. It returns **t** if it successfully aborts the process or **nil** on a failure to abort. When queried, the user can force the process to abort, examine it with the Debugger, wait longer for it to become abortable, or abandon the attempt to abort it. Each time the user forces a process to abort, an entry is made in the variable **si:*processes-forcibly-aborted***.

process-abort takes several keyword parameters.

:all	Default: **t**, aborts all the way (reset the process). If it is **nil**, aborts to the innermost command level (signal an "abort" condition).
:message	String or **nil**. The string is printed on the process's terminal-io. If **:message** is **nil**, nothing is printed there.
:stream	Overrides the default destination for **:message**.
:time-out	How long to wait (in 60ths of a second) when the process is not abortable.
:query	**nil** to give up after an interval of **:time-out**, **t** to query the user what to do, **:cold** to query the user via the cold-load stream, or **:pop-up** to query the user via a pop-up menu. **:pop-up** is the default.

The variable **si:*default-process-abort-timeout*** is the number of 60ths of a second to wait before consulting the user, when a process is to be reset or aborted but it is not abortable. The default value is 300 (5 seconds).

process-flush *process* *Function*

Forces *process* to wait forever. A process cannot flush itself. Flushing a process is different from stopping (killing) it, in that it is still active; thus, if it is reset or preset, it starts running again.

process-kill *process* &optional *(without-aborts* **:ask**)

Function

Terminates *process*. The process is reset, stopped, and removed from **sys:all-processes**.

without-aborts indicates what to do if the process is not currently abortable. It takes the values **:force**, **:ask**, and **nil**.

:force	The process is aborted anyway.

:ask Queries the user as to whether to force the process to abort or not.

nil Return a list of reasons why the process cannot be aborted and do not kill the process.

process-interrupt *process function* &rest *args*

Function

Forces the process to apply *function* to *args*. When *function* returns, the process continues the interrupted computation. If the process is waiting, it wakes up, calls *function*, then waits again when *function* returns.

If the process is stopped it does not apply *function* to *args* immediately, but later when it is activated. Normally the **process-interrupt** function returns immediately, but if the process's stack group is in an unusual internal state, **process-interrupt** might have to wait for the process to exit that state.

If *function* does a **throw** and *process* is not abortable, it could be subverted by having its computation aborted unexpectedly. In some cases **process-abort** should be used instead of **process-interrupt**. If *function* returns normally, this is not a problem.

Obsolete Scheduler Functions

make-process *name* &rest *init-args* &key *:initial-form :wait-function :wait-argument-list :run-reasons (:priority* **0***) :simple-p :flavor* &allow-other-keys *name* &rest *init-args* *Function*

This function is obsolete and retained only for compatibility. **process:make-process** should be used in new code.

Creates and returns a process named *name* (a string). The process will not be capable of running until it has been reset or preset in order to initialize the state of its computation.

The *init-args* are alternating keywords and values that allow you to specify things about the process; however, no options are necessary if you are not doing anything unusual. The following init-args are allowed:

:simple-p Specifying **t** here gives you a simple process. See the section "Process Flavors".

:flavor Specifies the flavor of process to be created. For a list of all the flavors of process supplied by the system: See the section "Process Flavors".

:stack-group The stack group the process is to use. If this option is not specified a stack group will be created according to the relevant options below.

:warm-boot-action What to do with the process when the machine is booted. See the method **(flavor:method :warm-boot-action process:process)**.

:quantum The number of seconds the process is allowed to be run without waiting before the scheduler runs another process. The value should be a fixnum in units of 60ths of a second. The default is 6 (0.1 second). See the method **(flavor:method :quantum process:process)**.

:priority The priority of the process. The default is 0. See the method **(flavor:method :priority process:process)**.

:run-reasons Lets you supply an initial list of run reasons. The default is **nil**. *Note: Do not use the :run-reasons init-option to start a process. The only way to start a process created with* **make-process** *is to preset it and explicitly specify the run-reason with the* **process-enable-run-reason** *function.*

:arrest-reasons Lets you supply an initial list of arrest reasons. The default is **nil**.

In addition, the options of **make-stack-group** are accepted. See the function **make-stack-group**.

If you specify **:flavor**, there can be additional options implemented by that flavor.

process-run-function *name-or-kwds function* &rest *args* *Function*

Creates a process, presets it so it will apply *function* to *args*, and starts it running. *name-or-kwds* can be a string that becomes the process's name, or it can be a list of alternating keywords and values to which the corresponding process attributes are set.

The keywords are:

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

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

:restart-after-boot Applies to both warm and cold booting, if the cold boot occurs after a disk save with the process in it. If this is **t**, booting the machine restarts the process. If this is **nil**, booting the machine kills the process. The default is **nil**.

:warm-boot-action If this option is provided, its value controls what happens when the machine is warm booted. It overrides **:restart-after-boot**. If it is **nil** or not provided, the value of the **:restart-after-boot** option takes effect. For a description of the value of the warm-boot action: See the section "Warm Boot Actions for Processes".

:priority The priority of the process. This priority is a **:foreground** priority, expressed as an old scheduler priority. The default is 0.

Process Messages

This section describes the messages that can be sent to any flavor of process. Certain process flavors can define additional messages. Not all possible messages are listed here, only those of interest to most users.

This message documentation is provided for compatibility with existing programs written using messages. New programs should not use these messages. Rather, they should use the functions with similar names, but with **process-** prepended to them. See the section "Obsolete Process Functions".

Process Attribute Messages

(flavor:method :name process:process) *Method*

This message should not be used. It is identical to the function **process-name**. See the function **process-name**.

(flavor:method :stack-group process:process) *Method*

This message should not be used. It is identical to the function **process-stack-group**.

See the section "Process Stack Groups".

(flavor:method :initial-stack-group process:process) *Method*

This message should not be used. It is identical to the function **process-initial-stack-group**.

See the function **process-initial-stack-group**.

(flavor:method :initial-form process:process) *Method*

This message should not be used. It is identical to the function **process-initial-form**.

See the function **process-initial-form**.

(flavor:method :wait-function process:process) *Method*

This message should not be used. It is identical to the function **process-wait-function**.

See the function **process-wait-function**.

(flavor:method :wait-argument-list process:process) *Method*

This message should not be used. It is identical to the function **process-wait-argument-list**.

See the function **process-wait-argument-list**.

(flavor:method :whostate process:process) *Method*

This message should not be used. It is identical to the function **process-whostate**.

See the function **process-whostate**.

(flavor:method :quantum process:process) *Method*

This method is obsolete.

(flavor:method :set-quantum process:process) *60ths* *Method*

This method is obsolete. It is included in the system for compatibility, but it has no effect.

si:default-quantum *Variable*

The number of 60ths of a second a process is allowed to run without waiting before rescheduling. It is used when a new process is created. The default is 6 (0.1 seconds).

(flavor:method :quantum-remaining process:process) *Method*

This method is obsolete.

(flavor:method :priority process:process) *Method*

This message should not be used. It is identical to the function **process-priority**.

See the function **process-priority**.

(flavor:method :set-priority process:process) *priority-number* *Method*

This message should not be used. It is identical to **zl:setf** of the function **process-priority**.

See the function **process-priority**.

(flavor:method :warm-boot-action process:process) *Method*

This message should not be used. It is identical to the function **process-warm-boot-action**.

See the function **process-warm-boot-action**.

(flavor:method :set-warm-boot-action process:process) *action* *Method*

This message should not be used. It is idential to **zl:setf** of the function **process-warm-boot-action**.

See the function **process-warm-boot-action**.

(flavor:method :simple-p process:process) *Method*

This message should not be used. It is identical to the function **process-simple-p**.

See the function **process-simple-p**.

Run and Arrest Reason Messages

(flavor:method :run-reasons process:process) *Method*

This message should not be used. It is identical to the function **si:process-run-reasons**. See the function **si:process-run-reasons**.

(flavor:method :run-reason process:process) *object* *Method*

This message should not be used. It is identical to the function **process-enable-run-reason**. See the function **process-enable-run-reason**.

(flavor:method :revoke-run-reason process:process) *object* *Method*

This message should not be used. It is identical to the function **process-disable-run-reason**. See the function **process-disable-run-reason**.

(flavor:method :arrest-reasons process:process) *Method*

This message should not be used. It is identical to the function **si:process-arrest-reasons**. See the function **si:process-arrest-reasons**.

(flavor:method :arrest-reason process:process) *object* *Method*

This message should not be used. It is identical to the function **process-enable-arrest-reason**. See the function **process-enable-arrest-reason**.

(flavor:method :revoke-arrest-reason process:process) *object* *Method*

This message should not be used. It is identical to the function **process-disable-arrest-reason**. See the function **process-disable-arrest-reason**.

(flavor:method :active-p process:process) *Method*

This message should not be used. It is identical to the function **process-active-p**. See the function **process-active-p**.

(flavor:method :runnable-p process:process) *Method*

This message should not be used. It is identical to the function **process-active-p**. See the function **process-active-p**.

Messages for Stopping the Process

(flavor:method :preset process:process) *function* &rest *args* *Method*

This message should not be used. It is identical to the function **process-preset**. See the function **process-preset**.

(flavor:method :reset process:process) &optional *unwind-option kill without-aborts* *Method*

This message should not be used. It is identical to the function **process-reset**. See the function **process-reset**.

(flavor:method :flush process:process) *Method*

This message should not be used. It is identical to the function **process-flush**. See the function **process-flush**.

(flavor:method :kill process:process) &optional *without-aborts* *Method*

This message should not be used. It is identical to the function **process-kill**. See the function **process-kill**.

(flavor:method :interrupt process:process) *function* &rest *args* *Method*

This message should not be used. It is identical to the function **process-interrupt**. See the function **process-interrupt**.

Process Flavors

There is one flavor of process provided by the system. It is possible for users to define additional flavors of their own.

process:process *Flavor*

This is the standard default flavor of process. See its instance variables, initializations, and methods by using the Flavor Examiner, SELECT-X.

Old Scheduler Version of simple-process

A simple process is not a process in the conventional sense. It has no stack group of its own; instead of having a stack group that gets resumed when it is time for the process to run, it has a function that gets called when it is time for the process to run. When the wait-function of a simple process becomes true, and the scheduler notices it, the simple process's function is called, in the scheduler's own stack group. Since a simple process does not have any stack group of its own, it cannot save "control" state in between calls; any state that it saves must be saved in data structure.

The only advantage of simple processes over normal processes is that they use up less system overhead, since they can be scheduled without the cost of resuming stack groups. They are intended as a special, efficient mechanism for certain purposes. For example, packets received from the Chaosnet are examined and distributed to the proper receiver by a simple process that wakes up whenever there are any packets in the input buffer. However, they are harder to use, because you cannot save state information across scheduling. That is, when the simple process is ready to wait again, it must return; it cannot call **process-wait** and continue to do something else later. In fact, it is an error to call **process-wait** from inside a simple process. Another drawback to simple processes is that if the function signals an error, the scheduler itself will be broken, and multiprocessing will stop; this situation can be hard to repair. Also, while a simple process is running, no other process is scheduled; simple processes should never run for a long time without returning, so that other processes can run.

Asking for the stack group of a simple process does not signal an error, but returns the process's function instead.

Since a simple process cannot call **process-wait**, it needs some other way to specify its wait-function. To set the wait-function of a simple process, use **set-process-wait**. So, when a simple process wants to wait for a condition, it should call **set-process-wait** to specify the condition, **setf** its **process-whostate** to a string that defines what it's waiting for and then return.

set-process-wait *simple-process wait-function wait-argument-list* *Function*

Sets the *wait-function* and *wait-argument-list* of *simple-process*. For more information: See the section "Old Scheduler Version of **simple-process**".

Process Priority Levels in the Old Scheduler

Normal processes run with a default priority of 0 when computing and 1 when they are interacting with a user. If the priority number is higher, the process receives higher priority. You should avoid using priority values higher than 9, since some critical system processes use priorities of 10 to 30; setting up competing processes could lead to degraded performance or system failure. You can also use negative values to get processes to run in the background. Values of -5 or -10 for background processes and 2 or 5 for urgent processes are reasonable.

Use the Command Processor command Show Processes to see the priorities used by existing processes.

Only the relative values of these numbers are important. (You could use floating-point numbers to squeeze in more intermediate levels, though there should never be any need to do so.)

Once these relative priority values are set, be advised that the process priorities are interpreted consistently. If a priority 1 process runs forever without calling **process-wait**, no lower-priority process will ever run.

Timer Queues

Periodically a system process wakes up and selects an item off the *timer priority queue* or *timer queue*. The timer queue is a list of items in time order. You can directly add functions to the timer queue or remove them. This mechanism enables you to perform a periodic action without the overhead of process waits and timeouts.

The timer queue list shows up in the Peek display. See the section "The Peek Program".

Timer queues are obsolete. They should be replaced by Timers and Periodic Actions in new code. See the section "Timers".

si:add-timer-queue-entry *time repeat name function* &rest *args* *Function*

Adds an entry to the timer queue. *function* is called with *args* when the timer fires. *time* can be in the form

```
(:absolute universal-time)
```

(or just universal-time) or

```
(:relative n-secs)
```

repeat is of the form **:once**, (**:forever** *n-secs*), or (*n-times n-secs*). *name* is a string that names the timer queue entry.

si:add-timer-queue-entry returns the id of the entry. To effect a "repeat function," the called function can (conditionally) add another timer queue entry.

si:remove-timer-queue-entry *timer-id* *Function*

Removes the entry which has *timer-id* as its id. Note: the timer-id is returned by **si:add-timer-queue-entry**. See the function **si:add-timer-queue-entry**.

si:print-timer-queue &optional *stream* *Function*

Prints the contents of the timer queue. Optionally the *stream* to which the queue is printed can be specified.

Initializations

Introduction to Initializations

A number of Genera programs and facilities require that "initialization routines" be run when the facility is first loaded, or when the system is booted, or both. These initialization routines can set up data structures, start processes running, open network connections, and so on.

An initialization that needs to be done once, when a file is loaded, can be done simply by putting the Lisp forms to do it in that file; when the file is loaded the forms are evaluated. However, some initializations need to be done each time the system is booted, and some initializations depend on several files having been loaded before they can work. Also, some initializations should be done once and only once, regardless of any particular file being reloaded.

The system provides a consistent scheme for managing these initializations. Rather than having a magic function that runs when the system is started and knows everything that needs to be initialized, each thing that needs initialization contains its own initialization routine. The system keeps track of all the initializations through a set of functions and conventions, and executes all the initialization routines when necessary. The system also avoids reexecuting initializations if a program file is loaded again after it has already been loaded and initialized.

An *initialization list* is a symbol whose value is an ordered list of *initializations*. Each initialization has a name, a form to be evaluated, a flag saying whether the form has yet been evaluated, and the source file of the initialization, if any. When the time comes, initializations are evaluated in the order that they were added to the list. The name is a string and lies in the **car** of an initialization; thus **assoc** can be used on initialization lists. All initialization lists also have a **si:initialization-list** property of **t**. This is mainly for internal use.

add-initialization *name form* &optional *keywords list-name* *Function*

Adds an initialization called *name* (a string) with the form *form* to the initialization list specified either by *list-name* or by *keywords*. If the initialization list already contains an initialization called *name*, it is removed and the new one is added.

list-name, if specified, is a symbol that has as its value the initialization list. If it is unbound, it is initialized to **nil**, and is given an **si:initialization-list** property of **t**. If the *keywords* specify an initialization list, *list-name* is ignored and should not be specified.

Two kinds of keywords are allowed. The first kind specifies which initialization list to use. This is the *which* keyword. All the *which* keywords are shown here:

:cold	Use the standard cold-boot list.
:warm	Use the standard warm-boot list. This is the default.
:before-cold	Use the standard before-disk-save list.
:once	Use the once-only list.
:system	Use the system list.
:login	Use the login list.
:logout	Use the logout list.
:site	Use the site list. (The *form* is evaluated immediately by default, as well as each time a site initialization is performed.)
:enable-services	Use the enable-services list.
:disable-services	Use the disable-services list.
:window	Use the window list.
:full-gc	Use the full-gc list.
:after-full-gc	Use the after-full-gc list.

For more information on these lists: See the section "System Initialization Lists".

The second kind of keyword specifies when to evaluate *form*. This is the *when* keyword, of which there are four:

:normal Only place the form on the list. Do not evaluate it until the time comes to do this kind of initialization. This is the default unless **:system**, **:site**, or **:once** is specified.

:now Evaluate the form now as well as adding it to the list. (This is the default for **:site**.)

:first Evaluate the form now if it is not flagged as having been evaluated before. This is the default if **:system** or **:once** is specified.

:redo Do not evaluate the form now, but set the flag to **nil** even if the initialization is already in the list and flagged **t**.

Actually, the keywords are compared with **string-equal** and can be in any package. If both kinds of keywords are used, the *which* keyword should come before the *when* keyword in *keywords*; otherwise the *which* keyword can override the *when* keyword.

The **add-initialization** function keeps each list ordered so that initializations added first are at the front of the list. Therefore, by controlling the order of execution of the additions, explicit dependencies on order of initialization can be controlled. Typically, the order of additions is controlled by the loading order of files. The **:system** list is the most critically ordered of the predefined lists. See the section "System Initialization Lists".

delete-initialization *name* &optional *keywords list-name* *Function*

Removes the specified initialization from the specified initialization list. Keywords can be any of the list options allowed by **add-initialization**.

initializations *list-name* &optional *redo-flag (flag* **t***)* *Function*

Performs the initializations in the specified list. *redo-flag* controls whether initializations that have already been performed are re-performed; **nil** means no, non-**nil** is yes, and the default is **nil**. *flag* is the value to be stored into the flag slot of an entry when the initialization form is run. If it is unspecified, it defaults to **t**, meaning that the system should remember that the initialization has been done. There is no convenient way for you to specify one of the specially-known-about lists because you should not be calling **initializations** on them.

reset-initializations *list-name* *Function*

Sets the flag of all entries in the specified list to **nil**, thereby causing them to be rerun the next time the function **initializations** is called on the initialization list.

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

si:initialization-keywords *Variable*

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

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

System Initialization Lists

The special initialization lists that are known about by the initialization functions allow you to have your subsystems initialized at various critical times without modifying any system code to know about your particular subsystems. This also allows only a subset of all possible subsystems to be loaded without necessitating either modifying system code (such as **sys:lisp-reinitialize**) or such awkward methods as using **fboundp** to check whether or not something is loaded.

The **:once** initialization list is used for initializations that need to be done only once when the subsystem is loaded and must never be done again. For example, some databases need to be initialized the first time the subsystem is loaded, but they should not be reinitialized every time a new version of the software is loaded into a currently running system. This list is for that purpose. The **initializations** function is never run over it; its "when" keyword defaults to **:first** and so the form is normally evaluated only at load-time, and only if it has not been evaluated before. The **:once** initialization list serves a similar purpose to the **defvar** special form, which sets a variable only if it is unbound.

The **:system** initialization list is for things that need to be done before other initializations stand any chance of working. Initializing the process and window systems, the file system, and the Chaosnet NCP falls in this category. The initializations on this list are run every time the machine is cold- or warm-booted, as well as when the subsystem is loaded unless explicitly overridden by a **:normal** option in the keywords list. In general, the system list should not be touched by user subsystems, though there can be cases when it is necessary to do so.

The **:cold** initialization list is used for things that must be run once at cold-boot time. The initializations on this list are run after the ones on **:system** but before the ones on the **:warm** list. They are run only once, but are reset by **zl:disk-save**, thus giving the appearance of being run only at cold-boot time.

The **:warm** initialization list is used for things that must be run every time the machine is booted, including warm boots. The function that prints the greeting, for example, is on this list. Unlike the **:cold** list, the **:warm** list initializations are run regardless of their flags.

The **:before-cold** initialization list is a variant of the **:cold** list. These initializations are run before the world is saved out by **zl:disk-save**. Thus they happen essentially at cold-boot time, but only once when the world is saved, not each time it is started up.

The **:login** and **:logout** lists are run by the **zl:login** and **zl:logout** functions, respectively. Note that **zl:disk-save** calls **zl:logout**. Also note that often people do not call **zl:logout**; they just cold boot the machine.

The forms on **:enable-services** are run by **sys:enable-services**. In addition, they are run automatically by **sys:lisp-reinitialize** when a nonserver Symbolics computer is warm- or cold-booted.

The forms on **:disable-services** are run by **sys:disable-services**. In addition, they are run automatically by **:before-cold** when you use **zl:disk-save**.

The **:window** initialization list is run when the Genera window system comes up, after the screen has been created but before the initial activity (usually the initial

Lisp Listener) has been displayed. In native systems such as the 3600 and XL400, the :window initializations run before the **:cold** and **:warm** initializations. In embedded systems such as the MacIvory and UX-family machines, the **:window** initializations run after the **:warm** initializations, and do not run at all if the host window system is being used instead of the Genera window system.

The forms on **:full-gc** are run by **si:full-gc** before running the garbage collector.

The forms on **:after-full-gc** are run by **si:full-gc** after it collects all the garbage.

User programs are free to create their own initialization lists to be run at their own times. Some system programs, such as the editor, have their own initialization list for their own purposes.

Storage Management

Overview of Storage Management

The Genera virtual memory system offers users and programmers the ability to run extremely large programs in a virtual memory which, depending on available disk space, can be on the order of 1 billion bytes.

Genera also has facilities for both automatic and manual (program-controlled) management of virtual storage. Simply stated, storage management is a strategy for allocating pieces of memory as they are needed by a program and then freeing the memory for reuse when it is no longer needed for the same purpose.

Automatic Storage Management

Some virtual memory systems concentrate exclusively (in the automatic case) on managing the stack, because they are optimized for programming languages that allocate most temporary storage on the stack.

In Lisp, however, management of the stack would in no way be sufficient, since programs nearly always allocate large structures and lists in "ordinary" virtual memory. Automatic storage management is nevertheless an extremely important aspect of Lisp programming, because deciding in an application program whether storage can be freed safely is such a difficult problem, difficult enough that programmers should not be faced with it routinely. Automatic storage management in Genera is performed by a suite of programs collectively called the garbage collector. See the section "The Garbage Collector Facilities".

Areas are also provided, which help you improve the locality of reference in programs without giving up the ease of automatic storage management. See the section "Areas".

See the section "How Garbage Collection Improves Locality of Reference".

Manual Storage Management

"Manual" storage management means that the allocation and freeing of virtual memory is controlled by your application program. It should be regarded as a special purpose technique, but it is nevertheless a necessity in some cases.

One of the primary manual storage management facilities is the *resource*. See the section "Resources".

See the section "Consing Lists on the Control Stack".

Manual storage management includes the use of the technique of *wiring* parts of memory. To wire a piece of memory means to lock its contents in main (semiconductor) memory and not allow it to be paged to disk. See the section "Wiring Memory". This technique is useful for critical applications in which a program cannot wait for certain information to be paged in when needed.

Areas

Storage in the Symbolics system is divided into *areas*. Each area contains related objects, of any type. Areas are intended to give you control over the paging behavior of your program, among other things. By putting related data together, *locality of reference* can be greatly increased.

Locality of reference is a desirable property of programs that run in a virtual memory environment like Genera. It means, essentially, that objects and their references (or more generally, any pieces of related information), are located near each other, that is, located at nearby addresses in virtual memory. When this is true, the paging system can avoid *thrashing*: swapping many pages in and out of main memory in order to access relatively few data.

The use of areas is a programming technique available in Genera that improves locality of reference in programs that allocate virtual memory in large amounts and for specific purposes. Areas are especially useful when the objects allocated are static, since the objects will then be left completely alone by most kinds of garbage collection.

Whenever you create a new object you can also specify the area it uses. For example, instead of using **cons** you can use **cons-in-area**. Object-creating functions that take keyword arguments generally accept a **:area** argument. You can also control which area is used by binding **sys:default-cons-area**; most functions that allocate storage use the value of this variable, by default, to specify the area to use. The default value of **sys:default-cons-area** is *working-storage-area*.

Specifying the area as an argument is usually preferred over binding the variable because it gives you more control and avoids accidentally getting other objects into your area.

Areas also give you a handle to control the garbage collector. Some areas can be declared to be *static*, which means that they change slowly and the garbage collector should not attempt to reclaim any space in them. This can eliminate a lot of useless copying.

Each area can potentially have a different storage discipline and a different paging algorithm. Each area has a name and a number. The name is a symbol whose value is the number. The number is an index into various internal tables. Normally the name is treated as a special variable, so the number is what is given as an argument to a function that takes an area as an argument. Thus, areas are not Lisp objects; you cannot pass an area itself as an argument to a function, you just pass its number. There is a maximum number of areas (set at cold-load generation time); you can only have that many areas before the various internal tables overflow. Currently the limit is 128 areas, of which about 30 already exist in a cold-booted system.

The area mechanism can be overused. If you put two objects into different areas, it is guaranteed that they will never be near each other in virtual memory. If you put each type of object in your program in a different area, you might cause performance degradations. For maximum benefit, objects in different areas should be completely unrelated or used at different times.

Regions Within Areas

The storage of an area consists of one or more *regions*. Each region is a contiguous section of address space with certain homogeneous properties. One of these is the *data representation type*. A given region can only store one type. The two types are *list* and *structure*. A list is anything made out of conses (a closure, for instance). A structure is anything made out of a block of memory with a header at the front: symbols, strings, arrays, instances, bignums, compiled functions, and so on. Since lists and structures cannot be stored in the same region, they cannot be on the same page. It is necessary to know about this when using areas to increase locality of reference.

When you create an area, no regions are created initially. When you create an object in some area, the system tries to find a region that has the right data representation type to hold it, and that has enough room for it to fit. If no such region exists, it makes a new one or, if possible, extends an existing one (or signals an error; see the **:size** option to **make-area**). The size of the new region is an attribute of the area (controllable by the **:region-size** option to **make-area**).

If regions are too large, memory can get taken up by a region and never used. If regions are too small, the system can run out of regions because regions, like areas, are defined by internal tables that have a fixed size (set at cold-load generation time). The limit is **sys:number-of-regions** regions, of which about 90 already exist when you start in a cold-booted system. The system will grow or shrink regions as required so these limitations are usually not a problem.

Area Functions and Variables

default-cons-area *Variable*

The value of this variable is the number of the area in which objects are created by default. It is initially the number of **sys:working-storage-area**. Giving **nil** where an area is required uses the value of ***default-cons-area***. Note that to put objects into an area other than **sys:working-storage-area** you can either bind this variable or use functions such as **cons-in-area** that take the area as an explicit argument. The latter technique is usually preferred since it avoids accidentally getting other objects into your area. (It is not wise to bind this variable to the number corresponding to a temporary area.)

sys:default-cons-area *Variable*

In your new programs, we recommend that you use ***default-cons-area*** which is the Symbolics Common Lisp equivalent of **sys:default-cons-area**.

The value of this variable is the number of the area in which objects are created by default. It is initially the number of **sys:working-storage-area**. Giving **nil** where an area is required uses the value of **sys:default-cons-area**. Note that to put objects into an area other than **sys:working-storage-area** you can either bind this variable or use functions such as **cons-in-area** that take the area as an explicit argument. The latter technique is usually preferred since it avoids accidentally getting other objects into your area. (It is not wise to bind this variable to the number corresponding to a temporary area.)

make-area &key *:name :size :region-size :representation :gc :read-only :swap-recommendations (:n-levels* **2***) (:capacity* **200000***) (:capacity-ratio* **0.5***) :room :fixed-size sys:%%region-level sys:%%region-space-type (sys:%%region-scavenge-enable* **1***) (:n-extra-levels* **0***)* *Function*

Creates a new area, whose name and attributes are specified by the keywords; it can also be used to change the characteristics of an existing area. You must specify a symbol as a name. The symbol is **setq**ed to the area-number of the new area, and that number is also returned, so that you can use **make-area** as the initialization of a **defvar**. The keywords beginning with % are similar to subprimitives; their meanings are system-dependent, and they should not be used in user programs.

The following keywords exist:

:name A symbol that will be the name of the area. This item is required. If it names an existing area, the effect is to change the characteristics of that area.

:size The maximum allowed size of the area, in words. Defaults to infinite. If the number of words allocated to the area reaches this size, attempting to cons an object in the area signals an error.

:gc The type of garbage collection to be employed. The choices are **:dynamic** (which is the default), **:ephemeral**, **:static**, **:safeguarded**, and **:temporary**.

:dynamic means that the area is subject to ordinary incremental garbage collection.

:ephemeral means that objects created in this area (while the ephemeral-object garbage collector is operating) are likely to become garbage soon after their creation; the ephemeral-object garbage collector will concentrate on this area.

:static means that the area will not be copied by the garbage collector, and nothing in the area or pointed to by the area will ever be reclaimed unless a garbage collection of this area is manually requested.

:safeguarded is similar to **:static** but prohibits the garbage collector from ever flipping the area, even by user request. This has limited use for device drivers and similar low-level applications which require that buffer addresses never change.

:temporary, a rarely used and risky option, is for manual storage management, wherein you clear the area by an explicit, programmed action instead of having the area garbage-collected automatically. See the section "The **sys:reset-temporary-area** Feature".

:n-levels
A fixnum (default 2) specifying the number of levels for ephemeral objects; this keyword is valid only for ephemeral areas. That is, the area must either be ephemeral already, or the call including this option must also include **:gc :ephemeral**.

:capacity
An integer specifying the capacity of the first level or a list of integers specifying the capacity of each level in words (default 200000 decimal). This keyword is valid only for ephemeral areas. That is, the area must either be ephemeral already, or the call including this option must also include **:gc :ephemeral**. If the list is too short, the last element is multiplied by the ratio, in same way as when a single number is supplied.

:capacity-ratio
A number (default 0.5) specifying the ratio of capacities in adjacent ephemeral levels. That is, **:capacity** gives the capacity of the first ephemeral level, which is multiplied by the ratio to give the second level's capacity, and so on. This keyword is valid only for ephemeral areas; that is, the area must either be ephemeral already, or the call including this option must also include **:gc :ephemeral**. **:capacity-ratio** applies after the **:capacity** list runs out.

:room With an argument of **t**, adds this area to the list of areas that are displayed by default by the **zl:room** function. The default is **nil**.

:read-only
With an argument of **t**, causes the area to be made read-only. Defaults to **nil**. If an area is read-only, any attempt to change anything in it (altering a data object in the area or creating a new object in the area) signals an error.

:swap-recommendations
Sets the number of extra pages to be read in from disk after a page from this area is brought in due to demand paging.

:fixed-size
If you set this to **t** then the system will never make your regions smaller, even if they contain unused space and address space is running out. **:fixed-size** does not prevent your regions from expanding if you fill them up and free address space is available after the end of the region. The default is **nil**.

:region-size
The approximate size, in words, for regions within this area. The default is the area size if a **:size** argument was given, otherwise the default size is chosen in a system-dependent fashion to optimize virtual memory allocation. It is usually not necessary to create a larger region than the default size. Note that if you specify **:size** and not **:region-size**, the area will have exactly one region.

:representation
The type of object to be contained in the area's initial region. The argument to this keyword can be **:list**, **:structure**, or a numeric code. If this option is specified, an initial region is created. Otherwise, no region is created until you cons something.

Examples of **make-area**:

```
(make-area :name '*foo-area*
           :gc ':dynamic)

(defvar *bar-area*
       (make-area :name '*bar-area*
                  :gc ':ephemeral
                  :capacity 100000
                  :capacity-ratio 0.75
                  :n-levels 3))
```

describe-area *area* *Function*

area can be the name or the number of an area. Various attributes of the area are printed.

sys:area-list *Variable*

The value of **sys:area-list** is a list of the names of all existing areas. This list shares storage with the internal area name table, so you should not change it.

sys:%area-number *address* *Function*

Returns the number of the area of *address*, or **nil** if it is not within any known area. *address* is either an object whose memory address is used, or an integer used directly.

sys:%region-number *address* *Function*

Returns the number of the region of *address*, or **nil** if it is not within any known region. *address* is either an object whose memory address is used, or an integer used directly. (This information is generally not very interesting to users; it is important only inside the system.)

sys:area-name *area* *Function*

Given an area number, returns the name. This "function" is actually an array.

See the function **cons-in-area**. See the function **list-in-area**. See the function **room**.

Interesting Areas

This section lists the names of some interesting areas and explains their use in the system. Many other less interesting areas exist. To see all the existing areas in your system, select the [Areas] option to Peek.

sys:working-storage-area *Variable*

This is the normal value of **sys:default-cons-area**. Most working data are consed in this area.

sys:permanent-storage-area *Variable*

This area is to be used for "permanent" data, that (almost) never becomes garbage. Unlike **working-storage-area**, the contents of this area are not continually copied by the garbage collector; it is a static area.

sys:pname-area *Variable*

Print-names of symbols are stored in this area.

sys:symbol-area *Variable*

This area contains most of the interned symbols in the Lisp world.

si:pkg-area *Variable*

This area contains packages, principally the hash tables with which **zl:intern** keeps track of symbols.

sys:compiled-function-area *Variable*

Compiled functions are put here by the compiler.

sys:property-list-area *Variable*

This area holds the property lists of symbols.

sys:stack-area *Variable*

This area contains the control, binding, and data stacks of stack groups. Each process uses a portion of this area.

The sys:reset-temporary-area Feature

Some programs use the dangerous **sys:reset-temporary-area** feature to deallocate all Lisp objects stored in a given area. Use of this technique is not recommended, since gross system failure can result if any outstanding references to objects in the area exist.

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

Memory Mapping Tools

Several functions are provided to allow you to apply an operation to entire regions or areas, to objects within these, and so on.

The general philosophy is that a mapping routine is called, possibly with one or more predicates, a function to apply, and additional arguments to that function. The function (not the mapping routine) is called with some arguments based on the mapping routine's contract, followed by any additional arguments supplied for it. This is similar to the **:map-hash** and **:modify-hash** philosophy of hash tables. (Lexical scoping removes most needs for the additional-arguments feature.)

Predicates control what areas and/or regions the mapping routine considers. The defined names start with **si:area-predicate-** and **si:region-predicate-**. If **nil** is supplied in lieu of the predicate, then the default predicate is used. You are free to define your own routines that select specific qualities of areas or regions.

Area and Region Predicates

These predicates identify qualities of specific areas or regions within areas.

si:area-predicate-all-areas *area* *Function*

This predicate returns non-**nil** for all areas. This is *not* the default predicate.

si:area-predicate-areas-with-objects *area* *Function*

This function returns non-**nil** for areas that contain objects. It is the default area predicate. There is at least one area (**sys:page-table-area**) that does not contain objects and is therefore not of interest to users.

si:region-predicate-all-regions *region* *Function*

This predicate returns non-**nil** for all regions. It is the default region predicate.

si:region-predicate-structure *region* *Function*

This predicate returns non-**nil** for regions that contain structures (as opposed to lists).

si:region-predicate-list *region* *Function*

This predicate returns non-**nil** for regions that contain lists (as opposed to structures).

si:region-predicate-not-stack-list *region* *Function*

This predicate returns non-**nil** for all regions (list and structure) except those of type "stack list" (for example, control stacks).

si:region-predicate-copyspace *region* *Function*

This predicate returns non-**nil** only for regions in copyspace. It might be useful for determining what is (or was) transported to copyspace.

Mapping Routines

These are the routines that apply a designated function to designated areas or regions. In these routines, if *other-function-args* are supplied, they are passed along to the supplied function as additional arguments.

si:map-over-areas *area-predicate function* &rest *other-function-args* *Function*

For each area that satisfies *area-predicate*, *function* is called with the area number followed by *other-function-args*.

For example, the following form invokes **describe-area** on all areas:

```
(si:map-over-areas #'si:area-predicate-all-areas #'describe-area)
```

si:map-over-regions-of-area *area region-predicate function* &rest *other-function-args* *Function*

For each region in *area* (an area number) that satisfies *region-predicate*, *function* is called with the region number followed by *other-function-args*.

For example, the following form prints the names of all compiled functions in **sys:compiled-function-area**:

```
(defun print-compiled-function-names ()
  (si:map-over-regions-of-area
    sys:compiled-function-area
    #'si:region-predicate-structure
    #'(lambda (region-number)
     (let* ((origin (sys:region-origin region-number))
         (free (+ origin (sys:region-free-pointer region-number))))
       (si:scanning-through-memory scan1 (origin free)
          (loop for address = origin then (+ address object-size)
              while (< address free)
              do (si:check-memory-scan scan1 address)
              as object = (%find-structure-header address)
              as object-size = (%structure-total-size object)
              when (typep object 'compiled-function)
                do (print (si:compiled-function-name object))))))))
```

A better way to do it, since **si:map-over-objects-in-area** takes care of the memory scanning, is as follows:

```
(defun print-compiled-function-names-2 ()
  (si:map-over-objects-in-area
    sys:compiled-function-area #'si:region-predicate-structure
    #'(lambda (ignore ignore header ignore ignore)
       (when (typep header compiled-function)
         (print (si:compiled-function-name header))))))
```

si:map-over-regions *area-predicate region-predicate function* &rest *other-function-args* *Function*

For each region that satisfies *region-predicate* and is in an area that satisfies *area-predicate*, *function* is called with the area number and region number followed by *other-function-args*.

For example, the following form prints all region numbers, with the name of the area:

```
(si:map-over-regions
    nil nil
    #'(lambda (area-number region-number)
        (print (list (area-name area-number) region-number))))
```

There is a similar set of mapping functions that map over objects (structures and lists). In addition to possible area and region arguments, the supplied functions are passed four other arguments:

address	A fixnum giving the virtual memory address where the system started scanning to find the extent of the object.
header	The object itself, for example, an array, compiled function, list, or closure.
leader	A locative to the base of the structure. Under most circumstances, the address portion of the leader is the same as the address. The header and leader do not necessarily point to the same location; the header sometimes points to the middle of an object, as with compiled functions.
size	The size of the object in words.

Most applications are only interested in the header (object) and, possibly, the size. The address and leader are usually ignored. Area number and region number, for those mapping routines that supply them, are usually ignored as well.

si:map-over-objects-in-region *region-number function* &rest *other-function-args* *Function*

For each object in *region-number*, *function* is called with the address, the header, the leader, and the size, followed by *other-function-args*.

si:map-over-objects-in-area *area-number region-predicate function* &rest *other-function-args* *Function*

For each object in each region in *area-number*, where the region satisfies *region-predicate*, *function* is called with the region number, the address, the header, the leader, and the size, followed by *other-function-args*. For an example: See the function **si:map-over-regions-of-area**.

si:map-over-objects *area-predicate region-predicate function* &rest *other-function-args* *Function*

For each object in each region that satisfies *region-predicate*, in an area that satisfies *area-predicate*, *function* is called with the area number, the region-number, the address, the header, the leader, and the size, followed by *other-function-args*.

Additionally, there is a technique for interacting with the paging system to avoid excessive page faults while scanning forward through a known section of virtual

memory. The object-scanning routines use this technique, which nearly eliminates page faults on the objects (but not necessarily on data pointed to by the objects).

si:scanning-through-memory *identifier-symbol* (*starting-address limit-address* &optional (*pages-per-whack* **16**)) &body *body* *Function*

The *body* is executed normally. The *starting-address* is the address where scanning begins. The *limit-address* is the (exclusive) address where scanning ends.

The argument *pages-per-whack*, default 16, is the number of pages to page out and in when prefetching needs to be done. The slower the rate at which memory is scanned (for example, when looking at many words or spending a lot of time working on each section), the smaller *pages-per-whack* can be, because the disk will be able to keep up. The faster the scanning rate (for example, when counting the number of objects), the larger *pages-per-whack* can be, to avoid taking page faults on pages not quite paged in. *pages-per-whack* should not be greater than about 32, or else the program will spend time waiting for the disk queue to empty before it can queue all the page transfers.

identifier-symbol identifies this set of parameters. This allows correct nesting of **si:scanning-through-memory** macros. *identifier-symbol* is not evaluated, so it must not be quoted.

si:check-memory-scan *identifier-symbol current-address* *Function*

The *identifier-symbol*, an unevaluated symbol, matches the identifier symbol of a lexically visible **si:scanning-through-memory**. The *current-address* is the next address the code is about to use. Each time the address advances by *pages-per-whack*, the paging system pages out previous addresses and pages in future addresses. (See the function **si:scanning-through-memory**.)

The Garbage Collector Facilities

Principles of Garbage Collection

It is fundamental to the nature of Lisp that programs and systems allocate memory dynamically and in large amounts. (The allocation of memory for a basic list element, or *cons*, or for any other purpose, is called *consing* for the purpose of this discussion and in most other writings on Lisp.) Even with the large amount of virtual memory on a Symbolics computer, it is possible for a program to use it all up. At this point the machine halts and must be rebooted. This event can always be delayed, almost indefinitely, if the underlying system can reclaim memory that is unused.

Objects that are no longer in use, with no references from other objects, are termed *garbage*. Garbage is distinguished from *good objects* or *good data* by the fact that it no longer serves any purpose in the current Lisp world. For example, if the car of a cons is changed from object A to object B, and there are no other

references to A, then A is garbage. Objects in the Genera environment can be said to have a *lifetime,* which means how long the object remains "good". Three lifespans are distinguishable:

Static	Object will probably never become garbage. Example: standard system functions.
Dynamic	Object will probably become garbage eventually. Example: lines in editor buffers.
Ephemeral	Object will probably become garbage very quickly. Example: intermediate structure generated by the compiler.

You can control the garbage collection status of your own areas with the **make-area** function.

Garbage collection (GC) involves these three steps:

- *Scavenging* virtual memory, that is, periodically sifting through areas of memory, separating good objects from the garbage
- *Transporting* good objects to a safe place
- Reclaiming the memory occupied by garbage

Several strategies for garbage collection exist. Some allow you to continue doing other work and some do a more complete job but require additional machine resources for some period of time.

Garbage collection need not be used at all. However, it cleans up after computations and allows you to run for longer periods of time between cold boots. It should be used either when you are running a program that allocates large amounts of virtual memory (where the total allocated might exceed the amount of free memory in a cold-booted system) or when the total allocations of many programs might, over a relatively long period of time, exceed the capacity. In either case, garbage collection is a strategy aimed primarily at preserving the state of an operating Lisp world as long as possible and avoiding a cold boot.

There are three basic modes of garbage collection, each with some variations possible:

- *Incremental garbage collection* works in parallel with other processes in the system, allowing you to continue working while it is in progress. This mode is based on *incremental copying*, so called because objects are copied one at a time and there is relatively little effect on the user's interaction with the system. *Dynamic-object garbage collection* incrementally collects garbage in all *nonstatic* areas of memory. *Ephemeral-object garbage collection* incrementally collects garbage, concentrating on specific parts of memory that are known to contain short-lived objects. Both kinds of incremental operation ignore *static* areas of memory that change slowly and so are unlikely to contain garbage. For an explanation of static memory, see the section "Theory of Operation of the GC Facilities".

- *Nonincremental, or immediate, garbage collection* takes less free memory and less total processor time to work successfully than does the incremental mode. Nonincremental garbage collection is normally done with the Start GC :Immediately command or with the **gc-immediately** function, although those directives still ignore static areas. These directives allow no other work to be done by the process running it, although other processes are still scheduled. In most cases, though, immediate garbage collection places a heavy enough burden on the machine that other processes are not useful while it is operating. The immediate garbage collection invoked by the function **si:full-gc** deals with static areas.

- *In-Place garbage collection* is similar to immediate (nonincremental) garbage collection, but uses a fundamentally different algorithm for storage reclamation. Because of this, the virtual memory (paging space) required for GC is reduced. However, In-Place Garbage Collection is much slower, completely noninterruptable, and results in less optimal paging behavior than normal Immediate Garbage Collection. In-Place Garbage Collection is typically used only for dynamic objects, but it may be used to reclaim static objects as well.

Note: Areas of memory can be specified as being static with the function **make-area**.

Invoking the Garbage Collection Facilities

This section explains how to invoke the various garbage collection facilities. For more information on garbage collection in Genera, see the section "Theory of Operation of the GC Facilities".

Running with No GC

Maximum program speed is usually achieved by running with no garbage collection at all, although the machine will run out of virtual memory much faster. Running with no GC turned on is not recommended, since many system facilities assume that at least the Ephemeral GC is turned on. When your address space becomes low, GC notifications will be sent informing you that you are in danger of running out of memory space. Should your memory space be exhausted, your only recourse is to cold boot or add a new paging file. See the section "Add Paging File FEP Command".

Turning on the Ephemeral GC

If you would like to preserve the state of your machine much longer, with the least effect on performance, you should run with the *ephemeral-object garbage collector* (ephemeral GC) turned on. Some programs run better with the ephemeral GC turned on than with no GC turned on, because there is less paging. (See the section "Ephemeral-Object Garbage Collection".) The ephemeral GC is turned on by default. You can also turn it on with the Start GC :Ephemeral or Set GC Options commands. Another way of invoking the function (for example, from your init file) is to use **sys:gc-on** or **zl:choose-gc-parameters**.

Turning on the Dynamic GC

To preserve the virtual memory of your machine as long as possible, you should run with both the ephemeral and dynamic garbage collectors turned on. When a certain limit is passed, the dynamic GC is invoked. The dynamic GC slows performance of other programs for a period of usually 20 to 30 minutes. The dynamic GC requires more virtual memory for its own use than does the ephemeral GC. The dynamic GC is turned off by default, but it can be turned on by the command Start GC :Dynamic or by evaluating **sys:gc-on** with no arguments.

GC Needs Sufficient Space in Order to Run

All the garbage collection facilities require some additional virtual memory for their own use. Until the scavenging process is complete, running with a garbage collector can require up to twice as much space as running without a garbage collector (depending on how much of old space was garbage, compared to how much had to be copied). If you have been running without the garbage collector for a long time, you might not have enough room to successfully run the garbage collector and collect all the garbage. If the garbage collector is not operating, the system sends notifications as you run out of free memory space. See the section "Storage Requirements for Garbage Collection".

One solution is to turn on the garbage collector sooner, so it is left with enough space to operate. Another is to use **gc-immediately**. Another is to increase the size of the paging space on your local disk. See the section "Increasing Available Paging (Swap) Space".

Garbage collection can be optimized for particular applications by manipulating areas and their attributes. See the section "Areas". The [Areas] option of the Peek utility can be used to examine the garbage-collection attributes of particular areas; try it, and then click Left on **sys:working-storage-area**, for example.

The rest of this section is a listing of the functions and associated Command Processor commands for invoking the various garbage collection facilities.

Controlling GC from the Command Processor

In a Dynamic Lisp Listener window, you can use the Command Processor to control the operation of the GC facilities. The primary commands are Set GC Options and Start GC.

Set GC Options invokes a menu with which you can set 30 parameters dealing with garbage collection. See the section "Set GC Options Command".

Start GC invokes up to three types of garbage collection. The keyword arguments are :Immediately, :Ephemeral, and :Dynamic. See the section "Start GC Command".

GC Cleanups

GC Cleanups are functions that you can request to run to make garbage collections more successful. A typical GC cleanup releases pointers to objects that are

not strictly necessary for continued operation of an application, only convenient. Once the pointers are released, a subsequent GC can reclaim those objects. It is not appropriate for GC cleanups to delete useful information, such as mail or editor buffers.

Predefined GC cleanups exist that remove most pointers from the Lisp System to user objects. For example, the output histories of dynamic interactor windows are cleared. (These keep pointers to objects for mouse sensitivity.)

GC Cleanups are run using the "Start GC Command" with the keyword :Cleanup.

```
:Start GC (keywords) :Cleanup (Yes, No, or Ask [default Yes]) Yes
```

GC Cleanups are defined using **si:define-gc-cleanup**.

Garbage Collection Functions and Variables

zl:choose-gc-parameters *Function*

Activates a menu that you can use to control the operation of the garbage collector. Most of its features, including the ability to turn garbage collection on or off, are available elsewhere, but this is a single and more convenient interface. The variable **si:*gc-parameters*** is a list that defines the variables controlled by this function.

Another way to invoke this function is to type Set GC Options to the Command Processor. See the section "Set GC Options Command".

si:define-gc-cleanup *name* (&rest *fquery-args*) &body *body* *Function*

Defines a GC Cleanup with the specified name. *fquery-args* are never evaluated, and are used to implement the Start GC :Cleanup Ask option.

Example:

```
(defvar *big-database* nil)

(defun lookup-in-big-database (object)
  (unless *big-database*
    (setq *big-database* (recompute-big-database)))
  (lookup-object object *big-database*))

(si:define-gc-cleanup clear-big-database-for-gc ("Clear *BIG-DATABASE*")
  (setq *big-database* nil))
```

See the section "GC Cleanups".

sys:gc-on &rest *options* &key (*:ephemeral* **(cl:getf gc-on :ephemeral)**) (*:dynamic* **(cl:getf gc-on :dynamic)**) (*:query-p* **t**) *Function*

Turns on ephemeral and dynamic garbage collection facilities. The dynamic GC is off by default. The keywords **:ephemeral** and **:dynamic** select the type(s) of

garbage collection employed; the defaults are **:ephemeral t** and **:dynamic t** if no options are specified. If either option is specified, the other defaults to **nil**; this allows you to turn on one form of garbage collection and leave the other one off.

If you do not explicitly specify one of the keyword options, this function leaves that option in its previous state. For example:

```
(sys:gc-on :ephemeral t)
```

turns on the ephemeral GC and leaves dynamic GC in its previous state, either on or off. The function returns a list of four values which constitute the current state of the gc:

```
(:ephemeral ephemeral-is-on :dynamic dynamic-is-on)
```

where **ephemeral-is-on** and **dynamic-is-on** are either **t** or **nil**.

Note: the Command Processor command Set GC Options provides a more comprehensive facility for specifying many parameters of garbage collection. See the section "Set GC Options Command".

sys:gc-off *Function*

Turns ephemeral and dynamic garbage collection off.

sys:gc-on *Variable*

The value of this variable is a list specifying the status of the ephemeral and dynamic GCs. Here is an example:

```
(:ephemeral t :dynamic nil)
```

This indicates that the ephemeral GC in on and the dynamic GC is off. **sys:gc-on** is useful in finding out whether the garbage collector has turned itself off (as it does when not enough free space remains to be able to complete a copying garbage collection).

si:enable-gc-progress-notes &key *:dynamic :ephemeral* *Function*

Controls whether progress notes are displayed for garbage collection, and what their priorities are in relation to other progress notes. It takes two keyword arguments, **:dynamic** and **:ephemeral**, allowing you to control the display of progress notes for each type of garbage collection independently. The default for both dynamic and ephemeral GC is that progress notes are displayed when the current process is waiting for the GC to complete (**:foreground**). Omitting an argument leaves its value unchanged. Each argument can have one of the following values:

nil	Progress notes are never displayed for GC.
:foreground	Progress notes are displayed only when the current process is waiting for GC to complete. This is the default for both ephemeral and dynamic GC.

:background Progress notes are displayed for GC regardless of the GC's effect on the current process.

:override Like **:background**, but causes GC progress notes to override all other progress notes in the status line display. In **:foreground** and **:background** modes, any other progress note overrides a GC progress note.

GC progress notes show the current internal state of garbage collection, and give little indication of the Garbage Collector's effect on the current process.

See also **tv:*show-system-internal-progress-notes*** , which controls progress notes for other internal processes.

Examples:

To suppress the display of progress notes for the ephemeral GC, and allow the dynamic GC to display progress notes when it is affecting the current process (the default state), you evaluate the following:

```
(si:enable-gc-progress-notes :ephemeral nil)
```

The following form suppresses the display of all progress notes for the GC:

```
(si:enable-gc-progress-notes :ephemeral nil :dynamic nil)
```

gc-immediately &optional *no-query* *Function*

Invokes nonincremental, immediate garbage collection. Nonincremental GC effectively takes over the system while it runs, not allowing user input. As a result, it takes less space and less total time than an incremental GC. The main advantage of nonincremental, immediate GC compared to incremental GC is that it requires less free space and hence can succeed where an incremental GC would fail because virtual memory was too full.

If *no-query* is not **nil**, **gc-immediately** commences garbage collection without asking any questions, regardless of how much space is available. If it is **nil**, and if an immediate garbage collection might require more space than the amount of free space, you are asked whether you want to proceed.

Another way to invoke this function is via the Start GC :Immediately command. See the section "Start GC Command".

You should usually call **gc-immediately** rather than **si:full-gc**. The difference is that **gc-immediately** does not lock out other processes, does not run various **si:full-gc** initializations, and does not affect static areas of virtual memory.

Suppose garbage collection has already started and that the flip has occurred but not all good data have been copied out of old space. **gc-immediately** then copies the rest of the good data but does not flip again.

si:gc-in-place &key *:static (:query-p* **si:*gc-in-place-default-query-p***) *Function*

Invokes the In-Place Garbage Collector. In-Place Garbage Collection requires less virtual memory (paging space) than Immediate Garbage Collection, but runs much more slowly, is completely non-interruptable, and results in a less optimal object ordering.

Normally, **si:gc-in-place** only collects garbage in dynamic areas. If the *static* argument is not **nil**, then all static and dynamic areas are subject to garbage collection. Note that this is different from **si:full-gc**, because it performs none of the optimizations of **si:full-gc** which typically release internal system references to static objects. Therefore, there is typically little gain in collecting static objects in this fashion.

If *query-p* is not **nil**, then the user is queried before commencing garbage collection. (If there is insufficient memory to execute an In-Place Garbage Collection, a warning and query is issued unconditionally.) The default for the *query-p* argument is the value of **si:*gc-in-place-default-query-p***, which is normally **t**. This variable may be set or bound as a user option to change the default querying behavior of In-Place Garbage Collection.

si:gc-in-place also accepts a keyword **:background** that, if set to **t**, inhibits some of the messages and warnings printed out by the command, and limits the length of those that it does print. **:background** defaults to **nil**.

Another way to invoke this function is via the Start GC :Immediately In-Place command. See the section "Start GC Command".

si:full-gc &key *system-release* *Function*

Garbage-collects the entire Genera virtual memory environment, including some static areas. However, because static areas change slowly and are not likely to contain much garbage, use **gc-immediately** or the command Start GC :Immediately instead. See the section "Start GC Command". **si:full-gc** leaves the garbage-collector facilities in the state that it originally finds them, that is, with the same dynamic and ephemeral option settings.

If you use **si:full-gc**, call it with no arguments. The option **:system-release** is reserved for use by Symbolics. **si:full-gc** does an immediate, complete, nonincremental garbage collection as a preparation for immediately saving a world.

si:full-gc performs these operations:

- Resets temporary areas.
- Sets up the static areas to be cleaned up.
- Flips.
- Scavenges and flushes oldspace.
- Makes static areas static again.

It is not useful to perform an Incremental Disk Save (IDS) after running **si:full-gc**. Perform a complete disk save, instead.

Note: The Command Processor command Optimize World is the preferred high-level interface to the functions **si:full-gc**, **si:reorder-memory**, and **si:optimize-compiled-functions**. See the section "Optimize World Command".

Using the Initialization Lists invoked by si:full-gc

Two initialization lists, accessed through the **full-gc** and **after-full-gc** keywords to **add-initialization**, are run by **si:full-gc**. See the section "Introduction to Initializations".

si:full-gc runs the forms on the **full-gc** initialization list and then garbage-collects without multiprocessing (inside a **without-interrupts** form). The machine essentially "freezes" and does nothing but garbage collection for the duration. This operation takes 20 minutes or more, depending on the size of the world. After the garbage collection is completed, and before it reenables scheduling and returns, **si:full-gc** runs the forms on the **after-full-gc** initialization list.

full-gc is a system initialization list. You can add forms to it by using the **:full-gc** keyword in the list of keywords that is the third argument of **add-initialization**. The **full-gc** initialization list is run just before a full garbage collection is performed by **si:full-gc**. All forms are executed without multiprocessing, so the evaluation of these forms must not require any use of multiprocessing: they should not go to sleep or do input/output operations that might wait for something.

Typical forms on this initialization list reset the temporary area of subsystems and make sure that what is logically garbage has no more pointers to it.

Theory of Operation of the GC Facilities

This section describes the theory and terminology of garbage collection (GC) on Symbolics computers.

Dynamic and Static Spaces

The garbage collector treats the machine's virtual memory as if it were divided into three spaces: *static*, *dynamic*, and *free* space.

Static space — The parts of memory in which relatively permanent objects are allocated. Objects allocated in static space are not likely to become garbage; examples are the standard system functions and other objects that are likely to be referenced throughout the lifetime of a particular program or application. Static areas are ignored by all forms of garbage collection except **si:full-gc**.

Dynamic space — The parts of memory in which user programs and other programs allocate most of their objects are collectively called dynamic space. Objects allocated in dynamic space are likely to

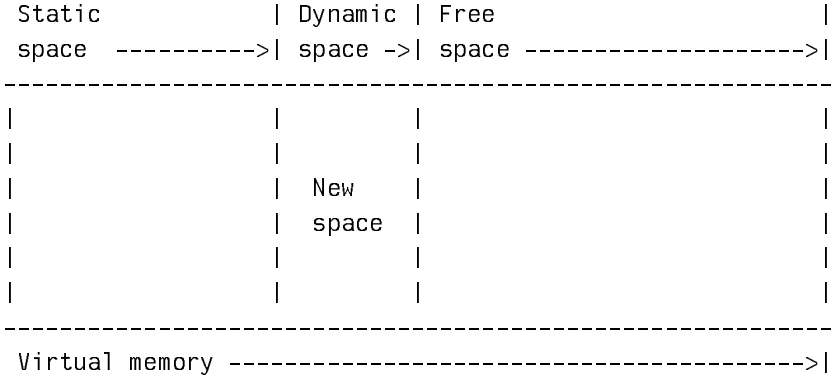
become garbage at some point, and all versions of garbage collection except **si:full-gc** pay exclusive attention to dynamic space. Dynamic space is further subdivided by the garbage collector into *old*, *new*, and *copy* spaces. (In addition, ephemeral *levels* are part of dynamic space; see the section "Ephemeral-Object Garbage Collection".)

Free space The unused space in paging files on the disk.

This diagram shows what the Genera virtual memory space looks like before the first garbage collection. This chapter contains a number of similar diagrams that show the division of Genera's virtual memory space into subspaces. In all these diagrams, keep in mind that these subspaces are not physically contiguous, but are rather scattered around in virtual memory. We show them as contiguous in order to simplify the visual presentation. In addition, the proportions of the spaces in the diagram are not necessarily true to scale.

When a machine has been cold-booted and used only slightly such that no garbage collection has yet started, virtual memory is divided into three spaces. Static space contains system functions and other long-lived objects. Dynamic space contains only newly-created objects and is therefore called *new space*. In a pristine system, all objects are allocated in new space. *Free space* is unused space in paging files on disk. The first diagram shows virtual memory space before the first garbage collection.

Before the first garbage collection:



Note that these spaces do not correspond directly to areas. All spaces can exist within a given area, but the area specifies the space in which its newly created objects reside. See the section "Areas".

Flipping

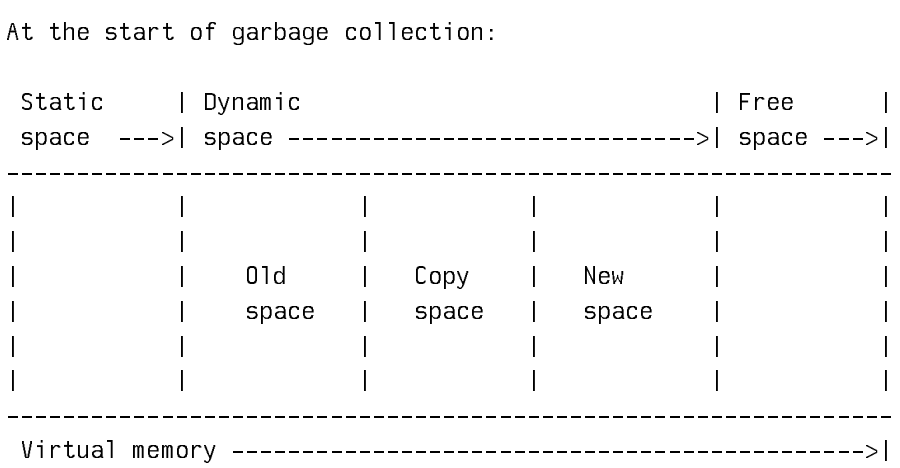
When the first garbage collection starts, two more subspaces are created within dynamic space: *old space* and *copy space*.

- Old space is the portion of dynamic space that previously was new and copy spaces and can still contain valid objects. The scavenger is looking for good ob-

jects here by perusing references in the current static and copy spaces. When the scavenger is finished, everything remaining in old space is garbage, so it is released to become free space.

- *Copy space*
  Copy space is the portion of dynamic space to which the transporter moves good objects found in old space.

Thus, when garbage collection starts, virtual memory looks like this:



Three agents are involved in garbage collection.

- A user program that creates new objects and so changes the contents of memory. This program is called the *mutator* for the purpose of this discussion.

- A *scavenger* program that reads through memory looking for references to objects that are in *old space*. It finds all accessible objects by starting at a *root set* of *static objects*, such as the hash table of all interned symbols, and recursively traces through the objects in the root set and the objects they reference. (The root set is a collection of data that is known to contain direct or indirect references to all valid data.) The scavenger runs during consing, during idle time, and (in the case of nonincremental garbage collection) in the user or garbage collector process.

- A *transporter* program invoked when either the mutator or the scavenger refers to an object in old space. If the object actually is still in old space, it evacuates the object (moves it to *copy space*). If the object has already been moved, the program locates its incarnation in copy space by following a forwarding pointer from old space. (Note that objects are copied only once.) The transporter redirects its client objects to copy space in either case.

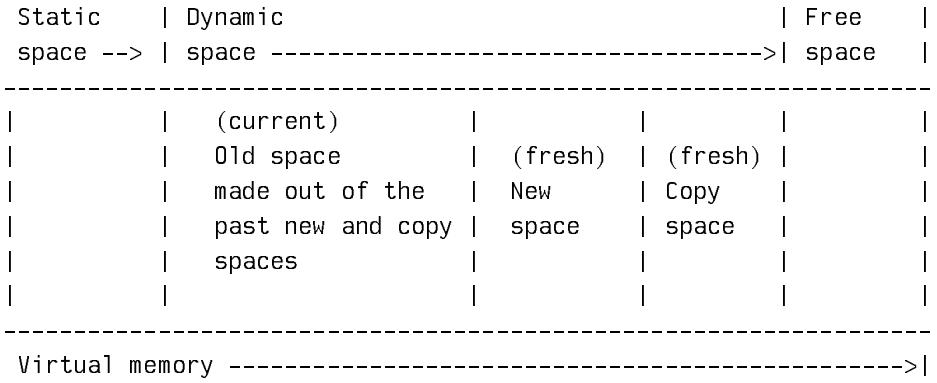
In subsequent garbage collections, the spaces are *flipped*. When a flip occurs, these three steps are taken by the system:

1. New space and copy space are lumped together to form a new version of old space. (This old space is then scavenged.)

2. A fresh new space is created; new objects will be allocated here while garbage collection of old space is in progress.

3. A fresh copy space is created; this space will receive copies of objects evacuated from old space. When an object is evacuated from old space, its incarnation there is replaced by a forwarding pointer that addresses the object's incarnation in copy space.

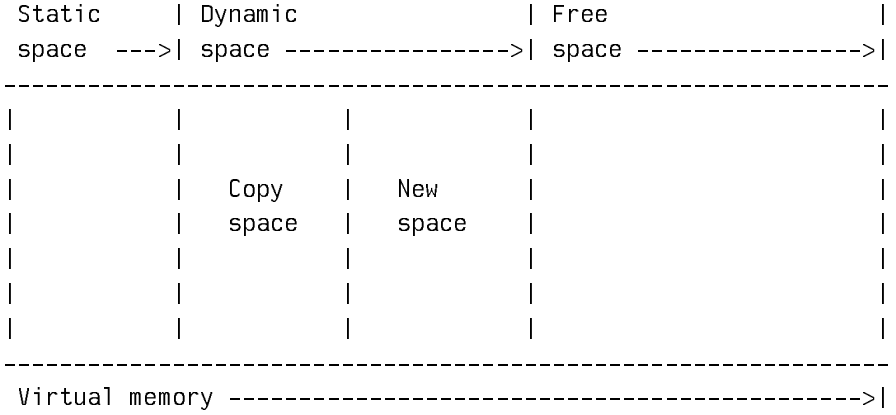
A GC causes the spaces to flip:



Once all good objects have been evacuated from old space to copy space, old space contains only garbage. Old space's memory is then reclaimed by the garbage collector, turns into free space, and becomes available for assignment to new space. Another flip can occur any time after old space has been reclaimed.

After garbage collection the spaces look like this:

After garbage collection:



The dynamic GC flips when it estimates that a large portion of the remaining free virtual memory will be needed for its own use.

A nonincremental garbage collection requires less virtual memory than an incremental one because the mutator is prevented from allocating new storage (consing) while the garbage collector is operating. See the section "Storage Requirements for Garbage Collection".

Ephemeral-Object Garbage Collection

Ephemeral-object garbage collection is a unique hardware-assisted incremental garbage collection method in which scavenger agents can pay special attention to short-lived (*ephemeral*) objects. A typical example of an ephemeral object is the intermediate structure generated by the compiler.

The ephemeral GC is effective on any area having the **:gc :ephemeral** characteristic as specified by **make-area**. Your **sys:working-storage-area** has the ephemeral characteristic by default. Since working storage is the initial value of ***default-cons-area***, objects created with no area specification are subject to ephemeral-object garbage collection while it is turned on.

The overall effects of the ephemeral GC are as follows:

- All objects created in ephemeral areas while the ephemeral collector is operating are considered ephemeral objects.

- The ephemeral-object garbage collector has means of tracking ephemeral objects, to avoid having to scan all of virtual memory for possible references to them.

- Garbage collection tends to increase the locality of objects and their references, so that ephemeral objects and their references are likely to be concentrated on relatively few pages.

- The above factors combine to dramatically reduce the amount of paging the garbage collector must do to find and process garbage, compared with dynamic GC, which operates on all of dynamic space rather than just the ephemeral portion of it. See the section "Dynamic and Static Spaces".

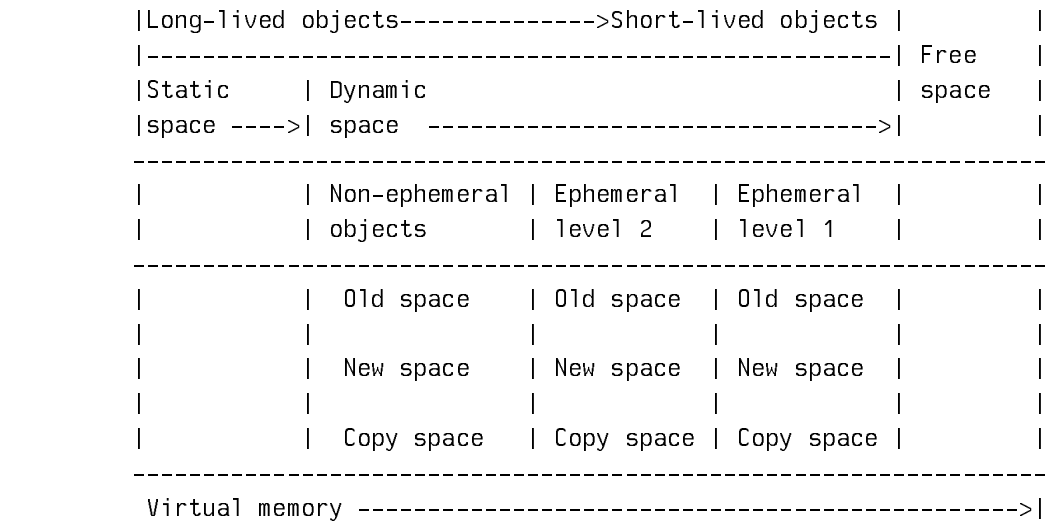
- When the dynamic (nonephemeral) objects are eventually garbage-collected, dynamic space contains less garbage than would otherwise be the case.

Ephemeral Levels

The ephemeral-object GC introduces the concept of ephemeral *levels*, which are subdivisions of a particular area of memory. The advantage of having several levels of ephemeral GC is that the garbage collector spends most of its time dealing with only a small fraction of the total number of objects and total storage in the system, namely, with the ephemeral levels. This greatly decreases paging, total time to complete a garbage collection, and the amount of virtual memory that has to be committed to the garbage collector's use.

Each ephemeral level contains its own old, new, and copy space. The diagram below is a schematic representation of two ephemeral levels in dynamic space, along with a non-ephemeral part of dynamic space. Level 1 of ephemeral space contains the shortest-lived objects in dynamic space.

For convenience in the diagram, we show the old, new, and copy spaces vertically in each ephemeral level and in dynamic space. Again, this diagram does not represent the actual physical allocation of these spaces in virtual memory.



Consider, for example, the following, abbreviated output of `(describe-area working-storage-area)`:

```
Area #4: WORKING-STORAGE-AREA has 15 regions,
max size 2000000000, region size 340000 (octal):
  First ephemeral level: 2 regions, capacity 196K, 416K allocated, 122K used.
  Second ephemeral level: 3 regions, capacity 98K, 336K allocated, 148K used.
  Last (dynamic) level: 10 regions, 2448K allocated, 2216K used.
.
.
.
```

The "first" ephemeral level is the one in which all new objects in this area are created. It, like other ephemeral levels, has a capacity in words. When the capacity of the first level is reached, the ephemeral level is flipped, and any objects that are not proven to be garbage are evacuated to the next level by the usual incremental garbage collection methods.

The levels after the first are flipped only when the first level is flipped. (You can see, in this example, that the second level has exceeded its capacity, because it is waiting for the first level to flip.)

When the last (dynamic) level has received enough objects from the ephemeral levels, it is flipped and garbage collected by the dynamic GC as usual for dynamic areas. It has no capacity in the sense of an ephemeral level because the decision

to flip in the dynamic GC is based on different principles. See the section "Storage Requirements for Garbage Collection".

The output of the function **zl:gc-status** or the command Show GC Status includes one line for each ephemeral level that exists.

By default, **sys:gc-on** or the Start GC command enables the ephemeral collector along with dynamic-object garbage collection. The area **sys:working-storage-area** has the ephemeral characteristic and two ephemeral levels by default, so the ephemeral feature is effective even if you do not explicitly manipulate areas.

You can get additional insight into the concept of levels by experimenting with the following features:

- Using the function **zl:choose-gc-parameters** or the Command Processor command Set GC Options, select the options for reporting the activity of the ephemeral GC.

- Using the [Areas] option of the Peek utility, examine the GC characteristics of particular areas, such as, for a start, **sys:working-storage-area**. (Point at this area and click left to see the details.) The **describe-area** function can be used for the same purpose.

- Using the **:capacity**, **:capacity-ratio**, and **:n-levels** options of the **make-area** function, you can define the number of ephemeral levels for specific areas. With programs that create mostly ephemeral objects, it can be possible to extend the length of a session considerably, by adding additional ephemeral levels.

How Garbage Collection Improves Locality of Reference

Locality of reference means that objects and their references (or more generally, any pieces of related information), are located near each other, that is, located at nearby addresses in virtual memory. When this is true, the paging system can avoid *thrashing*: swapping many pages in and out of main memory in order to access relatively few data.

One way to improve locality of reference is to use areas. See the section "Areas". This technique can greatly improve locality of reference in programs that allocate virtual memory in large amounts and for specific purposes. Areas are especially useful when the objects allocated are static. In this case, the objects are left completely alone by most kinds of garbage collection.

Another way to improve locality of reference in Genera is to use the garbage collection facilities. This improves locality of reference through dynamic memory space, including the **sys:working-storage-area**. How does GC improve locality of reference?

- First, the operation of copying good objects to a separate space (copy space) compacts objects on virtual memory pages. Good objects are not interleaved with garbage.

- Second, the use of separate new and copy spaces improves locality further, because new objects are likely to be "less related" to older ones, and the two are not interleaved.

- Finally, the garbage collector uses a technique called *approximately depth-first copying*, which improves locality in typical programs.

Approximately depth-first copying works as follows:

1. The scavenger concentrates on the most recent, partially filled page in copy space, looking for references to old space (that is, looking for objects that might have to be evacuated from old space).

2. If no such objects are found, or if the last page in copy space is full already, the scavenger looks at the first (lowest-addressed) page in copy space that has not yet been scavenged. It proceeds from this page forward, page by page, looking for old-space references.

3. As soon as an object is transported from old space to copy space, the scavenger returns its attention to the last page in copy space and considers the objects referenced by the newly transported object.

The effect is that object references and the corresponding objects tend to fall on the same page in virtual memory.

In-Place Garbage Collection

In-Place garbage collection is similar to immediate (nonincremental) garbage collection, but uses a fundamentally different algorithm for storage reclamation. As its name implies, In-Place garbage collection does not copy referenced objects into copy space and then dispose of oldspace memory. Instead, a modified *mark-sweep* algorithm is used to mark referenced objects and reclaim intervening storage by compacting good objects to the bottom of each region. Since there are never two copies of an object during the garbage collection, the virtual memory (paging space) required for GC is reduced.

Comparing In-Place garbage collection to Immediate garbage collection:

- *In-Place Garbage Collection requires less memory.* Because objects are not copied, some of the memory requirements of garbage collection are alleviated. However, In-Place garbage collection does require some memory for its own internal tables, and its execution can cause otherwise unmodified pages to be modified, moving them from the world load file into newly allocated storage in the paging file.

- *In-Place Garbage Collection is much slower.* Mark-sweep algorithms do not perform well in demand-paged environments compared to copying algorithms. Consequently, In-Place garbage collection takes 3 to 10 times longer than Immediage garbage collection.

- *In-Place Garbage Collection is non-interruptible.* Mark-sweep algorithms are not incremental, so the scheduler and all I/O are disabled during garbage collection. Additionally, storage is usually inconsistent during garbage collection, so the machine cannot be warm-booted. You can, however, add paging files from the FEP and continue if you run out of virtual memory during the GC.

- *In-Place Garbage Collection results in less optimal object ordering.* Copying algorithms allow lisp objects to be ordered more optimally for paging performance. Since In-Place garbage collection does not change the order of objects in memory, the older, less desirable object orderings are preserved.

Symbolics recommends In-Place garbage collection only for situations where execution speed and interaction are not important, for example when there is insufficient disk space for normal garbage collection, and the garbage collection can be performed overnight. Where possible, normal garbage collection or selective garbage collection is recommended.

In-Place garbage collection runs in four phases, as shown in Figure !:

1. *Mark.* Starting with the root set, referenced objects are marked and those objects referenced by referenced objects are marked, until transitive closure over all objects referenced by the root set is achieved.

2. *Build Relocation Tables in unused storage.* Tables for relocation of the pointers and referenced objects are constructed in unused storage.

3. *Relocate pointer references to new addresses.* Pointer references are relocated to reflect the new locations of the objects.

4. *Shift objects downward to new addresses.* The objects are shifted to their new locations.

Storage Requirements for Garbage Collection

Interpreting the Output of Show GC Status

Besides showing the state of the ephemeral GC levels, the output of the Show GC Status command (or **zl:gc-status** function) shows the storage requirement for dynamic garbage collection, in the form of a *committed guess*. For example, suppose the command reports the following information:

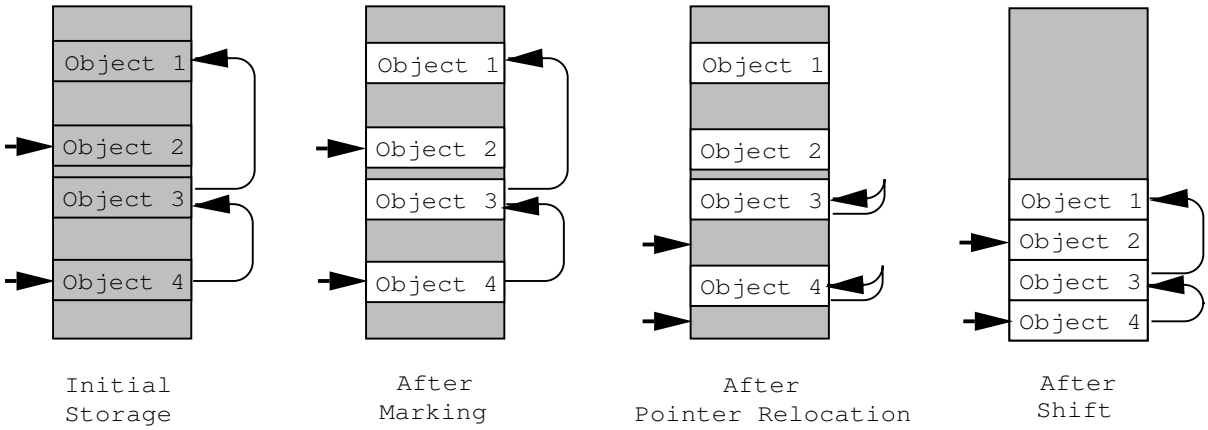


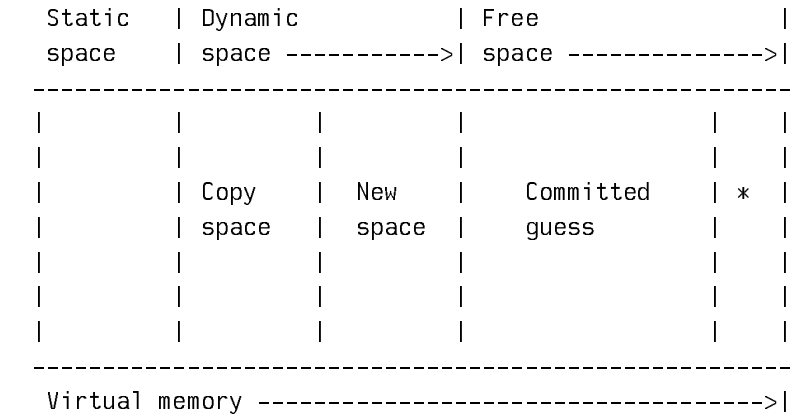
Figure 32. How In-Place Garbage Collection Works

```
Status of the dynamic garbage collector:                    Off
Dynamic (new+copy) space 11,849,700.  Old space 0.  Static space 18,311,838.
Free space 31,741,440.  Committed guess 29,795,586, leaving 1,683,710 to use before
your last chance to do Start GC :Dynamic without risking running out of space.
There are 9,683,726 words available before Start GC :Immediately might run out of space.
Doing Start GC :Immediately now would take roughly 54 minutes.
There are 31,342,736 words remaining before Start GC :Immediately In-Place
might run out of space.  (The current estimate of its
memory requirements is 398,704 words.)
Minimum scavenging remaining 70,178, maximum possible 70,178.
Free space 31,741,440 (of which 0 might be needed for copying).

Garbage collector process state: Await ephemeral full
Scavenging during cons: On, Scavenging when machine idle: On
The GC generation count is 414 (1 full GC, 4 dynamic GC's, and 409 ephemeral GC's).
Since cold boot 26,504,876 words have been consed, 19,023,571 words of garbage have
been reclaimed, and 29,823,957 words of non-garbage have been transported.
The total "scavenger work" required to accomplish this was 124,712,353 units.
Use Set GC Options to examine or modify the GC parameters.
```

The *free space* (or *free paging space*) is the total amount of unused space (in words, not bytes) allocated to paging on the local disk(s). If garbage collection is turned off, free space is the amount available for new objects. The free space minus the committed guess, minus a relatively small amount, should equal the amount left before flipping.

The committed guess is the garbage collector's estimate of the amount of free storage it will need for copying and for new consing. It is accurate for compute-bound programs, on which most of the underlying assumptions are based. For in-

teractive programs, it is conservative because the garbage collector runs during idle time and so finishes more quickly.




```
* Indicates available free space before a flip.
```

The computation goes as follows, assuming that **si:gc-flip-ratio** = 1:

```
Dynamic (new+copy) space 184,000.  Old space 0.  Static space 7,500,000.
Free space 17,000,000.  Committed guess 11,677,500, leaving 5,322,500 to
  use before flipping.
```

If you cons 5.32 megawords of dynamic space, in addition to the space you already have, and then the flip occurs, then at the instant the garbage collector completes (after it has copied all of old space but before old space is reclaimed), old space and copy space will each be 5.5 megawords. That accounts for 11 megawords; all but .184 megawords of that has to come out of your 17 megawords of free space.

To complete the garbage collection, the scavenger has to do 5.5 MWU (million work units) to copy 5.5 megawords from old space to copy space, plus 5.5 MWU to scan through that copy space looking for references to old space, plus 7.5 MWU to scan through static space looking for references to copy space, plus x MWU to scan through the x words of additional objects you might cons in static space during the garbage collection. (It has no way to distinguish these from objects that existed in static space before the garbage collection, so it can't take advantage of knowledge that objects created after the flip cannot contain references to old space; it does take advantage of this invariant for dynamic space, but not for static space). The total scavenger work to be done is therefore $18.5+x$ MWU. The rate at which the scavenger works is pegged to the rate of consing; the scavenger does 4 "work units" for each word consed. Thus the total consing during the garbage collection is $(18.5+x)/4$ megawords. In the worst case, all this consing will be in static space, hence $4x = 18.5+x$ or $x = 6.17$.

Thus you cons 5.32 megawords before the garbage collection and 6.17 megawords during the garbage collection.

The primary reason that nonincremental garbage collection (such as invoked by **gc-immediately** or Start GC :Immediately) requires less memory is that consing is prohibited in the invoking process (the mutator cannot run).

To check the computation: at the instant the garbage collection completes, the total space occupied will be 5.5 megawords of old space, 5.5 megawords of copy space, 7.5 megawords of old static space and 6.17 megawords of new static space; total = 24.67. The total you have right now is .184 megawords of dynamic space, 7.5 megawords of static space, and 17 megawords of free space; total = 24.68. So, you can see that you have just enough free space to be able to cons 5.322 megawords, flip, cons 6.17 megawords more during the garbage collection, and reclaim old space, creating more free space, just as you exhaust the last bit of free space. This is what the committed guess is all about.

Of course, this is all based on worst-case assumptions. If some of dynamic space is garbage, so copy space is smaller than 5.5 megawords, or some of your consing before the flip is in static space (making old space smaller than 5.5 megawords), or some of your consing after the flip is in dynamic space (making the scavenger not have to work as hard), the garbage collection will complete with some free space left over. Also, scavenging during idle time makes the garbage collection complete sooner.

The GC includes some safety factors. The committed guess is increased by the constant 256 Kwords and the amount you can cons before the flip is decreased by an additional 256 Kwords (value of **si:gc-delta**). So, you lose about .5 megawords of consing.

```
Dynamic (new+copy) space 184,000.  Old space 0.  Static space 7,500,000.
Free space 17,000,000.  Committed guess 11,939,644, leaving 4,798,212 to
  use before flipping.
```

If you cons 4.8 megawords of dynamic space, in addition to the space you already have, and then the flip occurs, old space and copy space will each be 4.98 megawords at the instant the garbage collection completes. That accounts for 10 megawords; all but .184 megawords comes out of your 17 megawords of free space.

The scavenger has to do 4.98 MWU to copy 4.98 megawords from old space to copy space, plus 4.98 MWU to scan through that copy space looking for references to old space, plus 7.5 MWU to scan through static space looking for references to copy space, plus x MWU to scan through the x words of additional objects you might cons in static space during the garbage collection. The total scavenger work to be done is therefore $17.46+x$ MWU. Thus the total consing during the garbage collection is $(17.46+x)/4$ megawords. In the worst case, all this consing will be in static space, hence $4x = 17.46+x$ or $x = 5.82$. At the time the garbage collection completes, the total space occupied will be 4.98 megawords of old space, 4.98 megawords of copy space, 7.5 megawords of old static space and 5.82 megawords of new static space; total = 23.23. You will have 1.4 megawords of free space left over. This provides a cushion against the effects of storage fragmentation caused by the use of multiple areas.

Swap Migration and Garbage Collection

When the Garbage Collection facilities estimate the amount of memory required, they do not take into account a phenomenon called *swap migration*. Because of this, in certain situations the actual memory required for a garbage collection can

be much higher then estimated, and running the garbage collector may cause your machine to run out of memory. This estimation error can only happen when

- you are about to run a *dynamic* or *In-Place* garbage collection,
- you have booted a world from your local machine, and
- you have not yet run a *dynamic* or *In-Place* garbage collection since you cold-booted.

There are two simple work-arounds for this problem:

1. Netboot your worlds.

2. Run one dynamic or In-Place garbage collection immediately or shortly after booting your world, while ample space remains.

The Causes of Swap Migration

All objects take up memory locations. In virtual-memory architectures such as on Symbolics computers, these objects are stored on disk when they are not actively in use, and copied into fast memory when they are actively in use. The amount of memory available is the amount of unallocated memory in paging files on your disk.

When you first boot a world on your local disk, the paging files are initially empty; that is, all the memory in the paging files is unallocated. When you create an object, memory is reserved in the paging file for that object. If an object is garbage collected, the memory is then freed and available for other objects.

If you reference an object which was saved in the world, then it is read from the world file, not the paging file. This works as long as the object is unmodified. However, if an object is modified, the object must be *migrated* into the paging file. (If it were stored back into the world file, then the change would be apparent the next time the world was booted, which would not be desirable.) This phenomenon is called *swap migration*. In brief, swap migration is the phenomenon where modifying an object will cause it to use up more memory.

The garbage collector can modify objects in two ways. First, it can copy or move an object subject to GC. Second, as a result of moving the object, it modifies all referencing objects to reference the new address of the object. Therefore, if a dynamic object or any referencing object is in the world file, a garbage collection will cause the object to be migrated to the paging file, where it will reduce the amount of free space available.

Swap Migration can occur even when objects are not modified, if you enable Load to Paging migration in the FEP. When FEP Load to Paging migration is enabled, objects are copied into the paging file whenever they are read from the world file, regardless of whether they have been modified. See the section "Enable Load to Paging Migration FEP Command" and also see the section "Disable Load to Paging Migration FEP Command".

Note that if you netboot, there is no local world file (other than the netboot core), so all objects exist in the paging file. Even if you boot locally, if you have already performed a dynamic or In-Place garbage collection, all world-load objects which could be modified by the garbage collector will have already been modified the first time. So in either of these cases, swap migration is not a concern.

Note that the effect of swap migration on *ephemeral* garbage collection is negligible, since most ephemeral objects become dynamic before a world is saved.

Why the Garbage Collector Estimates Do Not Take Swap Migration Into Account

Determining swap migration is tedious at best. Although the worlds distributed by Symbolics will show very little swap migration, applications vary widely in their use of static and dynamic objects. Scanning the entire world for all references to dynamic objects can take from ten minutes to over an hour. Even then, the estimate would not be accurate, since there are indeterminacies in the storage allocation process which are nearly impossible to predict.

Estimating swap migration may be possible in a future release of Genera. In Genera 8.0, a warning is printed by "Show GC Status" and "Start GC :Immediately In-Place" if swap migration could be a problem.

Controlling Garbage Collection

zl:gc-status *Function*

Prints statistics about the garbage collector. It prints different information depending on whether the scavenger is running or finished and how full virtual memory is.

Another way to invoke this function is via the Show GC Status command. See the section "Show GC Status Command".

```
(gc-status)

Status of the ephemeral garbage collector:                  On
First level of METERING:METERING-CONS-AREA: capacity 196K, 0K allocated, 0K
 used.
Second level of METERING:METERING-CONS-AREA: capacity 98K, 0K allocated, 0K
 used.

First level of DW::*EQL-DISPATCH-AREA*: capacity 98K, 256K allocated, 56K used.
Second level of DW::*EQL-DISPATCH-AREA*: capacity 49K, 0K allocated, 0K used.

First level of WORKING-STORAGE-AREA: capacity 196K, 448K allocated, 29K used.
Second level of WORKING-STORAGE-AREA: capacity 98K, 2048K allocated, 47K used.

Status of the dynamic garbage collector:                    On
```

```
Dynamic (new+copy) space 6,490,761.  Old space 0.  Static space 12,479,751.
Free space 26,574,848.  Committed guess 22,488,118, leaving 3,824,586 to use
 before flipping.
There are 9,779,900 words available before Start GC :Immediately might run out
 of space.
Doing Start GC :Immediately now would take roughly 33 minutes.
There are 26,574,848 words available if you elect not to garbage collect.

Garbage collector process state: Await ephemeral or dynamic full
Scavenging during cons: On, Scavenging when machine idle: On
The GC generation count is 328 (1 full GC, 2 dynamic GC's, and 325 ephemeral
 GC's).
Since cold boot 53,043,930 words have been consed, 45,867,153 words of garbage
 have
been reclaimed, and 11,658,295 words of non-garbage have been transported.
The total "scavenger work" required to accomplish this was 121,864,225 units.
Use Set GC Options to examine or modify the GC parameters.
```

In the **zl:gc-status** report, the free space figure minus the committed guess figure is approximately equal to the amount of memory available before flipping. (If the garbage collector were currently off, this field would show the amount of memory available before incremental garbage collection must be turned on, to avoid the risk of running out of space.)

Notice that a nonincremental garbage collection (**gc-immediately**) requires less memory, although it will run exclusively, in the invoking process, for a long time. An estimate of the time, which depends on the size of the world, is printed.

As shown here, when the garbage collector is on, the scavenger operates during consing and when the processor is idle (when no process wants to run). The operation of the scavenger is also signalled by the garbage collector's run bar; the left half of this bar, which appears under the package name on the machine's status line, blinks to indicate scavenging. The right half of the bar blinks when the transporter moves objects out of old space.

You could also turn off garbage collection at this point (with the Halt GC command or **sys:gc-off** function) and still have over 26 million words available before you ran out of virtual memory.

The "garbage collector process state" is the state of the process that starts a garbage collection when it is time (by flipping) and generally supervises the garbage collector.

si:print-gc-meters *Function*

Displays a history of garbage collection work done in the current world, including the number of times the transporter and scavenger were invoked, the time they consumed, their paging activity, and so on. It also shows statistics on the references handled by the garbage collector page table (GCPT) and the ephemeral space reference table (ESRT); these are, respectively, the ephemeral-object garbage

collector's tables of swapped-in and swapped-out pages that contain ephemeral objects.

si:inhibit-gc-flips *body ...* *Macro*

Prevents the ephemeral and dynamic garbage collectors from flipping within the body of the macro.

si:with-ephemeral-migration-mode *mode* &body *body* *Macro*

Controls what happens when ephemeral space is garbage collected and also determines the space in which new copies of ephemeral objects that survive garbage collection are created. Permissable *modes* include the following.

:dynamic	Put the copies in dynamic space.
:normal	Put the copies in the next ephemeral level or dynamic space if this is the last ephemeral level

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***.

The following variables control various aspects of the garbage collector's operation; all are accessible via the Command Processor command Set GC Options or the **zl:choose-gc-parameters** function.

si:gc-report-stream *Variable*

Specifies where to put output messages from the garbage collector.

Value	*Meaning*
t	Notifies you (default)
nil	Discards the output
stream	Sends output to the stream

si:gc-area-reclaim-report *Variable*

Controls reporting of reclaimed areas. If it is any of the values other than **nil**, each reclaimed area is reported individually.

Value	*Meaning*
nil	Does not report anything (default).
:dynamic	Reports only after dynamic garbage collection.
:ephemeral	Reports only after ephemeral-object garbage collection.
t	Reports after any kind of garbage collection.

si:gc-warning-threshold *Variable*

Controls the warnings to turn on the garbage collector. This warning-threshold indicates (when it is warning you for the first time) how close you are to the last safe point (in words of memory). After you have passed the last safe point, you cannot turn on the GC without probably running out of memory. The last safe point is arrived at by comparison to the amount of uncommitted free space.

When the storage manager notices that the amount of free space remaining has reached the threshold, it notifies you that you need to turn on the garbage collector before it is too late to do so. The default value is 1000000 (words). It is usually not necessary to change this value from the default.

si:gc-warning-ratio *Variable*

Controls how often (after the **si:gc-warning-threshold** has been passed) you see warnings that you need to turn on the garbage collector. Basically, this ratio is multiplied by the previous warning threshold to give a new warning threshold. For example, the default **si:gc-warning-ratio** is 0.75. With the default values for **si:gc-warning-threshold** and **si:gc-warning-ratio**, you would see warnings with 1000000, 750000, 562500, and 421875 words remaining, and so on. This variable has no effect if **si:gc-warning-interval** is set to **nil**, which is the default.

si:gc-warning-interval *Variable*

Contains the interval in 60ths of a second between repetitions of the same garbage collector warning; it applies only to reports that use the notification system. The rationale for this variable is that you can control how often you want to be bothered by such messages.

The default value is **nil**, which shuts off repetitious warnings. Each warning is given only once.

si:gc-flip-ratio *Variable*

Specifies when a flip takes place. When this number times the amount of committed free space (the "committed guess" reported by Show GC Status) is greater than the amount of free space, a flip occurs. The default value is 1.

The number can be less than 1. This would cause the garbage collector to wait longer before flipping at the risk of exhausting virtual memory if a larger fraction of dynamic space contains good objects than you expected. Rather than setting the ratio to a number less than 1, we recommend turning on the ephemeral-object garbage collector.

For a discussion of finer control over the onset of garbage collection: See the variable **si:gc-flip-minimum-ratio**.

si:gc-flip-minimum-ratio *Variable*

Contains a number that specifies when to turn the garbage collector off because memory is too full to allow copying anything. The default value is **nil**, which specifies that this ratio has the same value as **si:gc-flip-ratio**. Otherwise it should be a number less than **si:gc-flip-ratio**.

Putting 0.25 in **si:gc-flip-minimum-ratio** and 0.5 in **si:gc-flip-ratio** means that you believe that fewer than 25 per cent of the dynamic-space objects consed are good data and will need to be copied by the garbage collection. In spite of this, you want to flip when there is enough space to copy 50 per cent (half) of the objects. Thus, the flip ratio controls how often the garbage collector flips; the minimum ratio controls when it should get desperate.

The minimum ratio is most useful if you turn on **si:gc-reclaim-immediately-if-necessary**, to make the garbage collector do something useful when it is desperate. Even without that, it is useful if you would rather risk doing a garbage collection when there might not be enough memory left in preference to turning the garbage collector off, for example, when the machine is operating unattended and turning off the garbage collector would be guaranteed to make it exhaust memory.

Choosing good values for this variable is a matter of guesswork and experience with the particular application.

si:gc-reclaim-immediately *Variable*

When the value is **nil** (the default), the incremental (dynamic) garbage collector is not affected. When the value is not **nil**, then, in effect, an immediate garbage collection is performed as soon as the flip occurs.

si:gc-reclaim-ephemeral-immediately *Variable*

When the value is **nil** (the default), the ephemeral-object garbage collector is not affected. When the value is not **nil**, then, in effect, an ephemeral GC is performed as soon as the capacity of the first ephemeral level is exceeded.

si:gc-reclaim-immediately-if-necessary *Variable*

Controls whether the garbage collector starts nonincremental garbage collection or shuts down when space is running too low for incremental garbage collection. This variable is irrelevant when **si:gc-reclaim-immediately** is set because then the garbage collector always reclaims immediately, even if it does not need to.

The variable controls what happens when not enough free space remains to copy everything. When the value is **nil** (the default), it notifies you and turns itself off. For other values, it tries nonincremental garbage collection and shuts itself off only when it determines that nonincremental garbage collection is not guaranteed to work.

It is possible for so little space to remain that even a nonincremental garbage collection would exhaust virtual memory. The decisions about what would exhaust virtual memory depend on your prediction of the fraction of dynamic space that contains good (nongarbage) objects. (This is the value of **si:gc-flip-minimum-ratio**.)

si:gc-process-immediate-reclaim-priority *Variable*

Supplies the process priority at which nonincremental (immediate) garbage collection operates. Its default value is 5, which locks out other, computational processes. It is also accessible via the function **zl:choose-gc-parameters** and the command Set GC options. Note: This variable is not related to the **gc-immediately** function nor to the :Immediately option of the Start GC command.

si:gc-process-foreground-priority *Variable*

Sets process priority for the garbage collector while it is waiting to flip and in the process of flipping. Its default value is 5.

si:gc-process-background-priority *Variable*

This variable provides the priority (default 0) of the garbage collector process while it is reclaiming old space.

si:gc-flip-inhibit-time-until-warning *Variable*

Sets the reasonable time window for flipping. If flipping does not occur successfully during this time, the garbage collector notifies you about the problem. The time is expressed in 60ths of a second. The default is 1800 (30 seconds). Flipping cannot occur when some program (such as **maphash**) is running in an **si:inhibit-gc-flips** special form.

Strategy for Unattended Operation with the Garbage Collector

It is risky to leave very large compilations that do a lot of consing running unattended. You can set the following variables in order to control the assumptions

that the system makes about the amount of space needed or available. See the section "Controlling Garbage Collection".

si:gc-flip-minimum-ratio
si:gc-flip-ratio
si:gc-reclaim-immediately-if-necessary

More background information is available, to help you use these variables appropriately. See the section "Theory of Operation of the GC Facilities".

See the section "Principles of Garbage Collection".

Setting up GC Before Loading a Large System

Some people find it necessary to have garbage collection working in order to load large systems. Here are several recommended strategies.

- Before loading the system, turn on ephemeral-object garbage collection with the command `Start GC :Ephemeral` or with the form `(sys:gc-on :ephemeral t)`.
- After loading the system, do an immediate garbage collection with the with the command `Start GC :Immediately` or with the function **gc-immediately**.
- Do both the above.
- After loading the system, do a full garbage collection by calling **si:full-gc** with no arguments. Note, though, that **si:full-gc** does a lot of unnecessary work and disables multiprocessing, thus causing network connections to be lost. Then execute the command Optimize World in order to move things around in virtual memory so as to improve locality of reference and decrease paging.

Reporting the Use of Memory

The **room** function and variable allow you to examine the current use of physical and virtual memory in the machine. The current use of memory areas can also be examined with the Areas option of the Peek utility.

room &rest *args* *Function*

In CLOE, **zl:room** displays information concerning memory allocation and usage.

In Genera, **zl:room** displays the amount of physical memory in the machine, the amount of available virtual memory not yet filled with data (that is, the portion of the available virtual memory that has not yet been allocated to any region of any area), and the amount of "wired" physical memory (that is, memory not available for paging). Then it tells you how much room is left in some areas. For each area it tells you about, it prints out the name of the area, the number of regions that currently make up the area, the current size of the area in kilowords, and the

amount of the area that has been allocated, also in kilowords. If the area cannot grow, the percentage that is free is displayed.

(room) tells you about those areas that are in the list that is the value of the variable **room**. These are the most interesting ones.

(room *area1 area2...***)** tells you about those areas, which can be either the names or the numbers (applies to Genera only).

(room t) tells you about all the areas.

(room nil) In Genera, it does not tell you about any areas; it only prints the header. This is useful if you just want to know how much memory is on the machine or how much virtual memory is available.

In CLOE Runtime, it prints information on allocated storage for each data type that is listed, including the number of storage areas, number of bytes allocated, number of bytes used, and percentage used. If the number used reaches the number allocated, the next cons will cause a garbage collection if automatic GC is enabled. If automatic GC is disabled, or fails to free up enough storage of the given type, a new area will be allocated and added to the free storage for that data type.

room *Variable*

The value of **room** is a list of area names and/or area numbers, denoting the areas that the function **room** will describe if given no arguments. Its initial value is:

```
(working-storage-area compiled-function-area)
```

Resources

Introduction to Resources

Storage allocation is handled differently by various computer systems. With many languages, you must spend a lot of time thinking about when variables and storage units are allocated and deallocated. With Genera, freeing of allocated storage is normally done automatically by the Lisp system. When an object is no longer accessible to the Lisp environment, it is garbage collected. This relieves you of a great burden, and makes writing programs much easier.

Automatic freeing of storage incurs an expense: more computer resources must be devoted to the garbage collector. If your program uses a great deal of temporary storage that must be garbage collected, this expense can be high. In some cases, you might decide that it is worth putting up with the inconvenience of having to free storage under program control, rather than letting the system do it automatically. In this way you can eliminate a great deal of overhead from the garbage collector.

It is usually not worth worrying about the freeing of storage when the units of storage are very small things such as conses or small arrays. Numbers are not a

problem, either; fixnums and single-precision floating point numbers do not occupy storage. But when a program allocates and then gives up very large objects at a high rate (or large objects at a very high rate), it can be very worthwhile to keep track of that one kind of object manually. Several programs in Genera are examples of this case. For example, the Chaosnet software allocates and frees moderately large packets at a very high rate. The window system allocates and frees certain kinds of windows (such as menus), which are very large, moderately often. Both of these programs manage their objects by themselves, keeping track of when the the objects are no longer used.

When we say that a program frees storage, it does not really mean that the storage is freed in the same sense that the garbage collector frees storage. Instead, a list of unused objects is kept. When a new object is desired, the program first looks on the list to see if one already exists, and if so, uses it. Only if the list is empty does it actually allocate a new one. When the program is finished with the object, it returns it to this list.

The functions and special forms described in this section perform these tasks. The set of objects forming each such list is called a *resource*. For example, a Chaosnet packet could be specified as a resource. **defresource** defines a new resource; **allocate-resource** allocates one of the objects; **deallocate-resource** frees one of the objects (putting it back on the list); and **using-resource** temporarily allocates an object and then frees it.

Resources are not the only facility for manual storage management. See the section "Consing Lists on the Control Stack".

See the section "The Data Stack".

defresource *name parameters* &rest *options* *Function*

A special form that defines a new resource.

name should be a symbol; it is the name of the resource and gets a **defresource** property of the internal data structure representing the resource.

parameters is a lambda-list giving names and default values (if **&optional** is used) of parameters to an object of this type. For example, if you had a resource of two-dimensional arrays to be used as temporary storage in a calculation, the resource would typically have two parameters, the number of rows and the number of columns. In the simplest case *parameters* is **()**.

The keyword options control how the objects of the resource are made and kept track of. The syntax of each option is a keyword followed by a value. The following keywords are allowed:

:constructor

The value is either a form or the name of a function. It is responsible for making an object, and is used when someone tries to allocate an object from the resource and no suitable free objects exist. If the value is a form, it can access the parameters as variables. If it is a function, it is given the internal data structure for the resource and any supplied parameters as its

arguments; it needs to default any unsupplied optional parameters. This keyword is required, unless the **:finder** keyword is specified. **:constructor** is meaningless if **:finder** is provided, because **:finder** is expected to construct and manage its own objects.

:initial-copies
The value is a number (or **nil**, which means 0). This many objects are made as part of the evaluation of the **defresource**; this is useful to set up a pool of free objects during loading of a program. The default is to make no initial copies.

If initial copies are made and there are *parameters*, all the parameters must be **&optional** and the initial copies have the default values of the parameters.

:finder The value is a form or a function as with **:constructor** and sees the same arguments. If this option is specified, the resource system does not keep track of the objects. Instead, the finder must do so. It is called inside a **without-interrupts** and must find a usable object somehow and return it.

:matcher
The value is a form or a function as with **:constructor**. In addition to the parameters, a form here can access the variable **object** (in the current package). A function gets the object as its second argument, after the data structure and before the parameters. The job of the matcher is to make sure that the object matches the specified parameters. If no matcher is supplied, the system remembers the values of the parameters (including optional ones that defaulted) that were used to construct the object, and assumes that it matches those particular values for all time. The comparison is done with **zl:equal** (not **eq**). The matcher is called inside a **without-interrupts**. The matcher returns **t** if there is a match, **nil** if not.

:checker
The value is a form or a function, as above. In addition to the parameters, a form here can access the variables **object** and **in-use-p** (in the current package). A function receives these as its second and third arguments, after the data structure and before the parameters. The job of the checker is to determine whether the object is safe to allocate. The checker returns (not **in-use-p**). If no checker is supplied, the default checker looks only at **in-use-p**; if the object has been allocated and not freed it is not safe to allocate, otherwise it is. The checker is called inside a **without-interrupts**.

:initializer
The value is either a form or the name of a function. If the value is a form, it can access the parameters as variables. In addition to the parameters, a form here can access the variable **object** (in the current package). If it is a function, it is given the internal data structure for the resource, the object, and any supplied parameters as its arguments; it needs to default any unsupplied optional parameters. If the initializer is supplied, it is called by the resource allocator after an object has been allocated.

It sees **object** and its parameters as arguments when **object** is about to be allocated, whether it is being reused or was just created; it can initialize the object.

:deinitializer
The value is either a form or the name of a function. If it is a form, it can access the variable **object** (in the current package). If it is the name of a function, the function will be called with two arguments: the internal data structure for the resource, and the object.

If the deinitializer is supplied, it is called when the object is deallocated. If both **:finder** and **:deinitializer** are specified, the deinitializer is called when the object is deallocated even though the resource mechanism is not keeping track of the objects. **deallocate-whole-resource** calls the deinitializer for objects marked as in use. **clear-resource** does not.

:deinitializer should be used when an object being controlled via resources refers to other objects that have a chance to be reclaimed by the garbage collector. The deinitializer should clear references to such objects.

:free-list-size
The value is a number, with **nil** meaning the default value of 20 (decimal). **:free-list-size** is the initial size of the array that the resource uses to remember the objects it allocates and deallocates. The array expands if necessary.

Any function supplied to **defresource** for **:matcher**, **:checker**, or **:initializer** must supply defaults for any unsupplied optional arguments.

If these options are used with forms (rather than functions), the forms get compiled into functions as part of the expansion of **defresource**.

Most of the options are not used in typical cases. Here is an example:

```
(defresource two-dimensional-array (rows columns)
        :constructor (make-array (list rows columns)))
```

Suppose the array were usually going to be 100 by 100, and you wanted to preallocate one during loading of the program so that the first time you needed an array you would not have to spend the time to create one. You might simply put:

```
(using-resource (foo two-dimensional-array 100 100)
        )
```

after your **defresource**, which would allocate a 100 by 100 array and then immediately free it. Alternatively, you could do this:

```
(defresource two-dimensional-array
                        (&optional (rows 100) (columns 100))
        :constructor (make-array (list rows columns))
        :initial-copies 1)
```

Here is an example of how you might use the **:matcher** option. Suppose you wanted to have a resource of two-dimensional arrays, as above, except that when you allocate one you do not care about the exact size, as long as it is big enough. Fur-

thermore, you realize that you are going to have a lot of different sizes and if you always allocated one of exactly the right size, you would allocate a lot of different arrays and would not reuse a preexisting array very often. So you might do the following in Symbolics Common Lisp:

```
(defresource sloppy-two-dimensional-array (rows columns)
    :constructor (make-array (list rows columns))
    :matcher (and (≥ (array-dimension object 0) rows)
                  (≥ (array-dimension object 1) columns)))
```

Here, an array is filled with **nil** when it is initially allocated and when it is deallocated:

```
(defresource array-of-temporaries ()
  :constructor (make-array 100.)
  :initializer (fill object nil)
  :deinitializer (fill object nil))
```

allocate-resource *resource-name* &rest *parameters* *Function*

Allocates an object from the resource specified by *resource-name*. The various forms and/or functions given as options to **defresource**, together with any *parameters* given to **allocate-resource**, control how a suitable object is found and whether a new one has to be constructed or an old one can be reused.

Returns a resource and a *descriptor*. The *descriptor* is an object that points directly to this resource in the resource table. Using the *descriptor* with **deallocate-resource** speeds up deallocation.

Note that the **using-resource** special form is usually what you want to use, rather than **allocate-resource** itself.

Resources remember their parameters, so you can use `c-sh-A` with a resource to see its parameters.

deallocate-resource *resource-name object* &optional *descriptor* *Function*

Frees the object *resource-name*, returning it to the free-object list of the resource specified by *object*. *Descriptor* is an object that points to the resource table. A *descriptor* is the second object returned by **allocate-resource**. If *descriptor* is used with **deallocate-resource**, deallocation is faster.

deallocate-whole-resource *resource-name* *Function*

Deallocates all allocated objects of the resource specified by *resource-name*, returning them to the free-object list of the resource. You should use this function with caution. It marks all allocated objects as free, even if they are still in use. If you call **deallocate-whole-resource** when objects are still in use, future calls to **allocate-resource** might allocate those same objects for another purpose.

clear-resource *resource-name* *Function*

Forgets all the objects being remembered by the resource specified by *resource-name*. Future calls to **allocate-resource** create new objects. This function is useful if something about the resource has been changed incompatibly, such that the old objects are no longer usable. If an object of the resource is in use when **clear-resource** is called, an error is signalled when that object is deallocated.

map-resource *resource-name function* &rest *args* *Function*

Calls *function* once for every object in the resource specified by *resource-name*. *function* is called with the following arguments:

- The object
- **t** if the object is in use, or **nil** if it is free
- *resource-name*
- Any additional arguments specified by *args*

using-resource *(variable resource parameters...) body...* *Function*

The *body* forms are evaluated sequentially with *variable* bound to an object allocated from the resource named *resource*, using the given *parameters*. The *parameters* (if any) are evaluated, but *resource* is not.

using-resource is often more convenient than calling **allocate-resource** and **deallocate-resource**. Furthermore, it is careful to free the object when the body is exited, whether it returns normally or via **throw**. This is done by using **unwind-protect**.

Here is an example of **using-resource**:

```
(defresource huge-16b-array (&optional (size 1000))
  :constructor (make-array size :element-type '(unsigned-byte 16)))

(defun do-complex-computation (x y)
  (using-resource (temp-array huge-16b-array)
    ...                                ;Within the body, the array can be used
    (set (aref temp-array i) 5)
    ...))                              ;The array is returned at the end
```

using-resource also works with more than one resource, as in this example:

```
(using-resource ((r1 foo-array-resource foo-parameters)
                 (r2 bar-array-resource bar-parameters))
        <body>)
```

This example allocates several resources for the duration of <*body*>.

Resources remember their parameters, so you can use `c-sh-A` with a resource to see its parameters.

si:describe-resource *resource-name* *Function*

Describes the internal data structure for managing the resource named *resource-name*. It also tells how many objects have been created in the resource and, for each object, prints the object, the parameters, and whether or not the object is in use.

si:repair-resource *resource-name* *Function*

If you believe that a resource has become inconsistent due to typing `c-ABORT` while an allocation was in progress, this function will reclaim objects that the resource believes to be busy. It only reclaims objects that can safely be reclaimed. **si:repair-resource** does useful work only when there is no **:checker** or **:finder** supplied to **defresource**. Presumably the **:checker** would be able to repair similar damage on its own.

If a resource is aborted during an allocation or a deallocation with the resource locked, then the only damage that happens is that there is a possibility that the object being dealt with might never be allocated.

Device Interfaces

SCSI Interface

You can attach devices with SCSI interfaces to the XL400, UX-family, and MacIvory machines. Once the device is physically connected, you use functions and macros to communicate with the device. The SCSI software interface is the same for the XL400 and MacIvory.

Introduction to the SCSI Interface

The XL400 and MacIvory systems contain a SCSI (Small Computer Standard Interface) interface for controlling peripherals such as tape drives, optical disks, printers, etc. Symbolics does not support using SCSI disks on the XL400 for system use, such as for paging, FEPFS, LMFS, Statice, and so on.

SCSI is an ANSI standard I/O bus with well-specified mechanical and electrical characteristics and bus protocols, and the XL400 and MacIvory fully meets these specifications. See the document "Small Computer System Interface (SCSI)", published by the American National Standard for Information systems.

The XL400 system has two cable ports, one internal and one external, connected to a single SCSI bus. The internal cable is used for SCSI devices mounted in the system cabinet, such as the cartridge tape drive. The external connector is for any other peripherals; up to seven may be attached.

The SCSI bus includes a channel for high-bandwidth data transfer, and control signals and protocols for sharing that channel among up to eight devices. Usually the attached devices are classified as either *initiators* or *targets*; initiators are capable

of initiating an I/O operation, and targets are passive peripherals. The XL400 itself is a SCSI initiator, and will not respond as a SCSI target.

All SCSI device operations are specified by a high-level command protocol. For example, reading a given block of data is accomplished by issuing a single command to read that block, and that command is exactly the same regardless of the type or manufacturer of the target device. For most usage, a small set of core commands such as READ and WRITE will suffice, but there are often more specific commands available for a particular class of peripherals. The XL400 SCSI implementation allows all valid SCSI commands to be issued. For information on the SCSI commands, see the ANSI specification on SCSI.

Underlying the high-level SCSI command protocol are protocols for arbitrating between multiple initiators on the bus, connecting an initiator with a desired target, exchanging command and status information, and supervising data flow between an initiator and target. These protocols are handled transparently by the XL400 SCSI implementation, and user access to the underlying SCSI 'message' system is not provided. The XL400 fully supports the SCSI disconnect/reconnect protocol, for optimal bus sharing.

The XL400 SCSI hardware resides on the I/O board, and consists of an integrated SCSI host adaptor, an 8Kx32 buffer memory, and the internal and external cable ports. Data transfers are performed by hardware DMA between the host adaptor and the buffer memory, and the XL400 processor copies data between its virtual memory and the hardware buffer memory. This system supports peak data transfer rates of approximately 1.5 megabytes/second using the asynchronous protocol, and 4 megabytes/second using the synchronous protocol. Note, however, that the synchronous transfer option is not yet supported by the XL400 SCSI software.

Attaching a SCSI Device

The XL400 hardware and software enable you to communicate with SCSI devices attached to the machine. Any SCSI disk, tape drive, printer, or other kind of device can be connected. Up to seven peripheral SCSI devices may be connected.

Attaching a new SCSI device requires these steps:

1. Assigning the device a SCSI device number

   Each SCSI device must have a unique SCSI device number identifying it so that the XL400 can communicate with it. The XL400 can support seven peripheral SCSI devices, numbered 0 through 6. If you have an internal cartridge tape drive, it has a SCSI number assigned to it. When you connect a new device, you must assign it one of the unused SCSI device numbers. To find out which SCSI numbers are already assigned, use the Show Machine Configuration command. For information on how to assign a number to a device, refer to the vendor's documentation for that device.

2. Deciding whether to install the device internally or externally

There are two ways to add a SCSI device: internally (within the XL400 box itself), or externally. An internal device connects to the internal SCSI cable while an external device connects to the external SCSI bulkhead of the XL400. (Some XL400 machines are shipped with an internal cartridge tape drive, which is an example of an internal device.)

The internal connection should be reserved for a Symbolics-supplied or approved device. Symbolics must approve the device to ensure that there is sufficient power, cabling, and space for it.

Note: any device installed internally must have its terminator removed and must not supply termination power.

3. Controlling termination power (optional)

By default, the I/O board on the XL400 supplies termination power of 5 volts to the SCSI bus. (Note that the bottom light on the I/O board is turned on if the I/O board is supplying termination power, and is turned off otherwise.) If you are adding a SCSI device yourself, you must ensure that the device is not supplying termination power.

Note that termination power can come from one of two sources: the XL400 I/O board, or an external SCSI device. Note that an internal SCSI device must not supply termination power. It is your choice whether the I/O board or an external SCSI device supplies termination power. If you prefer that an external SCSI device supplies termination power, you must contact your Symbolics Customer Service representative, who will configure the machine for you.

4. Connecting the device and ending terminator properly

When connecting SCSI devices, it is necessary that there are exactly two terminators in the SCSI chain. One terminator should be placed at the beginning of the chain, and the other at the end of the chain.

If the terminators are not installed properly, then damage to your hardware can result. If more than two terminators are installed, damage to your hardware can result due to excessive drive currents being required from the SCSI bus drivers. If there are less than two terminators, or the terminators are not at either end of the SCSI bus, then SCSI data and control signal reflections may cause intermittent problems in the SCSI operation.

When you receive your XL400, the XL400 boardset is the first SCSI device in the chain, and the first terminator is correctly installed at the beginning of the chain. (This is inside the machine, and you won't see it, or have any need to change it.) The second terminator is placed on the external SCSI bulkhead of the XL400, at the other end of the XL400's SCSI bus.

Some SCSI devices come with internal terminators. Terminators are usually resistor packs that must be physically removed if you do not wish to terminate at the device. You should find out whether your device has an internal terminator. Remember, there cannot be more than two terminators in the SCSI chain; one terminator should be at each end.

To connect a device internally, verify that the device does not supply bus termination power. Next, remove any terminator in that device. You can terminate the chain with the terminator at the external SCSI bulkhead of the XL400.

XL400 machines are shipped with a cable that is routed into the internal drive area and then goes to the external SCSI bulkhead of the XL400. This means that internal devices must be placed in the top drive slot. An internal SCSI tape drive must be in the top drive slot. Place the device inside the XL400 machine and connect it to the power supply.

To connect a new SCSI device externally, verify that the device does not supply bus termination power (if you prefer that the device does supply termination power, you must contact your Symbolics Customer Service representative, who will configure your XL400 such that the I/O board does not supply termination power). Next, remove the terminator from the end of the chain. Connect the new device to the location where the ending terminator was. Finally, you need to terminate the end of the chain. You can do this either by ensuring that the device has an internal terminator, or by placing the Symbolics-supplied terminator at the new end of the chain. In either case, the new end of the chain is on the last SCSI device.

Examples of Using the SCSI Interface

Standard SCSI commands (usually called "Command Descriptor Blocks") have three basic representations: six, ten, or twelve bytes long. All of these representations include an 8-bit opcode as their first byte. These commands are represented in the XL400 SCSI system as vectors of element type **(unsigned-byte 8)**. The data representation in these commands is byte-swapped from that used by the Ivory processor; that and the complexity of SCSI command encodings makes it laborious to build these commands manually. The Octet Structure facility, particularly the **rpc:define-octet-structure** macro, can help with this task. See the section "Defining Octet Structures for SCSI Commands".

The files SYS:SYS;SCSI-DEFINTIIONS.LISP and SYS:SYS;SCSI-TOOLS.LISP contain definitions of some standard SCSI command and response formats, including the structures used in these examples.

SCSI Read Example

This example uses the SCSI Read command to read a block of data from a device.

```
;;; Define the accessors for a SCSI Read command descriptor
;;; block.
(rpc:define-octet-structure (scsi-direct-read-command :access-type
                                                      :byte-swapped-8)
  operation-code
  (+ ((logical-unit-number (load-byte (unsigned-byte 24) 21 3)))
     ((logical-block-address (load-byte (unsigned-byte 24) 0 21))))
  transfer-length
  *)

;;; Read a single block of data from the specified SCSI port
;;; into the given array.
(defun read-block (port block array)
  (stack-let ((command (make-scsi-read-command
                         :operation-code %scsi-command-read
                         :logical-block-address block
                         :transfer-length 1)))
    (scsi:scsi-port-check-status port
      (scsi:scsi-port-execute-read-command port command array))
    array))

;;; Calling sequence: use READ-BLOCK to read block 100 of device 0
;;; into an array.
(scsi:with-scsi-port (port 0)
  (read-block port 100 (make-array 512 :element-type '(unsigned-byte 8))))
```

SCSI Inquiry Example

This example uses the SCSI Inquiry command to determine the vendor and model of the device assigned a particular ID on the SCSI bus. The Octet Structure facility is used to specify the Inquiry command format, and also to interpret the device response. See the section "Defining Octet Structures for SCSI Commands".

```
;;; Define the accessors for a SCSI Inquiry command descriptor
;;; block.
(rpc:define-octet-structure (scsi-inquiry-command :access-type
                                                  :byte-swapped-8)
  operation-code
  (logical-unit-number (load-byte (unsigned-byte 8) 5 3))
  *
  *
  allocation-length
  *)
```

```
;;; Define the accessors for the response to a SCSI Inquiry command.
(rpc:define-octet-structure (scsi-inquiry-data :access-type
                                               :byte-swapped-8)
  device-type
  (+ ((device-type-qualifier (load-byte (unsigned-byte 8) 0 7)))
     ((removable-medium (rpc::boolean-bit (unsigned-byte 8) 7))))
  (+ ((ansi-version (load-byte (unsigned-byte 8) 0 3)))
     ((ecma-version (load-byte (unsigned-byte 8) 3 3))))
  (respond-data-format (load-byte (unsigned-byte 8) 0 4))
  additional-length
  vu
  *
  *
  (vendor-name (vector rpc:character-8 8))
  (product-name (vector rpc:character-8 16))
  (revision-level (vector rpc:character-8 4)))

;;; Determine the vendor and model of the device associated with
;;; the specified port.
(defun scsi-port-vendor-and-model (port)
  (stack-let ((command (make-scsi-inquiry-command
                         :operation-code %scsi-command-inquiry
                         ;; The Inquiry response is 36 octets long.
                         :allocation-length 36))
              (response (make-array 36 :element-type '(unsigned-byte 8))))
    (scsi:scsi-port-check-status port
      (scsi:scsi-port-execute-read-command port command response))
    (values (scsi-inquiry-data-vendor-name response 0)
            (scsi-inquiry-data-product-name response 0))))

;;; Calling sequence: use scsi-port-vendor-and-model to probe
;;; device 0.
(scsi:with-scsi-port (port 0)
  (scsi-port-vendor-and-model port))
```

Defining Octet Structures for SCSI Commands

The Octet Structure facility enables you to conveniently define octet structures for any purpose, including defining SCSI commands and defining octet structures for interfacing to the Macintosh from Lisp. This section describes the mechanisms most useful for defining SCSI commands. For examples:

See the section "SCSI Read Example".
See the section "SCSI Inquiry Example".

For details on how to use the Octet Structure facility to interface to the Macintosh, see the section "Defining Octet Structures".

rpc:define-octet-structure *name-and-options* &body *fields* *Macro*

Defines an octet structure.

name-and-options The name of the octet structure or the list containing the name of the structure and some number of keyword value pairs.

This macro takes the keyword arguments **:conc-name**, **:constructor, :default-pointer,** and **:include**, which behave in a way similar to the corresponding keywords for **defstruct**. See the section "Options for **defstruct**" for further information. In addition, **rpc:define-octet-structure** takes the following keyword arguments:

:access-type Specifies how references are made by default as one of **:octet**, **:unsigned-8**, or **:byte-swapped-8**. To define octet structures to represent Macintosh structures, for use by the Toolbox remote entries, always use **:byte-swapped-8**. The default is **:octet**.

:alignment Controls the automatic insertion of padding. This is useful when defining structures that you want to correspond directly to structures defined in another language or on a different architecture or both. **:alignment** takes an integer value as an argument. Specifying an alignment of *n* means that all structure fields of size greater than or equal to *n* should be aligned with the next offset evenly divisible by *n*. Where the field is a vector it will be aligned based on the element size of the vector.

For example, when defining octet-structures in Lisp to represent C structs as defined by THINK C on the Macintosh, you will want to specify an alignment of 2. This will align all structure fields of 2 or more bytes to an even byte boundary, and make sure that the Lisp accessors defined by **rpc:define-octet-structure** correspond precisely to where the data is stored by C. In any situation where you are using octet-structures to represent data that is created by one machine/language and manipulated by another, it is essential that you take into account the storage conventions of the other implementation.

:define-accessors Inhibits definition of macros.

:export As for **defstruct**, takes a list of keywords from the set (**:structure-name :accessors :constructor**).

:default-type An integer type. :**integer-8** is the default.

:default-index Makes the second argument to each accessor optional, defaulting it to the value you supply with this argument.

Elements of *fields* can be a symbol for single unsigned byte fields, or a list of field name and type. * used for a name allocates space, but doesn't define accessors. * used as a type defines subfields that overlap, such as bit fields. + used as a name defines unions.

Predefined Octet Structure Field Data Types

Atomic field types

When defining octet-structures it is often useful to define a new field type in terms of some conversion routine, or some expansion, of a previously defined type. Although most of the MacOS types are already defined as octet-structure field types, the following is a short list of the basic field types: those which are most useful when defining your own octet structures and octet-structure field types.

unsigned-byte (&optional *(size 8)*)

signed-byte (*size* &)

integer-32 Equivalent to (**signed-byte 32**).

cardinal-32 Equivalent to (**unsigned-byte 32**).

integer-16 Equivalent to (**signed-byte 16**).

cardinal-16 Equivalent to (**unsigned-byte 16**).

integer-8 Equivalent to (**signed-byte 8**).

cardinal-8 Equivalent to (**unsigned-byte 8**).

padding (*base-type* &optional *(repeat 1)*)
Just occupies space; the field cannot be accessed. *base-type* can be any field type.

vector (*type length*) *length*
Can be a form that references earlier structure elements (such as repeat byte count). Referencing returns a vector of the elements, or you can use the **octet-structure-field-elements loop** iteration path.

load-byte (*base-type position size*)
Accesses a subfield of *base-type* access. Useful with *.

bit (*base-type bit-number*)
Equivalent to **(load-byte base-type bit-number 1)**.

boolean (*base-type*) Equivalent to *base-type* with not-temp test.

boolean-bit (*base-type bit-number*)
Equivalent to **(boolean (bit** *base-type bit-number*)**)**.

member (*base-type set*)
set is a form to evaluate to a sequence indexed by field.

subset (*base-type keywords*)
One bit for each position to indicate the presence of corresponding element.

character-8 () Equivalent to unsigned-8 with code-char input and char-code output.

ascii-character-8 () Equivalent to unsigned-8 with ASCII-char input and char-ASCII output.

Dictionary of SCSI Functions

scsi:*scsi-minimum-maximum-buffer-size* *Constant*

The maximum buffer size (in octets) that may be used in a call to data transfer operations (**scsi:scsi-port-execute-write-command** and **scsi:scsi-port-execute-read-command**) depends on the system. For embedded systems, this value may be dependent on the host's available memory and, in fact, may be dynamic. All systems, however, guarantee that all data transfer operations will accept any buffer whose size is no more than **scsi:*scsi-minimum-maximum-buffer-size*** octets.

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).

scsi:scsi-port-check-status *port status* *Function*

A default function for handling SCSI errors. The *status* argument is the status code returned by the most recently executed SCSI command, and the *port* argument is the relevant port. If the *status* argument indicates that the command completed successfully, then this function does nothing. Otherwise, it queries the device (using the SCSI request sense command) to get more information about its status, and signals an error if appropriate.

scsi:scsi-port-execute-control-command *scsi-port command* *Function*

Executes a SCSI command that transfers no data. Returns the status of the command, which is either a SCSI status code (if it is less than 256) or a special code to indicate a SCSI bus failure.

scsi-port is the port over which the SCSI command should be given. *command* is of type **(vector (unsigned-byte 8))** and contains the command bytes.

scsi:scsi-port-execute-read-command *scsi-port command buffer* &key *(:start* **0***) (:end* **(length scsi::buffer)***) (:block-size* **4***) (:stream-size* **0***)* *Function*

Executes a SCSI command that reads data from the device into the buffer.

scsi-port is the port over which the SCSI command should be given. *buffer* is of type **(vector (unsigned-byte 8))** and receives the data. *start* and *end* indicate the portion of *buffer* to be used.

block-size allows the caller to indicate that the device will transfer data in units of that number of octets (units of eight bits). *stream-size* can be used to limit the amount of data allowed in the hardware I/O buffer. These two arguments can be used to improve the performance of the driver when the bus is heavily used. For example, if you know that your SCSI device always writes only 5000 bytes, then supply **:stream-size 5000**; the system will then read no more than 5000 bytes from the buffer. If you know that your device always writes data in chunks of 1000 bytes, then supply **:block-size 1000**, and the system will read data in multiples of 1000 bytes.

This function returns two values. The first value is the status of the command, which is either a SCSI status code (if it is less than 256) or a special code to indicate a SCSI bus failure. The second value is the number of octets transferred.

scsi:scsi-port-execute-write-command *scsi-port command buffer* &key *(:start* **0***) (:end* **(length scsi::buffer)***) (:block-size* **4***) (:stream-size* **0***)* *Function*

Executes a SCSI command that writes data from the *buffer* to the SCSI device.

scsi-port is the port over which the SCSI command should be given. *buffer* is of type **(vector (unsigned-byte 8))** and supplies the data. *start* and *end* indicate the portion of *buffer* to be used.

block-size allows the caller to indicate that the device will transfer data in units of that number of octets. *stream-size* can be used to limit the amount of data allowed in the hardware I/O buffer. These two arguments can be used to improve the performance of the driver when the bus is heavily used. For example, if you know that your SCSI device always asks for 5000 bytes, then supply **:stream-size 5000**; the system will then put no more than 5000 bytes into the buffer. If you know that your device always accepts data in chunks of 1000 bytes, then supply **:block-size 1000**, and the system will write data in multiples of 1000 bytes.

This function returns two values. The first value is the status of the command, which is either a SCSI status code (if it is less than 256) or a special code to indicate a SCSI bus failure. The second value is the number of octets transferred.

scsi:scsi-port-maximum-buffer-size *scsi-port* *Function*

Returns the largest buffer size (in octets) that may be used in a call to data transfer operations (**scsi:scsi-port-execute-write-command** and **scsi:scsi-port-execute-read-command**). For embedded systems, this value may be dependent on the host's available memory and, in fact, may be dynamic. All systems, however, guarantee that all data transfer operations will accept any buffer whose size is no more than **scsi:*scsi-minimum-maximum-buffer-size*** octets.

scsi-port is the port whose maximum buffer size is desired.

scsi:scsi-port-open *unit* &key *(:if-locked* **:error***) :controller* *Function*

Attaches a SCSI device, and returns a SCSI port object which is used to refer to the device to all other entrypoints.

unit is an integer between 0 and 7, representing the SCSI bus address of the device to be used. Note that on an XL400, unit 7 is assigned to the XL400 itself.

The keyword **:if-locked** can be either **:error** to signal an error if the device is already in use, or **nil** to return **nil**.

The keyword **:controller** can be **nil** to indicate the default controller, or 0 and 1 to indicate which I/O board.

Generally, **scsi:with-scsi-port** should be used instead of **scsi:scsi-port-open** and **scsi:scsi-port-close**.

scsi:scsi-port-close *port* &key *:abort* *Function*

Undoes the effects of **scsi:scsi-port-open**.

The *abort* keyword indicates whether the closing is normal (a **nil** value) or abnormal (a true value). Supply **:abort t** for an abnormal termination. The default is **nil**.

Generally, **scsi:with-scsi-port** should be used instead of **scsi:scsi-port-open** and **scsi:scsi-port-close**.

scsi:with-scsi-port *(port unit* &key *(:if-locked* **:error***)) :controller* &body *body* *Macro*

Attaches a SCSI device, and keeps it open for the duration of *body*. This macro should generally be used instead of using the **scsi:scsi-port-open** and **scsi:scsi-port-close** functions.

port is a variable which is bound to the SCSI port object; this is used to refer to the device in all other entrypoints within the *body*.

unit is an integer between 0 and 7, representing the SCSI bus address of the device to be used. Note that on an XL400, unit 7 is assigned to the XL400 itself.

The keyword **:if-locked** may be **:error** to signal an error if the device is already in use, or **nil** to return **nil**. The default is **:error**.

The keyword **:controller** can be **nil** to indicate the default controller, or 0 and 1 to indicate which I/O board.

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:

Modifier	*Description*
#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 **:data-type** 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.

Disk System User Interface

This chapter describes the portions of the disk system that are available to the user. The discussion is organized as follows:

Three sections introduce some basic definitions and concepts. For a discussion of the terms used throughout this chapter: See the section "Disk System Definitions and Constants".

For descriptions of the disk array and disk event data structures that are the basic buffers for data and synchronization information: See the section "Disk Arrays". See the section "Disk Events".

Three sections describe disk transfers in detail. For a description of the low-level user disk transfer mechanism that is the basis for more sophisticated interfaces, such as the FEP file system: See the section "Disk Transfers".

To learn about the error-handling mechanism: See the section "Disk Error Handling".

For a discussion of the FEP file system and disk streams: See the section "FEP File System and Disk Streams".

For a discussion of disk performance, along with some basic approaches for achieving high performance: See the section "Disk Performance".

For examples that illustrate concepts introduced in all the sections mentioned above: See the section "Examples of High Disk Performance".

For a description of the disk utilities such as the FEP file system verifier, and of routines to mount disk units: See the section "Disk and FEP File System Utilities".

Definitions and Constants

The Genera disk system is capable of transferring data in either *user mode*, in which data are packed 32 bits per memory word, as fixnums, or *system mode*, in which data are packed 36 bits (3600 family) or 40 bits (Ivory family) per memory word. These modes only affect how the data are represented in memory; the data are stored as a stream of bits on the disk in either case. This section does not describe system mode, which is used only by the virtual memory system.

Data are stored on a *disk pack*. To access the disk pack, you must use a *disk drive*. The 3600 family can address multiple disk drives, but only one disk pack at a time can be mounted per disk drive. Most of the disk drives available on 3600 and Ivory family systems have nonremovable disk packs.

Each disk drive is assigned a unique small nonnegative number, called the *unit number*, that identifies the drive. A unit number ranges from 0 up to, but excluding, 32 decimal. However, the disk drive hardware can restrict the maximum to a smaller value, such as 8. The term *disk unit* refers to the combination of the disk drive and a mounted disk pack.

The space available on a disk unit is divided into equal-sized blocks called *disk blocks* or *disk pages*. A disk block is the smallest unit that can be transferred be-

tween the disk and virtual memory. On 3600 family systems, each disk block contains 9,216 bits of data, which are viewed as 288 fixnums in user mode or 256 tagged words in system mode. On Ivory family systems, each disk block contains 10,240 bits of data, which are viewed as 320 fixnums in user mode or 256 tagged words in system mode. The symbolic constant **si:disk-sector-data-size32** indicates the number of fixnums that fit into a disk block for the running system.

A *disk address* is a unique identifier for a disk block residing on a mounted disk pack. A disk address, also called a *disk page number (DPN)*, is composed of a unit number and a block number relative to that unit. Note that **sys:%logdpb** should be used when constructing DPNs from their constituents; a DPN must always be a fixnum.

sys:%%dpn-unit *Variable*

A byte specifier for accessing the unit number field in a disk address.

sys:%%dpn-page-num *Variable*

A byte specifier for accessing the block number field in a disk address. Block numbers are relative to a disk unit, where zero addresses the first disk block, and successive integers address consecutive blocks.

si:disk-sector-data-size32 *Variable*

The number of user-mode data words (as fixnums) available in a disk block.

si:disk-block-length-in-bytes *Variable*

The number of bytes available in a disk block.

Disk Arrays

Disk arrays are arrays that buffer disk transfers and are specially allocated to satisfy page alignment constraints imposed by the disk system. The data contained in consecutive disk blocks are stored in the array elements of a disk array; each element of a disk array contains a 32-bit datum from a disk block.

Disk arrays are resource objects, and so must be allocated and deallocated explicitly by the **allocate-resource** and **deallocate-resource** functions, or by the **using-resource** special form. (For more information about resources: See the section "Resources".)

sys:disk-array &optional *length* &rest *make-array-options* *Resource*

The set of all disk arrays currently known by the system. The *length* resource parameter specifies the minimum number of elements the disk array should contain;

its default value is **si:disk-sector-data-size32**. The length of the disk array actually allocated can be greater. *make-array-options* is a list of keywords and values to pass to **zl:make-array**. Only the following keywords are permitted in *make-array-options*:

:area	The area the array should be allocated in. The area's **:gc** attribute must be **:static**. The default area is **si:disk-array-area**.
:type	The type of the array to be allocated. Only fixnums should be stored in the disk array. The default type is **sys:art-fixnum**.
:initial-value	The initial value to fill the array with, which must be a fixnum. The default value is zero.

The **sys:disk-array** resource allocator returns a disk array object at least *length* elements long and with matching **:area** and **:type** values, filled with the value of **:initial-value**. If a matching disk array object cannot be found, a new one is created.

si:disk-array-area *Variable*

The default area to allocate disk arrays in.

storage:disk-array-block-count *disk-array* *Function*

Accesses the slot in *disk-array* describing the number of disk blocks that the disk array can contain.

Disk Events

Disk events are structures used for synchronizing disk transfers and for storing disk error information. Disk events are resource objects, and so must be allocated and deallocated explicitly by the **allocate-resource** and **deallocate-resource** functions, or by the **using-resource** special form. (For more information about resources: See the section "Resources".)

Synchronization is accomplished through the use of *disk event tasks*. A disk event task is a disk command that is enqueued into the disk queue in the same way that disk reads and disk writes are enqueued. When the disk system dequeues the task, the task is flagged as being completed. **si:disk-event-task-done-p** is a predicate that examines this flag, returning true when the task is completed. For example, if the disk queue contains a disk read, then a disk event task, and finally a disk write, the disk event task is flagged as completed after the disk finishes reading but before the disk starts writing.

Disk event tasks are identified by a task number that must be explicitly allocated and deallocated by the **si:disk-event-enq-task** and **si:return-disk-event-task** functions, or by the **si:with-disk-event-task** special form.

Synchronization may also be accomplished simply by waiting for all the pending disk transfers associated with a given disk event to complete, using **storage:wait-for-disk-event**

In addition to synchronizing disk transfers, disk events are also *associated* with disk transfers in case of a disk error. (For a detailed description of disk error handling: See the section "Disk Error Handling".) You associate a disk event with a disk transfer via the **sys:disk-read** and **sys:disk-write** functions.

sys:disk-event *Resource*

The set of disk event objects currently known by the system. The resource allocator returns a disk event object, creating a new one if all the current disk events are already in use.

Synchronization Functions

The following functions manipulate disk event tasks for synchronizing disk transfers:

si:with-disk-event-task *(task-var disk-event)* &body *body variable disk-event* &body *body* *Function*

Allocates and enqueues a task in *disk-event* and binds the task number to *variable*. The task is deallocated on exit or if the body is aborted.

si:disk-event-enq-task *disk-event* *Function*

Allocates a free task in *disk-event*, and enqueues it in the disk queue. The return value is the task number.

si:return-disk-event-task *disk-event task-id disk-event task-number* *Function*

Deallocates the *task-number* task in *disk-event*.

si:disk-event-task-done-p *disk-event task-id disk-event task-number* *Function*

Returns **t** if the *task-number* task in *disk-event* has completed, **nil** if it has not completed.

si:wait-for-disk-event-task *disk-event task-id disk-event task-number* *Function*

Waits for the *task-number* task in *disk-event* to complete.

storage:wait-for-disk-event *disk-event* *Function*

Waits for all outstanding disk transfers associated with *disk-event* to complete.

storage:wait-for-disk-done *Function*

Waits for all outstanding disk transfers to complete, regardless of which disk event the transfer is associated with, or whether the transfer is in user or system mode.

Disk Event Accessor Functions

The following accessor functions refer to the error information and task counters stored in a disk event. Most of the error information is meaningless if an error has not occurred yet. The **si:disk-event-error-type** accessor function is the correct predicate to use to determine if an error has occurred for a disk transfer associated with the disk event.

si:disk-event-size *disk-event* *Function*

Accesses the slot in *disk-event* containing the number of disk event tasks that can be concurrently allocated.

si:disk-event-count *disk-event* *Function*

Accesses the slot in *disk-event* containing the number of disk event tasks currently allocated.

si:disk-event-error-type *disk-event* *Function*

Accesses the slot in *disk-event* containing a disk error code or **nil** if no disk transfer associated with *disk-event* has generated an error. A disk error code is a number indicating the type of disk error, as described elsewhere: See the section "Disk Error Codes". This accessor function is the predicate for determining if an error has occurred for a disk transfer associated with *disk-event*.

si:disk-event-suppress-error-recovery *disk-event* *Function*

Accesses the slot in *disk-event* that indicates if the automatic error recovery for specific error codes is suppressed for transfers associated with *disk-event*. All other transfers are unaffected. The bits in the mask correspond to the disk error code numbers. If the bit is set (a value of one) the corresponding error is not automatically recovered from and instead is signalled immediately. If the bit is clear (a value of zero) an error causes the disk system to attempt to recover from the error, signalling an error only if it cannot recover from the disk error. See the section "Disk Error Codes".

Setting the disk event's **si:disk-event-suppress-error-recovery** mask immediately affects any pending disk transfers that are associated with the disk event in addition to any subsequently associated transfers. The error recovery remains suppressed until the corresponding bit in the mask is cleared.

For example, to turn off the automatic recovery of ECC errors so that an error would be signalled on any ECC error in a transfer associated with a given disk event, even if the ECC error is correctable, use the form:

```
(setf (ldb (byte 1 sys:%disk-error-ecc)
           (si:disk-event-suppress-error-recovery disk-event))
      1)
```

The following form returns a value of 1 if the disk event's ECC error recovery is suppressed, or 0 if it is not.

```
(ldb (byte 1 sys:%disk-error-ecc) ; Make a PPSS byte specifier
     (si:disk-event-suppress-error-recovery disk-event))
```

Disk Transfers

This section describes the low-level interface for initiating disk read and write transfers. The FEP file system provides a higher-level interface built upon these functions and is the standard way to access the disk. For details on the FEP file system: See the section "FEP File Systems".

Disk transfers can be either disk reads or disk writes. A disk read copies data from the disk into disk arrays. A disk write copies data from disk arrays to the disk. The data transferred must always be a multiple of a disk block, due to constraints imposed by the disk system.

Transfers are always performed in the order they are enqueued. This permits a sequence of transfers that must be performed in a particular order to be enqueued without having to wait for completion between each transfer.

For example, when the FEP file system creates a new file, it first enqueues the writes of the modified blocks in its free page data structure. It then enqueues a write of the file's page table, followed by a write of the directory entry pointing to the file's page table, without waiting for the individual writes to complete before enqueuing the next. These data structures must be written in this particular order to ensure that the copy of the file system on the disk is always consistent. When the FEP file system enqueues the writes it specifies a *hang-p* argument of **nil** to **sys:disk-write**, and uses the same disk event for all the transfers in the sequence. Since all the transfers are associated with the same disk event, if one transfer fails and is aborted, all subsequent transfers are also aborted. (For more details on error handling: See the section "Disk Error Handling".) Thus, if the write of the file's page table fails and is aborted, the write of the directory page is also automatically aborted.

All the disk arrays and the disk event must be *wired* for the duration of the disk transfer. (Wiring a structure locks it in memory until it is explicitly unwired, permitting the disk system to use physical memory addresses for the data transfers.) Disk arrays are wired with the **storage:wire-disk-array** and **storage:with-wired-disk-array** functions, while disk events are wired with the **storage:wire-structure** and **storage:with-wired-structure** functions.

If the *hang-p* argument to the disk transfer function is true, the function wires and unwires the disk arrays and disk event itself. Otherwise these must be wired by the caller and unwired only after the disk transfer has completed. See the section "Synchronization Functions". The functions described there can be used to determine when the disk transfer has completed.

sys:disk-read *disk-arrays disk-event dpn* &optional *n-blocks* (*hang-p* **t**) (*block-offset* **0**) *Function*

Causes the disk to start reading the consecutive disk blocks beginning with the block at disk address *dpn*, storing the data from the disk into the arrays in *disk-arrays*. *disk-arrays* can be a disk array or a list of disk arrays. *n-blocks* is the number of disk blocks to read, and defaults to the number of blocks *disk-arrays* can contain. When *n-blocks* is greater than one, each disk array is completely filled before using the next disk array in *disk-arrays*. Unused disk arrays or portions of disk arrays remain unmodified.

When *hang-p* is **t** (its default value), **sys:disk-read** waits for all the reads to complete before returning. If *hang-p* is false, **sys:disk-read** returns immediately upon enqueuing the disk reads without waiting for completion. When *hang-p* is false, all the *disk-arrays* and the *disk-event* must be wired before calling **sys:disk-read**, and must remain wired until the disk reads complete.

disk-event must be the disk-event to associate with all the disk reads.

block-offset is an offset into *disk-arrays*. Use it when you wish to transfer the data to a starting position other than the beginning of the first array.

sys:disk-write *disk-arrays disk-event dpn* &optional *n-blocks* (*hang-p* **t**) (*block-offset* **0**) *Function*

Causes the disk to start writing the consecutive disk blocks beginning with the block at disk address *dpn* with the data stored in the disk arrays in *disk-arrays*. The arguments to **sys:disk-write** are identical to those of **sys:disk-read**.

storage:with-wired-disk-array (*disk-array*) &body *body* *Function*

Before the body is entered, *disk-array* is made permanently resident in main memory. When control leaves the body, either normally or abnormally (via a **throw**, such as by an error which was not handled within the body), the array is made eligible for replacement by the memory system.

storage:wire-disk-array *disk-array* *Function*

Makes *disk-array* be permanently resident in main memory until **storage:unwire-disk-array** is called. Disk arrays must be wired for the duration of a disk transfer. The transfer functions automatically wire disk arrays if they also wait for the transfer to complete; otherwise the programmer must explicitly wire and unwire the disk arrays.

It is preferable to use **storage:with-wired-disk-array** rather than explicit calls to **storage:wire-disk-array** and **storage:unwire-disk-array**.

storage:unwire-disk-array *disk-array* *Function*

Makes *disk-array* eligible for replacement by the virtual memory system. There must be a matching **storage:unwire-disk-array** for every **storage:wire-disk-array**. **storage:unwire-disk-array** is usually called as a cleanup handler in an **unwind-protect** form.

storage:with-wired-disk-event *(disk-event)* &body *body* *Macro*

Makes *disk-event* permanently resident in main memory for the duration of the execution of the body. When control leaves the body, either normally or abnormally (via a throw, such as by an error which was not handled within the body), the event is made eligible for replacement by the memory system.

storage:wire-disk-event *event* *Function*

Makes *disk-event* permanently resident in main memory until **storage:unwire-disk-event** is called. Disk events must be wired for the duration of a disk transfer. The transfer functions automatically wire disk arrays if they also wait for the transfer to complete; otherwise the programmer must explicitly wire and unwire the disk events.

It is preferable to use **storage:with-wired-disk-event** rather than explicit calls to **storage:wire-disk-event** and **storage:unwire-disk-event**.

storage:unwire-disk-event *event* *Function*

Makes *disk-event* eligible for replacement by the virtual memory system. There must be a matching **storage:unwire-disk-event** for every **storage:wire-disk-event**. **storage:unwire-disk-event** is usually called as a cleanup handler in an unwind-protect form.

Disk Error Handling

The disk system automatically attempts to recover from a disk error by resetting the relevant disk state and retrying the failed disk transfer. (The associated disk event's **si:disk-event-suppress-error-recovery** slot can be used to selectively suppress the automatic error recovery for a set of disk error types.) After the number of retry attempts fail, the error is considered to be unrecoverable and the failed transfer is aborted.

The disk system permits related disk transfers to be grouped together by associating them with the same disk event. If one of the transfers fails, the remaining transfers in its group are aborted. This makes it possible to enqueue transfers that

must be performed in a particular order without having to wait for each transfer to complete. Aborting the remaining transfers in a group does not interfere with transfers in other groups.

Disk errors are signalled after they actually occur because they are detected at a low level in the system, asynchronous to the execution of the responsible process. In order to make condition handling of disk errors possible, the error is signalled when a process waits for the disk transfers to finish.

The disk system performs the following sequence of events when an error is detected:

1. It suspends processing of the disk queue at the failed disk transfer.

2. It resets the relevant hardware and retries the failed disk transfer, depending on the type of error. If the retry succeeds, no error is signalled and processing of the disk queue resumes.

3. If the disk error recovery logic cannot automatically recover from the error, or if error recovery is being suppressed, the error becomes *unrecoverable* and the disk system aborts the failed disk transfer.

4. If the failed disk transfer does not have an associated disk event, the unrecoverable error becomes fatal and the disk system halts the machine. (Most system mode disk transfers do not have an associated disk event.) Otherwise, it stores the information describing the error in the disk event.

5. The disk system removes from the disk queue any remaining pending transfers that are associated with the same disk event as the failed transfer. The **si:disk-event-error-flushed-transfer-count** slot in the disk event contains the number of transfers that were removed from the disk queue, including the failed transfer.

6. The disk system resumes processing of the remaining transfers that are not associated with the failed transfer's disk event.

7. It discards any subsequent attempts to initiate a disk transfer associated with the failed transfer's disk event, incrementing the disk event's flushed transfer counter.

8. When **storage:wait-for-disk-event** or **si:wait-for-disk-event-task** waits for a task in the failed transfer's disk event, an **si:disk-error-event** condition (which is built upon the **sys:disk-error** condition) is signalled. These synchronization functions are also used by the transfer functions when their *hang-p* argument is true.

The **si:disk-event-error-type** slot of a disk event can also be explicitly checked to determine if an error has occurred.

Disk Error Conditions

si:disk-error-event *Flavor*

Signalled while waiting for a task in a disk event that is associated with a disk transfer that generated a disk error. **si:disk-error-event** is based upon the **sys:disk-error** condition; condition handlers should use the **sys:disk-error** condition.

(flavor:method :disk-event si:disk-error-event) *Method*

Returns the disk event associated with the failed transfer. This is especially useful when transfers associated with multiple disk events can be handled by the same condition handler.

(flavor:method :error-type si:disk-error-event) *Method*

Returns the error type code number. For a list of the possible disk error code numbers, see the section "Disk Error Codes".

(flavor:method :flushed-transfer-count si:disk-error-event) *Method*

Returns the number of disk transfers that were not performed because of the error, including the failed transfer.

Disk Error Codes

A disk error code is a number indicating the type of the disk error. System constants containing the disk error code numbers exist so the codes can be referred to mnemonically.

sys:*disk-error-codes* *Variable*

A list of symbols corresponding to the disk error code numbers. You can convert a disk error code number into the symbol of its corresponding constant as follows:

```
(nth disk-error-code-number sys:*disk-error-codes*)
```

The following list shows the disk error constants and describes the corresponding error causes.

sys:%disk-error-drive-fault *Variable*

The selected disk drive signaled some fault condition. This usually means that either the drive was not ready (was not spinning at its rated speed) or that a write was issued to a write-protected drive. It can also indicate a malfunction in the drive.

sys:%disk-error-controller-fault *Variable*

Usually indicates a malfunction in the disk controller.

sys:%disk-error-select *Variable*

The disk unit could not be selected. For a disk unit to be selectable the drive must be properly connected to the machine and a unique disk unit number set in the drive's unit address switches. The error recovery logic tries to reselect the unit before failing with an unrecoverable select error.

sys:%disk-error-not-ready *Variable*

The disk unit was selected, but was not ready. A disk unit is ready when the drive is spinning at its rated speed. Some drives are not ready when they are in a device fault. When a disk is started, the unit is not ready for a short period (10 to 50 seconds for most drives) while the disk is spinning up. This error is specific to 3600-Family machines.

sys:%disk-error-device-check *Variable*

The disk unit is in a *device fault*, also called a *device check*, state. Device faults indicate a write to a write-protected drive or a malfunction in the disk system. If the fault was caused by a write to a write-protected drive, an error is signalled. This error is specific to 3600-Family machines.

sys:%disk-error-seek *Variable*

An error was detected during a seek. This can occur if an invalid disk address is specified in the transfer request, or if the disk system malfunctions. Most disk drive specifications allow for a small percentage of errors generated by seeks.

sys:%disk-error-search *Variable*

The disk block addressed by a disk transfer could not be found. This can occur if the addressed track on the disk is improperly formatted, if the disk address is invalid, or if the disk selected the wrong track.

sys:%disk-error-overrun *Variable*

The disk attempted to transfer data faster than the machine could accommodate. This error is expected to occur occasionally, due to conflicts when multiple I/O devices attempt to access memory simultaneously.

sys:%disk-error-ecc *Variable*

The data read from the disk has at least one invalid bit. The disk error recovery logic first attempts to correct the data. If the correction fails, it retries the transfer several times before signalling an unrecoverable ECC error. The disk array contains the incorrect data that was read from the disk for the block generating the ECC error.

sys:%disk-error-data *Variable*

The disk controller detected some form of data corruption other than a disk ECC error, such as a checksum error or a datapath parity error.

Disk Error Meters

These meters are updated when the disk system detects an error, including errors from which it automatically recovers. Meters that are primarily affected by system mode transfers are not included here. Most of these meters can be inspected with the Peek utility; press SELECT P and click Left on [Meters].

The value of the following meters is the number of:

si:*count-total-disk-errors* *Variable*

All types of disk errors.

si:*count-disk-select-errors* *Variable*

sys:%disk-error-select errors.

si:*count-disk-search-errors* *Variable*

sys:%disk-error-search errors.

si:*count-disk-overruns* *Variable*

sys:%disk-error-overrun errors.

si:*count-disk-ecc-errors* *Variable*

sys:%disk-error-ecc errors.

si:*count-disk-seek-errors* *Variable*

sys:%disk-error-seek errors.

FEP File System and Disk Streams

Disks on Symbolics computers contain a file system called the *FEP file system*. The entire disk is divided up into *FEP files* (that is, files of the FEP file system). FEP files have names syntactically similar to those of the files in the Symbolics computer's own local file system. However, the FEP file system and the Lisp Machine File System (LMFS) are completely distinct.

The FEP file system was designed to contain a small number of files that do not get created or deleted very often. It is not intended to contain active files. The FEP file system manages the disk space available on a disk pack, grouping sets of data into names structures called *FEP files*. A single FEP file system cannot extend beyond a single disk pack; each disk pack has its own separate FEP file system.

FEP file systems support multiple file versions, soft deletion, and expunging. They also use hierarchical directories.

Although the FEP is the *front-end processor*, the FEP file system is managed by the main Lisp processor. It is called the FEP file system because the FEP can read files stored in the FEP file system. For example, the FEP uses the FEP file system for booting the machine and running diagnostics.

The need to allow the FEP to access FEP files — while at the same time, allowing the rest of the system to use them — imposes these constraints on the design of FEP file systems:

- The internal data structure of files within FEP file systems must be simple enough to permit the FEP to read them.
- A small amount of concurrent access by both the FEP and Lisp must be allowed.
- A FEP file's data blocks need a high degree of locality on the disk, to minimize access time.
- FEP file systems must be reliable; the FEP needs to use them for basic operations, such as the running of diagnostics, and the booting of each machine.

Symbolics computers can have more than one local disk, and each machine's FEP can access all of them. Currently, hardware limits the maximum number of any one 3600-family machine's disks to eight. MacIvory machines have a maximum peripheral device limit of seven.

The form FEP: refers to the disk (by default) from which the current world was booted. Disk 0 is usually the default, so typing FEP: is usually equivalent to typing FEP0: . Besides using the default, you can specify disks explicitly, using forms such as FEP1: or FEP7: .

FEP File Systems on 3600-Series and XL400 Systems

Each 3600-family or XL400 disk must have a FEP file system on it that describes the disk space available on it.

Each disk unit is presumed to contain one FEP file system. FEP file systems are named FEP*n* (where *n* is the disk unit number on which the FEP file system resides).

This scheme allows gaps in the sequence of FEP file system names. For example, a machine with disk unit 0 and disk unit 2 (but no disk unit 1) has FEP file systems named FEP0 and FEP2 (but none named FEP1).

FEP File Systems on MacIvory Systems

MacIvory systems share disk space (swap space) between the Ivory and Macintosh processors. In order for the Ivory processor to access a Macintosh's disk to find a world load, for example, that disk must have at least one Ivory partition on it. (Symbolics recommends that you limit the number of Ivory partitions on each disk to one.)

Ivory partitions represent disk space to which the Macintosh processor does not have access. Each Ivory partition must contain a FEP file system that describes the disk space available in it.

Each time you power up or boot a MacIvory, the system checks the disk from which you booted. Next, it checks the remaining disks, according to their respective Small Computer Serial Interface (SCSI) bus priorities.

The first Ivory partition that the system finds is presumed to contain the FEP file system named FEP0. Any remaining Ivory partitions are presumed to contain the FEP file systems named FEP1, FEP2, and so on. This scheme does not allow gaps in the sequence of MacIvory FEP file system names.

FEP File Systems on Symbolics UX-Family Systems

UX-family systems share disk space (swap space) between the Ivory and Sun processors. In order for the Ivory processor to access a Sun's disk, to find a world load for example, that disk must have at least one partition on it.

Ivory partitions are represented as large UNIX files in the Sun file system, known as FEP partition files. Each FEP partition file must contain a FEP file system that describes the disk space available in it.

Each time you start UX-family life support (usually as part of the UNIX startup process), it reads a configuration file that describes the resources available to Ivory. This configuration file includes the names of the FEP partition files available to each UX-family machine. The first FEP partition file specified is presumed to contain the FEP file system named FEP0. Any remaining FEP partition files are presumed to contain FEP file systems named FEP1, FEP2, and so on. This scheme does not allow gaps in the sequence of UX-family machine FEP file system names.

FEP Pathnames

FEP pathnames can include references to a host, disk-unit, directory, filename, file type, and version. Separate the host-name from the rest of a file pathname by using a vertical bar (you can see this in the example that follows).

Delimit FEP pathnames like this:

```
Picasso|FEP0:>directory-name>filename.type.version
```

Pathname components are:

Host — Specifies which machine's FEP file system you are referencing. The default is the local machine.

Disk-unit — Specifies the disk unit number on which the local host's FEP file system resides. The initial default is FEP0. Later, the default becomes the local disk unit from which the world was booted. Netbooting doesn't change the default. Symbolics suggests that you specify the disk-unit number explicitly, since the default may be different for different worlds.

Directory — Indicates the name of the FEP file system directory (directory names cannot exceed 32 characters). There is no limit on the total length of a hierarchical directory specification.

Filename — Indicates the name of the FEP file (filenames cannot exceed 32 characters).

File type — Indicates the type of the FEP file (file types cannot exceed 4 characters).

Version — Indicates the version number of the FEP file (this must be a positive integer or the word "newest").

Note: Although you can access FEP files on other hosts from Genera, the FEP has access only to the local host.

For information about Ivory-based machines and the host pathname syntax for them, see the section "Accessing the Macintosh File System".

FEP File Types

By convention, the FEP file system uses the following extensions to deliniate file types:

boot — Files with the .boot extension contain commands that can be read and executed by the FEP.

load — Files with the .load extension contain a world load image, (sometimes called a band) for Symbolics 3600-family machines. Files with the `.lod` extension can only be copied between Symbolics 3600-family machines.

ilod — Files with the .ilod extension contain a world load image, (sometimes called a band) for Ivory machines. Files with the `.ilod` extension can only be copied between MacIvory, XL400, and Symbolics UX-family machines.

mic — Files with the .mic extension contain a microcode image, plus the contents of other internal high-speed memories that are initialized when Symbolics 3600-family machines are booted. For example, >3640-mic.mic.428 contains version 428 of the microcode for 3640 and 3670 machines.

fspt — In order to use the local Lisp Machine File System (LMFS), Lisp must have access to the File System Partition Table (FSPT). The File System Partition Table is contained within a FEP file named `fspt.fspt` that lists the LMFS partitions.

file — Files with the .file extension are Lisp Machine File System (LMFS) partitions.

page — Files with the .page extension are used exclusively as virtual memory swap space during the current boot session.

flod — Files with the .flod extension are FEP overlay (flod) files. Such files contain binary code (FEP software).

fep — Files with the .fep extension are FEP-specific; they contain information about the organization of fep files on the disk.

Note: Since FEP-specific files are system files, not user files, they should not be written to by user programs.

>free-pages.fep — This file describes which blocks on the disk are free.

>bad-blocks.fep — This file lists all of the blocks that contain media defects.

>sequence-number.fep — This file contains the highest sequence number in use. The FEP file system uses sequence numbers to uniquely identify files. (These help to rebuild the file system, should a catastrophic disk failure occur.)

>disk-label.fep — This file contains the disk pack's physical disk label. The label is used to identify the pack, and to describe its characteristics.

>kernel.fep — This file exists only on Ivory-based machines. It contains the FEP software which, on Symbolics 3600-family machines, resides in EPROM.

>reserve.fep	This file is reserved for use by Symbolics software.
>unique-id.fep	This file is reserved for use by Symbolics software.

dir — Files with the .dir extension are FEP subdirectories. Use the Show Directory FEP command or the Command Processor (CP) Show Directory Command to see the contents of FEP subdirectories. For more information about these commands,

- See the section "Show Directory FEP Command".
- See the section "Show Directory Command".

Note: Since the directory `>root-directory.dir` is a system file, not a user file, it should not be written to by user programs.

Disk Streams

Disk streams are I/O streams that, when opened, provide read and/or write access to FEP files. For more information about streams, see the section "Types of Streams". Also see the section "Stream Operations". Disk streams are opened by the **open** function. When disk streams are opened with a **:direction** keyword of **:input** or **:output**, they read or write bytes, buffering the data internally as required.

When disk streams are opened with a **:direction** keyword of **:block**, they can read and/or write specified disk blocks. Block mode disk streams address blocks with block numbers relative to the beginning of a file, starting at file block zero.

The **open** function also recognizes these keywords:

:if-locked — Specifies the action to be taken if a file is locked. This keyword is ignored for files on remote hosts. The **:if-locked** keyword itself recognizes two keywords:

:error	Signals an error. This is the default.
:share	Opens the specified file even if it is already locked, incrementing the file's lock count. This mode permits multiple processes to write to the same file concurrently. (For more information about FEP file locks, see the section "FEP File Locks".)

:number-of-disk-blocks — The number of disk blocks to buffer internally if the **:direction** keyword is **:input** or **:output**. The default is two. This keyword

is ignored for other values of **:direction** and for files on remote hosts.

Disk Stream Messages

All disk streams handle the messages described within this section.

For information about the additional messages that input and output disk streams can handle, see the section "Input and Output Disk Streams".

For information about the additional messages that block disk streams can handle, see the section "Block Disk Streams".

For more information about disk stream operations in general, see the section "File Stream Operations". Also see the section "General-Purpose Stream Operations".

:grow &optional *n-blocks* &key **:map-area :zero-p** *Message*

Allocates *n-blocks* of free disk blocks and appends them to the FEP file. The value of *n-blocks* defaults to one. If **:zero-p** is true the new blocks are filled with zeros; otherwise, they are not modified. The return value of **:grow** is the file's data map. The format of the data map is provided within the description of **:create-data-map** (in this section). The value of **:map-area** is the area to allocate the data map in, which defaults to **default-cons-area**.

:allocate *n-blocks* &key **:map-area :zero-p** *Message*

Ensures that the FEP file is at least *n-blocks* long, allocating additional free blocks as required. Returns the file's data map. The format of the data map is provided within the description of **:create-data-map** (in this section). **:map-area** specifies the area to create the data map in, and defaults to **default-cons-area**. The newly allocated blocks are filled with zeros if **:zero-p** is true. **:zero-p** defaults to **nil**.

:file-access-path *Message*

Returns the disk stream's file access path. You can find out on what unit number a FEP file resides, like this:

```
(send (send stream :file-access-path) :unit)
```

:map-block-no *block-number grow-p* *Message*

Translates the relative file *block-number* into a disk address, and returns two values: the first value is the disk address, and the second is the total number of disk blocks, starting with *block-number*, that are in consecutive disk addresses. *grow-p* specifies whether the file should be extended if *block-number* addresses a block that does not exist. When *grow-p* is true, free disk blocks are allocated and ap-

pended to the FEP file to extend it to include *block-number*. Otherwise, if *grow-p* is false, **nil** is returned if *block-number* addresses a block that does not exist.

:create-data-map &optional *area* *Message*

Returns a copy of the FEP file's data map allocated in area *area*, which defaults to **default-cons-area**. A FEP file data map is a one-dimensional **art-q** array. Each entry in the file data map describes a number of contiguous disk blocks, and requires two array elements. The first element is the number of disk blocks described by the entry. The second element is the disk address for the first block described by the entry. The array's fill-pointer contains the number of active elements in the data map times two.

:write-data-map *new-data-map disk-event* *Message*

Replaces the file's data map with *new-data-map*. *disk-event* is the disk event to associate with the disk writes when the disk copy of the file's data map is updated. This message overwrites the file's contents and should be used with caution.

Input and Output Disk Streams

Input and output disk streams read and write bytes of data (respectively), starting at the current byte position in the FEP file and updating byte position as the data is read.

Bytes of data are stored in buffers internal to the stream. The **:number-of-disk-blocks** keyword to the **open** function controls how many disk blocks the internal buffers can hold.

When the current pointer moves beyond a disk block boundary, the buffered disk block is written to the file for an output stream, or the next unbuffered block is read from the file for an input stream. Output streams also write out all buffered disk blocks when the stream is sent a **:close** message without an **:abort** option.

See the section "Disk Stream Messages". In addition to the messages described in that section, input and output disk streams support standard stream protocols for input and output. For more information about standard stream protocols, see the section "General-Purpose Stream Operations".

Block Disk Streams

Block disk streams can both read and write disk blocks at specified file block numbers. A file block number is the relative block offset into the file. The first block in the file is at file block number zero, the second is at file block number one, and so on.

Block disk streams do not buffer any blocks internally and can be used only on the local FEP file system. See the section "Disk Stream Messages". In addition to the messages described in that section, block disk streams support the following messages:

:block-length *Message*

Returns the length of the FEP file in disk blocks.

:block-in *block-number n-blocks disk-arrays* &key **:hang-p :disk-event** *Message*

Causes the disk to start reading data from the disk into the *disk-arrays*, starting with the file at *block-number*, and continuing for *n-blocks*. *Disk-arrays* is a list containing one or more disk arrays. The value of *n-blocks* is the number of disk blocks to read. When *n-blocks* is greater than one, each *disk array* is completely filled before using the next one.

Disk array checkwords are reserved for use by the FEP file system. Unused disk arrays (or portions of them) remain unmodified.

See the section "Disk System Definitions and Constants".

When the value of **:hang-p** is true, which it is by default, **:block-in** waits for all the reads to complete before returning. If the value of **:hang-p** is false, **:block-in** returns immediately upon enqueuing the disk reads without waiting for completion. In this case, all *disk-arrays* and the *disk-event* must be wired before sending the **:block-in** message, and must remain wired until the disk reads complete.

If the **:disk-event** keyword is supplied, its value is the disk event to associate with the disk reads. Otherwise the **:block-in** message allocates a disk event for its duration. A **:disk-event** must be supplied when **:hang-p** is false.

:block-out *block-number n-blocks disk-arrays* &key **:hang-p :disk-event** *Message*

Causes the disk to start writing the data in the disk arrays in *disk-arrays* onto the disk, starting with the file block number *block-number*, and continuing for *n-blocks*. The arguments to the **:block-out** message are identical to those of the **:block-in** message.

FEP File Properties

FEP file properties store information about FEP files (such as when they were last written, and whether they can be deleted).

File properties are read by the **fs:file-properties** function, and modified by the **fs:change-file-properties** function. The function **fs:directory-list** returns the file properties of several files at once.

The following file properties can be both read and modified:

:creation-date	The universal time at which a file was last written. (See the section "Dates and Times".)
:author	The user-ID of the last writer to a file: a string.
:length-in-bytes	The length of a file, expressed as an integer.

:deleted	When **t**, a file is marked as deleted. Disk space is not reclaimed until you expunge the directory in which a deleted file resides.
:dont-delete	When **t**, attempting to delete or overwrite a file signals an error. When **nil**, indicates that a file can be written to or deleted.
:comment	Written comments displayed in brackets: a string.

These file properties are returned by the **:properties** message. They cannot, however, be modified by **:change-properties**:

:byte-size	The number of bits in a byte. The value of this property is always eight.
:length-in-blocks	The block length of a file expressed as an integer.
:directory	When **t**, the file is a directory, otherwise **nil**.

FEP File Locks

A FEP file is *locked* from the time it is opened for reading or writing until it is closed. If the **:direction** keyword is **:input**, the file is *read-locked*; if the **:direction** keyword is **:output** or **:block**, the file is *write-locked*.

When the **:if-locked** keyword is **:error** (the default), a file that is read-locked can still be opened for reading but signals an error if opened for writing. This permits multiple readers to access (read) a file concurrently, while prohibiting writes to it.

When the **:if-locked** keyword is **:share**, a read or write operation succeeds in opening the file even if the file is already read- or write-locked.

An expunge operation on an open file that is either read- or write-locked will fail. If expunging a directory fails to expunge a deleted file, close the file, and expunge the directory again.

Disk Performance

You can improve the disk performance of a program by overlapping the disk transfers with computation and by reducing the *disk latency* by grouping contiguous transfers together.

The disk latency is the amount of time required by the disk unit to transfer a number of disk blocks. The minimum disk latency is the absolute lower bound on the time required to transfer a number of blocks; if shorter transfer times are required, a higher blocking factor or a faster disk unit is required. The software overhead can be determined by subtracting the minimum disk latency from the total time to transfer a number of blocks.

You overlap transfers with computation by specifying that a transfer request should not wait for the transfers to actually complete before returning. Computa-

tions can then continue while the disk is transferring the data. When your program actually requires data, the process can wait for the disk transfer to complete.

For example, if data is to be read from one block on the disk and then written to another block, the read request can be immediately followed by the write request without waiting for the read to actually finish, since disk transfers are always performed in the order in which they were enqueued. The time required to read and write the data is reduced since the write transfer can be enqueued while the disk is performing the read, so by the time the read completes the disk can immediately start writing the block.

Disk latency can be reduced by enqueuing multiple disk transfers to consecutive disk addresses without waiting for completion between transfers. This permits the disk to perform multiple transfers on the same disk revolution, or at least with a minimum of seeking.

The equation below yields the approximate minimum disk latency for transferring N contiguous disk blocks.

Equation 1:

$$T_n = T_a + T_r/2 + NT_r/S + T_s\,6((A \bmod HS)+N-1)/HS7$$

Where:

T_n	Minimum time to transfer N blocks.
T_a	Average seek time.
T_r	Rotation time.
N	Number of blocks to transfer.
S	Number of blocks per track.
T_s	Average single cylinder seek time.
A	The disk block number. The **sys:%%dpn-page-num** field of the disk address.
H	Number of data heads, excluding any servo heads.
$6x7$	*Floor* of x. The truncated integer value of x.

The terms in Equation 1 account for the various phases of a disk transfer, where:

- The first term accounts for the average seek time to position the heads to the cylinder the first block resides on.
- The second term accounts for an average initial delay of half a rotation for the first block to be positioned under the disk heads.
- The third term yields the time to actually transfer N blocks of data.
- The last term yields the time spent seeking to adjacent cylinders.

The time required to switch heads is insignificant, since head switching time is small enough not to affect the disk latency. Enough space is provided on the disk, between the last and first blocks on a track, for the head switch to complete after the last block has been transferred but before the first block of the next track passes under the heads. No extra rotation delays are incurred.

The values of the constants used in Equation 1 can be found in Table 1 for some of the available disk drives. To find the values for drives that are not listed, check the disk specifications supplied in the manual shipped along with the disk drive.

Table 1: Selected Disk Specifications

	M2284	M2351	T-306	D2257
H	10	20	19	8
S	16	22	16	16
T_a	27ms	18ms	30ms	20ms
T_r	20.24ms	15.15ms	17.5ms	17.09ms
T_s	6ms	5ms	7.5ms	5ms

If N single block transfers are requested to consecutive disk blocks, Equation 1 becomes:

Equation 2:

$$T_n = T_a + NT_r/2 + NT_r/S + T_s6((A \bmod HS)+N-1)/HS7$$

Equation 2 shows that, in addition to the cost of not performing computations concurrently with disk transfers, the minimum disk latency is increased by an average of a half rotation per disk transfer when single block disk transfers are made to consecutive blocks, waiting for each transfer to complete. However, Equation 2 is true only if the position of the disk is random with respect to the disk block being accessed. For example, if single transfer requests are made to consecutive disk blocks without a delay between transfer requests, the minimum disk latency would be increased by a full rotation per transfer.

Examples of High Disk Performance

Initializing a FEP File

The following function is an example of how you can achieve high disk performance. It writes zeroes over an entire FEP file.

```
(defun zero-fep-file (file)
  ;; FILE should be an open block disk stream.
  ;; Allocate a disk array and disk event
  (using-resource (disk-array si:disk-array)
    (using-resource (disk-event si:disk-event)
      ;; Wire both the disk array and disk event into memory for the
      ;; duration of all the transfers.  This is required when
      ;; HANG-P is NIL.
      (si:with-wired-disk-array (disk-array)
        (si:with-wired-structure disk-event
          ;; Iterate over all blocks in the file enqueuing a
          ;; write without waiting for the write to complete.
          (loop for block-number below (send file :block-length)
                doing (send file :block-out block-number 1 disk-array
                            :disk-event disk-event
                            :hang-p nil))
          ;; Finally, wait for all the writes to complete before
          ;; unwiring and returning the disk array and disk event.
          (si:wait-for-disk-event disk-event))))))
```

The **zero-fep-file** function writes the same disk array over all the blocks in the file without waiting for each write to finish before enqueuing the next write. This minimizes the time required to zero the FEP file since the write transfers are enqueued concurrent with the disk actually writing the data, and the transfers are enqueued in ascending file block number order. The FEP file system attempts to make FEP files as contiguous as possible with the disk addresses ascending in file block number order, so **zero-fep-file** writes as many blocks as can fit on a sector in one disk rotation.

Copying FEP Files

The next examples show alternative algorithms for copying a FEP file, starting out with a slow but simple example and developing it into a much faster version.

The following function shows a simple way to copy a FEP file. To simplify the example, the *source-file* and *dest-file* must be complete file specifications, and file properties, including the byte length, are not copied.

(Note that none of these functions copies any of the file's properties, not even the length-in-bytes. In a real file-copying application, you might want to copy some of the properties.)

```
(defun slow-copy (source-file dest-file)
  (with-open-file (source source-file
                          :direction :block
                          :if-exists :overwrite)
    (with-open-file (dest dest-file
                          :direction :block
                          :if-exists :overwrite
                          :if-does-not-exist :create)
      ;; First preallocate the same number of disk blocks for the
      ;; destination file as is required by the source file.
      ;; Allocating many blocks at once is much faster than implicitly
      ;; allocating a block at a time, and results in better locality
      ;; on the disk.
      (send dest :allocate (send source :block-length))
      ;; Allocate a disk array to buffer the data and a disk event
      (using-resource (disk-array si:disk-array)
        (using-resource (disk-event si:disk-event)
          ;; Now iterate over all blocks in the source file, copying
          ;; the block to the destination file.
          (loop for block-number below (send source :block-length)
                do
                (send source :block-in block-number 1 disk-array
                      :disk-event disk-event)
                (send dest :block-out block-number 1 disk-array
                      :disk-event disk-event)))))))
```

While the **slow-copy** function is simple, it is also very slow. The reason for this is that the **:block-in** message waits for the disk read to complete before the **:block-out** message can be enqueued. This function can be sped up by over a factor of two and a half by supplying a **:hang-p** keyword with a value of **nil**, allowing the **:block-in** and **:block-out** messages not to wait for completion. For example:

```
(defun quick-copy (source-file dest-file)
  (with-open-file (source source-file
                          :direction :block
                          :if-exists :overwrite)
    (with-open-file (dest dest-file
                          :direction :block
                          :if-exists :overwrite
                          :if-does-not-exist :create)
      ;; First preallocate the same number of disk blocks for the
      ;; destination file as is required by the source file.
      (send dest :allocate (send source :block-length))
      ;; Allocate a disk array to buffer the data and a disk event
      (using-resource (disk-array si:disk-array)
        (using-resource (disk-event si:disk-event)
          ;; The disk array and disk event must be wired for the
          ;; duration of all the transfers.  When HANG-P is true, the
          ;; transfer functions automatically wire and unwire the disk
          ;; event and disk arrays.  But since this function specifies a
          ;; HANG-P of NIL for speed, it must do the wiring itself.
          (si:with-wired-disk-array (disk-array)
            (si:with-wired-structure disk-event
              ;; Iterate over all the blocks in the source file,
              ;; enqueuing reads and then enqueuing writes
              ;; to the destination file.
              (loop for block-number below (send source :block-length)
                    do
                    ;; Enqueue the source read without waiting for the
                    ;; transfer to actually complete.
                    (send source :block-in block-number 1 disk-array
                          :disk-event disk-event :hang-p nil)
                    ;; Enqueue the destination write while the
                    ;; source read is still in progress.  This does not
                    ;; have to wait for the read to complete since
                    ;; disk transfers are always performed in the
                    ;; order they were enqueued.
                    (send dest :block-out block-number 1 disk-array
                          :disk-event disk-event :hang-p nil))
              ;; Wait for all pending transfers to complete.
              (si:wait-for-disk-event disk-event)))))))
```

quick-copy has increased speed by overlapping disk requests with computation. This keeps the disk queue full, so that the disk is continually copying the file without having to stop and wait for the next disk transfer to be enqueued. But the disk is still reading a block, then seeking to the destination block, then writing a block, and seeking back to the next source block. Performance can be further enhanced by reducing the disk latency if both the source and destination files reside on the same disk unit.

The disk latency can be reduced by reading multiple source blocks, then seeking to the destination file and writing multiple destination blocks, eliminating disk seeks. Thus, the following function combines minimized disk latency (achieved by using a large blocking factor between seeks) with overlapped computations and disk transfers. The resulting speed is about three times faster than **quick-copy**, and seven times faster than **slow-copy**.

```
(defun fast-copy (source-file dest-file &optional (blocking-factor 20.))
  (with-open-file (source source-file
                          :direction :block
                          :if-exists :overwrite)
    (with-open-file (dest dest-file
                          :direction :block
                          :if-exists :overwrite
                          :if-does-not-exist :create)
      ;; First preallocate the same number of disk blocks for the
      ;; destination file as is required by the source file.
      (send dest :allocate (send source :block-length))
      (let ((disk-arrays (make-array blocking-factor)))
        ;; Allocate a disk event.
        (using-resource (disk-event si:disk-event)
          ;; The disk event must be wired for the duration of all the
          ;; transfers.
          (si:with-wired-structure disk-event
            (unwind-protect
              (progn
                ;; Allocate and wire the disk arrays.  The disk arrays
                ;; must be wired for the duration of the disk transfer.
                (dotimes (i blocking-factor)
                  (let ((disk-array (allocate-resource 'si:disk-array)))
                    (si:wire-disk-array disk-array)
                    (setf (aref disk-arrays i) disk-array))
                (loop
                  with blk-length = (send source :block-length)
                  for start-blkn from 0 by blocking-factor below blk-length
                  do
                  ;; Enqueue the source reads without waiting for the
                  ;; transfers to actually complete.
                  (loop for blkn from start-blkn below blk-length
                        for array being the array-elements of disk-arrays
                        do
                        (send source :block-in blkn 1 array
                              :disk-event disk-event :hang-p nil))
                  ;; Enqueue the destination writes while the
                  ;; source reads are still in progress.  This does not
                  ;; have to wait for the reads to complete since
                  ;; disk transfers are always performed in the
                  ;; order they were enqueued.
                  (loop for blkn from start-blkn below blk-length
                        for array being the array-elements of disk-arrays
                        do
                        (send dest :block-out blkn 1 array
                              :disk-event disk-event :hang-p nil))))
              ;; Wait for all pending transfers to complete.
              (si:wait-for-disk-event disk-event)
```

```
;; Finally, return the disk arrays.
(loop
  for disk-array being the array-elements of disk-arrays
  when disk-array
  do
  (when (si:structure-wired-p disk-array)
    (si:unwire-disk-array disk-array))
  (deallocate-resource 'si:disk-array disk-array)))))))))
```

This example still does not include some functionality that would make it complete. However, it does illustrate how to use disk-events effectively. To make it a reasonable function, other features, such as preserving file properties, offering pathname defaulting and merging, and using **unwind-protects**, should be included.

Disk and FEP File System Utilities

Initializing a Disk Unit

Before a disk unit can be used, it must be formatted and have a valid disk label. Disks are formatted by the FEP, which can also write the label and initialize the FEP file system from cartridge tape. See the section "The Front-End Processor". In addition, the following functions are available:

si:write-fep-label *unit* *Function*

Writes the disk label for unit number *unit*, interactively asking for any necessary information. After the label is written the disk unit is left mounted.

si:edit-fep-label &optional *(unit* **0***)* &optional *unit* *Function*

Permits the disk label of the disk unit *unit* to be edited by exposing a choose variable values window. *unit* defaults to disk unit 0.

si:read-fep-label *unit label-array disk-event* *Function*

Reads the disk label for unit *unit* into the disk array in *label-array*, associating the read transfers with *disk-event* in case of an error.

Mounting a Disk Unit

Disk units can be *mounted* either by the FEP or by Lisp. See the section "The Front-End Processor". When a disk unit is mounted, its disk label is read and the system's disk unit tables are updated. A disk unit must be mounted before it is available for disk transfers.

storage:mount-disk-unit *unit* *Function*

Makes the disk unit available to the Lisp system by reading its label and updating the system's disk unit tables. *unit* is the unit number to mount, and must be the address of an online disk unit.

Verifying a FEP File System

The following function checks for and fixes inconsistencies in the FEP file system.

si:verify-fep-filesystem &optional (*unit* **0**) &key (*fix-checkwords* **':ask**) *Function*

Checks the FEP file system on disk unit *unit*, which defaults to zero, reporting any detected inconsistencies and offering to correct certain types of failures. If **:fix-checkwords** is **:ask** (the default), you are prompted if anything has to be fixed; the other options are **:yes** (always fix), **:no** (never fix), **:silently** (always fix without a message), and **:inform-only** (send messages only, do not fix, do not ask).

si:print-fep-filesystem &optional (*unit* **0**) *Function*

Outputs a textual description of the FEP file system on disk unit *unit*. The default value of *unit* is 0.

si:resequence-fep-filesystem &optional (*unit* **0**) *Function*

Resequences all the FEP files in the FEP file system on unit *unit*. The value of *unit* defaults to zero. The files are resequenced by iterating over all files in the FEP file system and assigning each a unique sequence number, starting with zero. Sequence numbers are used by the FEP file system to check for consistency and identify pages in the file system. They can be used to rebuild the FEP file system or find missing files in case of a catastrophic failure.

The Serial I/O Facility

Introduction to Serial I/O

Symbolics computers have a serial input/output facility, which uses the EIA RS-232 protocol to receive and transmit serial data. Many computer peripherals can communicate using the RS-232 protocol, and so can be connected to a Symbolics computer through this facility. This chapter explains the capabilities of the facility, gives a brief description of the hardware performing the serial I/O and how to interface to it, and describes the software driving that hardware.

Before reading this chapter, you should be familiar with the basic concepts of serial data communication, including the RS-232 standard. You should also be familiar with Symbolics Common Lisp, which is the systems programming language for

Symbolics computers. In particular, you should understand what *streams* are. See the section "Streams".

The Serial I/O Facility was rewritten for Genera 7.4 Ivory. It now works as documented.

You should note the following minor changes:

- The **uss** package has been removed. Users should not have been using symbols in this package. However, if you are using any of these symbols, such as **uss::*serial-interfaces*** and **uss::serial-port-lock-holder**, you should change your code to use the documented interfaces.

- It used to be possible to create your own flavor of serial stream and supply all the options to **si:make-serial-stream** in the **:default-init-plist** option to **defflavor**. Now you can supply stream-specific options only to **:default-init-plist**; others must be supplied with **:after** methods. See the section "Creating Your Own Flavor of Serial Stream".

- It should be noted that the serial chip used by the XL400 has a slightly different behavior with respect to parity checking than the chips used in earlier Symbolics machines. The 68562 chip checks parity *before* checking for XON/XOFF characters, the older chip checked *after*. This means that you have to be more careful about parity settings than previously. If the parity is wrong, XON/XOFF flow-control does not work.

Hardware Description for Serial I/O

This section gives a brief description of the hardware that performs serial I/O on Symbolics computers. You do not have to understand everything in this section to use the serial I/O facility.

Overview of Serial I/O Hardware

Symbolics computers have different numbers of serial ports. Here is a list of machine models and the number of serial ports each supports. 3600-family and XL400 machine serial ports have integer names starting with 0.

3600	Supports four serial I/O ports. Three ports are located on the bulkhead at the back of the processor. The fourth port is located in the rear of the console.
3650	Supports three serial I/O ports. Two ports are located on the bulkhead at the back of the processor. The third port is located in the rear of the console.
3620	This machine model is delivered with only one bulkhead port; others can be ordered from Symbolics.

3610AE	This machine model does not come standard with any bulkhead ports; one can be ordered from Symbolics.
XL400	This machine comes with one bulkhead port. You can add I/O boards to get additional ports. Since the XL400 has a VME bus, there also the possibility of having a VME board with one or more additional serial ports.
UX400S	This machine uses the serial ports on the SUN in which it is embedded. The ports are referenced by their UNIX names, for example, `:unit "/dev/ttyA"`.
MacIvory	This machine has two serial ports: the printer port, which is unreliable for incoming data at speeds above 300 baud, and the modem port. They are referenced as `1` and `2`.

The external data communication signals appear on RS-232 25-pin D-type connectors. In the 3600 machine model, all serial I/O communication is controlled by the computer's FEP. In the 3610AE, 3620, and 3650 machine models, serial I/O communication is through the console SLB (a small board behind the FEP) and the Z8530 (a serial chip). This speeds up serial I/O communication.

The RS-232 protocol provides for communication between Data Circuit Terminating Equipment (DCEs, also known as "data sets"; for example, modems), and Data Terminal Equipment (DTEs, also known as "data terminals"; for example, computer terminals, computers, or most devices that use serial lines).

The single console port is configured differently from ports on the bulkhead:

On the 3600 machine model, each of the three ports on the bulkheads is a DTE. You can connect a bulkhead serial port directly to a DCE, but if you want to connect the serial port to a DTE, you must supply a *null modem*.

In contrast, the console port is a DCE. You can connect the console serial port directly to a DTE, but if you want to connect the serial line to a DCE, you must supply a *null terminal*.

For example, a Kanji tablet is configured as a DCE. You can connect a Kanji tablet directly to a bulkhead port (a DTE), but you must supply a null terminal to connect a Kanji tablet to the console port (a DCE).

Console Serial I/O Port

The external data communication signals appear on one female RS-232 25-pin D-type connector in the rear of the console of Symbolics 3600-family computers. The console serial I/O port is labelled "RS-232".

The correspondence between connector pins and RS-232 signals is given in Table 1.

Note: The cable for modems is **not symmetrical**. If it doesn't seem to work, swap ends and try again.

Console connector pin	*RS-232 signal*
2	Transmitted Data [Input]
3	Received Data [Output]
4	RTS (Request To Send) [Input]
5	CTS (Clear To Send) [Output]
6	DSR (Data Set Ready) [Output]
8	DCD (Data Carrier Detect) [Output]
20	DTR (Data Terminal Ready) [Input]
1	Chassis Ground
7	Signal Ground

Table 7. Assignment of RS-232 Signals to Pins

To build a cable that includes a null terminal for asynchronous communications, follow the wiring instructions in Table 2. Both ends of the cable should be male 25-pin RS-232 connectors.

Pin at back of the console (DCE)		*Pin at remote end (DTE)*	
2	Transmit data (to DCE)	3	Transmit data (from DTE)
3	Receive data (from DCE)	2	Receive data (to DTE)
4	Request to send (to DCE)	5	Clear to send (to DTE)
5	Clear to send (from DCE)	4	Request to send (from DTE)
7	Signal ground	7	Signal ground
8	Carrier detect (from DCE)	20	Data Terminal ready (from DTE)
18	Special DTE-DSR input	6	Data set ready (to DTE)
19	Special DTE-RI input	22	Ring Indicator (to DTE)
20	Data terminal ready (to DCE)	8	Carrier detect (to DTE)

Table 8. Assignment of RS-232 Signals to Pins in Asynchronous Null Terminals

Note: The console serial I/O port has architectural limitations that currently prevent the console processor from reporting errors during transmission. For this reason, Symbolics cannot recommend use of this port for applications that require a completely reliable data stream.

The console serial I/O port handles graphics and kanji tablets with no difficulty. (However, leaving the tablet puck or stylus on the pad makes the tablet send a continuous stream of coordinates when connected to any serial port. Therefore, users should keep the puck or stylus off the tablet when not actually using it.)

Any user protocol that performs its own end-to-end reliability checking, such as Kermit or XModem, is also acceptable.

In its own operations, Symbolics uses LGP2 printers connected to the console serial I/O port with no difficulty, but some customers have had problems doing the same thing.

There is no guarantee that every Print Spooler request is correctly handled. Screen dumps are faster when the printer is connected to the bulkhead port than to the console port.

Bulkhead Serial I/O Ports

The external data communication signals appear on three RS-232 25-pin D-type connectors on the rear bulkhead (in the back of the processor).

The gender and labeling of these connectors varies with the processor model:

- The 3600 I/O bulkhead presents 3 female connectors labelled "EIA 1", "EIA 2", and "EIA 3". (The male connector labelled "EIA 4" is not a serial port at all, but the connection to an inboard Vadic VA3450 modem, if present.
- The 3670 I/O bulkhead presents 3 male connectors labelled "EIA 1", "EIA 2", and "EIA 3".
- The 3640 I/O bulkhead presents 3 male connectors labelled "SERIAL 1", "SERIAL 2", and "SERIAL 3".

The correspondence between connector pins on the rear bulkhead and RS-232 signals is given in Table 1.

Rear bulkhead connector pin	*RS-232 signal*
2	Transmitted Data [Output]
3	Received Data [Input]
4	RTS (Request To Send) [Output]
5	CTS (Clear To Send) [Input]
6	DSR (Data Set Ready) [Input]
8	DCD (Data Carrier Detect) [Input]
20	DTR (Data Terminal Ready) [Output]
1	Chassis Ground
7	Signal Ground

Table 9. Assignment of RS-232 Signals to Pins

To build a cable that includes a null modem for asynchronous communications, follow the wiring instructions in Table 2.

One side	*Other side*	*RS-232 signal*
3	2	Data Out (from data set to terminal)
2	3	Data In (from terminal to data set)
5	4	RTS (Request To Send)
4	5	CTS (Clear To Send)
20	6	DSR (Data Set Ready)
20	8	DCD (Data Carrier Detect)
6	20	DTR (Data Terminal Ready)
8	20	DTR (Data Terminal Ready)
1	1	Chassis Ground
7	7	Signal Ground

Table 10. Assignment of RS-232 Signals to Pins in Asynchronous Null Modems

Note that this null modem is suitable only for asynchronous communications; a synchronous null modem is considerably more complex.

When using the 3600-family computer with a device that does not supply RS-232 modem control signals, it is necessary to supply Clear To Send and Data Carrier Detect inputs to the 3600-family computer, for example by jumpering pin 4 to pin 5, and pins 6, 8, and 20 together. This should be done in the cable or in the device connector, *not* in the 3600-family computer's connector or inside the 3600-family computer. See tables 9 and 10.

The Serial I/O Stream

The function of the serial I/O facility is to receive and transmit data over a serial communications channel. The unit of communication is the *character*; each character is represented as a binary number. The facility has two independent parts: a *receiver*, which receives a sequence of characters from the external device, and a *transmitter*, which transmits a sequence of characters to the external device.

A Symbolics Common Lisp program uses the facility through an I/O stream. The output operations, such as **:tyo**, send characters to the transmitter and from there to the external device; the input operations, such as **:tyi**, read characters from the receiver, which gets them from the external device. In addition to regular I/O operations, the serial I/O stream also supports special operations that examine and alter parameters of the serial I/O facility. To perform serial I/O, a program should first get the serial I/O stream by calling the function **si:make-serial-stream**, set-

ting up the parameters of the serial I/O facility as it needs them; then it can use normal stream operations to read and write characters. When the program is done with the serial I/O stream, it should close it; programs that use the serial I/O stream should include an **unwind-protect** form whose cleanup handler closes the stream. The **with-open-stream** special form is a good way to do this when the entire lifetime of the stream is to be enclosed in the body of one Symbolics Common Lisp form. Closing the stream frees up a buffer in main memory, frees up use of the port, and disables interrupts.

The serial I/O stream is different from most streams in that the characters you send to it and get from it are probably *not* interpreted as being in the Symbolics character set. Of course, the interpretation of the characters depends completely on the external device, but most devices that are likely to use serial communications use the standard ASCII character set. You can tell the stream whether or not to convert between ASCII characters and Symbolics characters.

The serial I/O stream is also different from some streams in being buffered on the output side. If you send characters to the serial stream using, for example, **:tyo** or **:string-out**, the characters are placed into a buffer for eventual transmission over the serial line. They are not actually transmitted until the buffer fills up, the serial stream is closed, or a **:force-output** operation is done on the stream. The **:force-output** option to **si:make-serial-stream** causes characters to be transmitted immediately; this makes the serial stream easier to use but degrades its performance.

The serial I/O stream has several parameters. Each parameter is denoted by a keyword symbol. These keywords are passed to the **si:make-serial-stream** function and to the **:get** and **:put** operations to specify which parameter the caller is interested in. (Some parameters make sense only when creating a stream, or affect the flavor of the stream; these parameters are not valid for **:get** and **:put**.) For descriptions of the parameters:

See the section "Parameters for Serial I/O".

si:make-serial-stream &rest *options* *Function*

Initializes the serial I/O facility and returns the serial I/O stream.

options are alternating keyword symbols, naming parameters, and initial values for those parameters. They let you initialize parameters when you start using the serial I/O stream. You can change most of them later with the **:put** operation.

si:make-serial-stream, which accesses a serial line, causes the accessing process to wait if the port unit requested is in use. The command `c-m-SUSPEND` allows you to invoke a restart handler to close a line that you believe has been left open by mistake.

For documentation of parameters for serial I/O: See the section "Parameters for Serial I/O".

Obsolete Parameters for si:make-serial-stream

Note: The following parameters for **si:make-serial-stream** are now obsolete but are still supported; use the generic function indicated instead of the parameter:

:clear-to-send
Use the **clear-to-send** generic function instead.

See the method **:clear-to-send**.

:request-to-send Defaults to **t**. If you want to change this to **nil**, use the **setf-request-to-send** generic function to do so. See the method **:setf-request-to-send**.

:data-terminal-ready
Defaults to **t**. If you want to change this to **nil**, use the **setf-data-terminal-ready** generic function to do so. See the method **:setf-data-terminal-ready**.

The serial I/O stream supports all standard stream operations. Of the optional input operations, it supports **:listen** and **:clear-input**; the latter is relevant because input from the serial port is buffered. There is also a **:reset** operation, which resets the state of the hardware. The **:tyi-no-hang** special-purpose operation is supported as well. The **:force-output** and **:finish** optional output operations are supported, since output is buffered.

The serial I/O stream also supports two nonstandard operations: **:put** and **:get**. These operations examine and alter various properties of the serial I/O facility.

(flavor:method :get si:serial-stream) *parameter* *Method*

parameter should be one of the symbols that name parameters of the serial I/O facility. This message returns the value of that parameter. See the section "Parameters for Serial I/O".

(flavor:method :put si:serial-stream) *parameter value* *Method*

parameter should be one of the symbols that name parameters of the serial I/O facility. The value of that parameter is set to *value*. See the section "Parameters for Serial I/O".

If you are using serial I/O streams, you might also be interested in the remote login facilities:

See the section "Using the Remote Login Facilities for ASCII Terminals".
See the function **neti:enable-serial-terminal**.

You can also use the following generic functions to manipulate the serial I/O stream:

send-break
Sends a message to the device to stop the output stream. When using this generic function, make sure that there is no output buffered in the stream by sending a **:force-output** message.

carrier-detect
Returns **t** if a carrier detect is asserted at the port. Otherwise, it returns **nil**.

setf-data-terminal-ready
If the value of this parameter is **t**, assert the DTR ("data terminal ready") signal; otherwise do not. The default is **nil**.

clear-to-send
Returns **t** if the external device is asserting the CTS ("clear to send") signal; otherwise it is not.

setf-request-to-send
Asserts a request RTS ("request to send") signal. The default is to send a stream from the port if value is non-**nil**.

data-set-ready
Returns **t** if asserted at the port. Otherwise, it returns **nil**.

ring-indicator
Returns **t** if asserted at the port. Otherwise, it returns **nil**.

Parameters for Serial I/O

This section lists all parameters of the serial I/O facility. For each parameter, it lists the keyword symbol, the meaning of the parameter, and the default value. A few parameters can be examined but not altered; they are so marked in their descriptions. Parameters whose functions are similar are grouped together.

Parameters from the following group are used *only* when the stream is being created, as arguments to **si:make-serial-stream**. You cannot use the **:put** operation with them, and you can use the **:get** operation only with **:unit**.

:unit This parameter says which of the serial ports to create a stream to. Its value can be an **1**, **2**, or **3** to indicate one of the three bulkhead ports (each of which is a DTE); or **0** to indicate the serial I/O port located at the back of the console (a DCE). On UX400 machines, it is the name of the port in UNIX, for example "`/dev/ttyA`". The default is **2**. For more information on the serial I/O ports: See the section "Overview of Serial I/O Hardware".

:ascii-characters If the value of this parameter is **t**, the serial stream is a Zetalisp character stream. The characters are translated from ASCII to the Symbolics internal character set on input, and to ASCII on output. If the value of this parameter is **nil**, the serial stream is a binary stream. Binary streams do not support **:lineout**, **:fresh-line**, and similar messages. The default is **nil**.

:flavor The value of this parameter is the flavor of stream to create. Normally, the value is computed automatically, based on the values of the **:ascii-characters** and **:force-output** parameters; this parameter is needed only if you want to use some special flavor that includes the serial stream flavors and other mixins.

:force-output If the value of this is **t**, a **:force-output** stream operation is done after every **:tyo** and every **:string-out**. If the value of this is **nil** output is not transmitted until the output buffer fills up, a **:force-output** is done explicitly, or the stream is closed (and the close mode is not **:abort**). The nonforcing mode is usually more efficient, although efficiency depends on the application. The default is **nil**.

The following group of parameters controls the format of the transmitted characters. It is important to set the parameters to be compatible with the external device, or else proper communication is impossible. These parameters apply to both the transmitter and the receiver.

:baud The data transmission rate, in bits per second. This an integer (in decimal): **300**, **600**, **1200**, **1800**, **2000**, **2400**, **3600**, **4800**, **7200**, **9600**, **19200** for 3600 family machines. The maximum is **9600** for 3600-family machines. Valid baud rates for XL machines are **50**, **75**, **110**, **134.5**, **150**, **200**, **300**, **600**, **1050**, **1200**, **2000**, **2400**, **4800**, **9600**, **19200** and **38400**. Note that **38400** is the maximum for XL machines. For embedded Ivory machines, it is the speed of the embedding host.

:number-of-data-bits
The number of bits in each character. This should be one of the following fixnums: **5**, **6**, **7**, or **8**. The default is **7** for 3600-family and XL400 machines. For embedded Ivory machines, it is the same as the embedding host.

:parity The kind of parity bit that should be sent. If the value of this parameter is **nil**, no parity bit is sent. If it is **:even**, even parity is transmitted. If it is **:odd**, odd parity is transmitted. The default is **:even**. This parameter also controls what kind of parity checking is done on received characters. The XL400 checks parity before checking for XON/XOFF flow control characters. For embedded Ivory machines, parity is the same as the embedding host.

:number-of-stop-bits
The number of "stop" bits transmitted after each character. It should be one of the following numbers: **1**, **1.5**, or **2**. The default is **1**.

The following parameters control error checking in the receiver. After a character is read by an input stream operation, the stream checks for error conditions detected by the receiver when the character arrived. If any of the enabled error conditions occurred, the stream signals an error. For embedded Ivory systems where the embedding host does not support all of these parameters, the serial I/O facility tries to map the parameters it receives to those the system supports. Any unsupported parameters are ignored.

:check-parity-errors

If the value of this parameter is **nil**, parity errors are ignored; if it is **t**, when a character with an error is read, and there is pending input, the parity error is buffered along with the incoming characters. After the pending input is read, the parity error is signalled. The default is **nil**. A parity error occurs when the parity of the data bits disagrees with the value of the received parity bit. This never happens if parity checking is not being used, that is, if the **:parity** option is **nil**. The XL400 checks parity before checking for XON/XOFF flow control characters.

:input-error-character

The value is a character to be substituted for any input character in which a parity error is detected. This is independent of the **:check-parity-errors** flag. If the value is **nil** (the default), the character is left alone.

:check-overrun-errors

If the value of this parameter is **nil**, over-run errors (low level errors) are ignored; if it is **t**, when a character with an error is read, and there is pending input, the over-run error is buffered along with the incoming characters. After the pending input is read, an over-run error is signalled. The default is **nil**. An over-run error occurs if input arrives faster than it can be read.

:check-framing-errors

If the value of this parameter is **nil**, framing errors are ignored; if it is **t**, when a character with an error is read, and there is pending input, the framing error is buffered along with the incoming characters. After the pending input is read, a framing error is signalled. The default is **nil**. The framing error occurs when the "stop" bit (the bit after all the data bits, and after the parity bit if parity is being checked) is not **1**. This indicates a line error, a baud rate mismatch between the external device and the receiver, or the sending of a "break".

:check-break When the value of this is **t**, if a break is detected in the input stream and there is pending input, the break is buffered along with incoming characters. After the pending input is read, the check break is signalled. The default is **nil**.

:check-overflow-errors

When the value of this is **t**, if a Lisp buffer overflow is detected in the input stream, when a character with an error is read, and there is pending input, the overflow error is buffered along with the incoming characters. After the pending input is read, an overflow error is signalled. The default is **t**.

:want-unsolicited-input-exceptions
When the value of this parameter is **t**, if there is input while the process is waiting for output in the Lisp buffer to complete, this causes an error to be signalled when the character is received while waiting for output to complete. The default is **nil**.

:timeout When the value of this parameter is **t**, specifies the amount of time, in 60th of a second, to wait before timing out while waiting for output to finish. This causes an error to be signalled in the Lisp buffer when the timeout expires. The default is **nil**.

The following parameters control the use of the XON/XOFF protocol. For embedded Ivory systems where the embedding host does not support all of these parameters, the serial I/O facility tries to map the parameters it receives to those the system supports. Any unsupported parameters are ignored.

:xon-xoff-protocol If this is **t**, output to the serial stream is flow-controlled using the ASCII XON/XOFF (Control-S/Control-Q) protocol. While the stream is transmitting characters, it checks the receiver to see if any characters have arrived. If an ASCII XOFF or Control-S character (octal 23, decimal 19) has arrived, transmission is stopped. Then the stream reads characters from the receiver until an ASCII XON or Control-Q character (octal 21, decimal 17) arrives, and then proceeds with the transmission.

This feature allows the external device to limit the rate at which characters are transmitted to it by the serial I/O facility. The default is **nil** (XON/XOFF feature not enabled).

Note: You can modify the Control-S/Control-Q characters with the parameters **:input-xoff-character** and **:input-xon-character**.

Interpretation of incoming XON/XOFF signals is done at interrupt level in the FEP, and is therefore quite fast. After an XOFF is received, the 3600-family computer ceases transmission after two or three characters (buffered in the multiprotocol chip).

You should note that the XL400 checks for parity before checking for XON/XOFF characters.

:generate-xon-xoff If the value of this parameter is **t**, then the serial port generates XON and XOFF controls itself. This can be used to accept input at high speed from devices that understand the XON/XOFF protocol. The default is **t**.

The XON and XOFF characters are transmitted directly by the FEP, so the response time is excellent. After the FEP transmits an XOFF, the device is required to cease transmission after no more than about 100 characters, so the device is not required to act very quickly.

On machine models 3610AE, 3620, and 3650, the microcde receive task wake-ups (interrupts) generate the XON and XOFF controls. This is slightly slower than the serial port generation, but it is still very fast. After the machine transmits the **:output-xoff-character**, the device is required to cease transmission after no more than 50 characters.

:rts-cts-protocol
When the value is **t**, specifies to use the hardware RTS/CTS protocol for transmitting to the device. The default is **nil**.

:generate-dtr When the value is **t**, specifies to use a hardware DTR/CTS protocol. The default is **nil**.

:input-xoff-character
The value is a character that is used to control flow of data from the external device to the Symbolics computer. It is sent by the Symbolics computer to suspend the flow of data when the **:generate-xon-xoff** flag is set. ASCII Control-S is **19**, and ASCII Control-Q is **17**. The default is **19**.

:input-xon-character
The value is a character that is used to control flow of data from the external device to the Symbolics computer. It is sent by the Symbolics computer to resume the flow of data when the **:generate-xon-xoff** flag is set. ASCII Control-S is **19**, and ASCII Control-Q is **17**. The default is **17**.

:output-xoff-character
The value is a character that is used to control flow of data from the Symbolics computer to the external device. It is used to suspend the flow of data when the **:xon-xoff-protocol** parameter is set. ASCII Control-S is **19**, and ASCII Control-Q is **17**. The default is **19**.

:output-xon-character
The value is a character that is used to control flow of data from the Symbolics computer to the external device. It is used to resume the flow of data when the **:xon-xoff-protocol** parameter is set. **:xon-xoff-protocol** parameter is set. ASCII Control-S is **19**, and ASCII Control-Q is **17**. The default is **17**.

Creating Your Own Flavor of Serial Stream

You can define your own flavor of serial stream with **defflavor**. However, you should be aware that you cannot give default values to system stream options using the **:default-init-plist** option to **defflavor**.

For example:

```
(defflavor my-serial-stream
   (my-option)
   (serial:serial-binary-stream)
  :default-init-plist :my-option t)
```

You can put your own options on the default-init-plist, but not the options to **si:make-serial-stream**. The list of options that are not allowed is:

:mode	**:flavor**
:force-output	**:ascii-characters**
:baud	**:unit**
:xon-xoff-protocol	**:generate-xon-xoff**
:input-xon-character	**:input-xoff-character**
:output-xon-character	**:output-xoff-character**
:number-of-data-bits	**:number-of-stop-bits**
:parity	**:rts-cts-protocol**
:check-breaks	**:check-parity-errors**
:check-framing-errors	**:check-overrun-errors**

You can use **:after** methods to give default values to these options.

Simple Examples: Serial I/O

The following two examples illustrate the use of the serial I/O facility. For further information on the function **si:make-serial-stream** and its parameters:

See the section "The Serial I/O Stream".
See the section "Parameters for Serial I/O".

Both examples below assume that the serial I/O port numbered 1 is hooked to an ASCII computer terminal operating on a normal RS-232 asynchronous connection at 300 baud, with one stop bit and odd parity. It types the characters "Hello there." on the terminal. A null modem is used between the serial port and the terminal, because both ends are acting as DTEs.

The first example illustrates creating a serial stream, saving the result in a variable, sending output to the stream, and closing the stream:

```
(setq ss (si:make-serial-stream
           :unit 1
           :baud 300
           :ascii-characters t
           :number-of-stop-bits 1
           :parity :odd
           :force-output t))
(send ss :string-out "Hello there.")
(close ss)
```

The second example uses **:with-open-stream**, thereby enclosing the entire lifetime of the serial stream in the body of one Symbolics Common Lisp form:

```
(defun type-greeting-message ()
  (with-open-stream (stream (si:make-serial-stream
                              :unit 1
                              :baud 300
                              :ascii-characters t
                              :number-of-stop-bits 1
                              :parity :odd))
    (send stream :string-out "Hello there.")))
```

You can also use the function **neti:enable-serial-terminal** to enable a terminal to communicate with a Symbolics computer: See the function **neti:enable-serial-terminal**.

Troubleshooting: Serial I/O

If you have trouble making your device communicate with the 3600-family computer through a serial port, there are several things to try:

- Make sure that the baud rate, the number of data bits, the parity checking, and the number of stop bits are set the same way on the device as they are in your serial stream parameters.

- Make sure that the device is connected to the proper serial port. The bulkhead serial ports are labelled "EIA1" (or on 3640s, "SERIAL 1"), "EIA2" ("SERIAL 2"), and "EIA3" ("SERIAL 3"). The console serial port is labelled "RS-232". You must use the port corresponding to the value of the **:unit** keyword to **si:make-serial-stream**. The default value is **2**, so if you do not specify anything, the "EIA2" ("SERIAL 2") connector is the appropriate one.

- If you are using any one of the three bulkhead serial ports (units **1**, **2**, or **3**, you can connect the port directly to a DCE device. However, if the device is a DTE, make sure that there is a null modem between your device and the serial connectors. Since most devices are DTEs, the null modem is probably necessary.

- If you are using the console serial port (unit **0**) you can connect the port directly to a DTE device. However, if the device is a DCE, make sure that there is a null terminal between your device and the serial connectors.

- Try using a different port. Remember both to plug your device into a different connector, and to change the program to specify a different value for the **:unit** keyword.

Notes on Serial I/O

The receiver is implemented using the Symbolics computer's general front end processor (FEP) "channel" facility. When a character arrives at the serial port, the FEP buffers it and transfers it to the Symbolics computer over a "channel".

Therefore, it is not necessary for the program doing input from the stream to read in characters as quickly as they arrive from the external device. The **:clear-input** operation to the serial stream resets this buffer (including the buffers in Symbolics Common Lisp, and the buffers in the FEP). The buffering capacity is about 500 characters. If the buffer is full and another character arrives, an over-run error occurs; if the **:check-overrun-errors** parameter is used, this is reflected by the signalling of an error.

A useful debugging technique is to create a serial stream with the desired parameters and set a variable (say, **s**) to it, and do:

```
(stream-copy-until-eof s standard-output)
```

This prints received characters on the screen until you type c-ABORT. This technique works only with the **:number-of-data-bits** parameter set to **7**, so that the Symbolics computer does not see the ASCII parity bit. Unless character set translation is enabled (via the **:ascii-characters** parameter), ASCII control characters, including carriage return and line feed, are displayed as special symbols, such as circle-cross or delta, because of the differences between the Symbolics character set and ASCII. See the section "The Character Set".

Using the Terminal Program with Hosts Connected to the Serial Line

You can connect a Symbolics machine to another host via the serial line. Specifically, you can use the terminal program to communicate with another host when the Symbolics computer's serial line is connected to a terminal port on the other host.

The network system treats the set of hosts connected to the serial lines of a Symbolics computer as a special network, a *pseudonet*. Before you can use the terminal program to talk to another host over the serial line, you must use the Edit Namespace Object command to create this network object and assign an address on that network to the Symbolics computer. You might want to create or modify the remote host object as well.

1. Create the network. Give it a **name** attribute associated with the Symbolics computer and a **type** attribute of **serial-pseudonet**.

   In the following example, Merrimack is the name of the Symbolics computer:

   ```
   NETWORK MERRIMACK-SERIAL
   TYPE SERIAL-PSEUDONET
   ```

2. Add an entry to the **address** attribute of the Symbolics computer to specify that the Symbolics computer is connected to the new network. Each **address** entry is usually a pair of the form (*network address*). By convention, the Symbolics computer is assigned address 0 on a serial pseudonet. Following is an example of a new **address** entry for the Symbolics computer Merrimack:

   ```
   ADDRESS MERRIMACK-SERIAL 0
   ```

3. If the line rate of the serial line is other than 9600 baud, supply a **peripheral** entry for the Symbolics computer giving the correct baud rate. The peripheral type is **serial-pseudonet**, and the **unit** attribute is the unit number of the serial line. Following is an example of a **peripheral** entry for the Symbolics computer:

```
PERIPHERAL SERIAL-PSEUDONET UNIT 2 BAUD 4800
```

4. You can specify the NUMBER-OF-STOP-BITS, NUMBER-OF-DATA-BITS, and the PARITY for serial-pseudonet peripherals.

5. If you want the terminal program to start out simulating one of the supported terminal types, add a **terminal-type** attribute to the peripheral. Currently supported terminal types are the VT100 and Ann Arbor Ambassador. For example, to make the terminal program simulate an Ambassador, add to the Symbolics computer a **peripheral** entry of this form (note that the entry must actually be on one line):

```
PERIPHERAL SERIAL-PSEUDONET UNIT 2 BAUD 9600
TERMINAL-TYPE Ambassador
```

You can now use the terminal program to connect to the remote host. At the "Connect to host:" prompt, you must supply an address of the form `MERRIMACK-SERIAL|2`. If you want to type a name or nickname of the remote host instead, add **address** and **service** entries for the remote host's namespace object. If the remote host does not exist in the network database, use the Edit Namespace Object command or the function **tv:edit-namespace-object** to create it.

For the **address** entry, specify the serial pseudonet and an address that corresponds to the unit number of the serial line to which the host is connected. The **service** entry is a triple of the form (*service medium protocol*). For the regular host login server, *service* is **login**, *medium* is **serial-pseudonet**, and *protocol* is **tty-login**. Following is an example of **address** and **service** entries for the remote host Blue connected to the Symbolics computer Merrimack:

```
HOST BLUE
SYSTEM-TYPE TENEX
ADDRESS MERRIMACK-SERIAL 2
SERVICE LOGIN SERIAL-PSEUDONET TTY-LOGIN
```

You can also use the serial line to connect to servers other than normal login on a remote host. You must add a **service** entry for the remote host to specify the kind of service, the **serial-pseudonet** medium, and the protocol that the remote host uses. You must also add an **address** entry on the serial pseudonet for the remote host. In the **address** entry, specify the address in the form *protocol=unit* instead of just *unit*. Following are examples of **address** and **service** entries for a file server using protocol **myftp** on remote host Blue:

```
HOST BLUE
SYSTEM-TYPE TENEX
ADDRESS MERRIMACK-SERIAL MYFTP=2
SERVICE FILE SERIAL-PSEUDONET MYFTP
```

For information on the Terminal program: See the section "Using the Terminal Program".

For information on network and host attributes: See the section "Setting Up and Maintaining the Namespace Database".

For information on services, media, and protocols: See the section "Symbolics Generic Network System".

For information on the valid parameters: See the section "Parameters for Serial I/O".

Hardcopy Streams

The functions in this chapter are provided so that you can write an interface between an applications program and a supported printer. That is, they handle the process of getting the thing to be hardcopied to the printer. They assume that you have one of the printers supported by Symbolics; documentation is not provided to write the support for a different type of printer.

Supported Hardcopy Devices

Symbolics currently supports the following hardcopy devices: LGP2, LGP3, ASCII, and DMP1.

The Symbolics LGP3 Laser Graphics Printer is a table-top laser-beam printer based on the Apple LaserWriter, extended with proprietary Symbolics software.

The Symbolics DMP1 Dot-Matrix Printer is a compact, heavy-duty, impact dot-matrix printer with 24-wire print head.

The Hardcopy Stream Model

The interface between Genera and a particular printing device is implemented using a *hardcopy stream*. A hardcopy stream is an output stream. (See the section "Types of Streams".) It handles the usual output operations, such as **:tyo**, **:line-out**, and **:string-out**. In addition it handles operations such as **:set-cursorpos** and **:allocate-margins**. (See the section "Using Hardcopy Streams".) It can handle page breaks and formatting information that you specify when the stream is created.

The various hardcopy menus and commands accept a pathname as an argument. **hardcopy:hardcopy-text-file** or **press:hardcopy-press-file** is then called, depending on the type of file, as determined from the file-type extension in the pathname or as specified by the user. **hardcopy:hardcopy-text-file** and **press:hardcopy-press-file** are the *front end* functions that make sure the appropriate file type and format keywords are included with the file. These functions call

hardcopy:hardcopy-file, which opens the file and then calls the appropriate *formatting* function, **hardcopy:hardcopy-from-stream** or **press:hardcopy-press-stream**, to handle the creation of a hardcopy stream and the actual sending of escape codes to a printer object. A diagram of the hardcopy stream model appears in !.

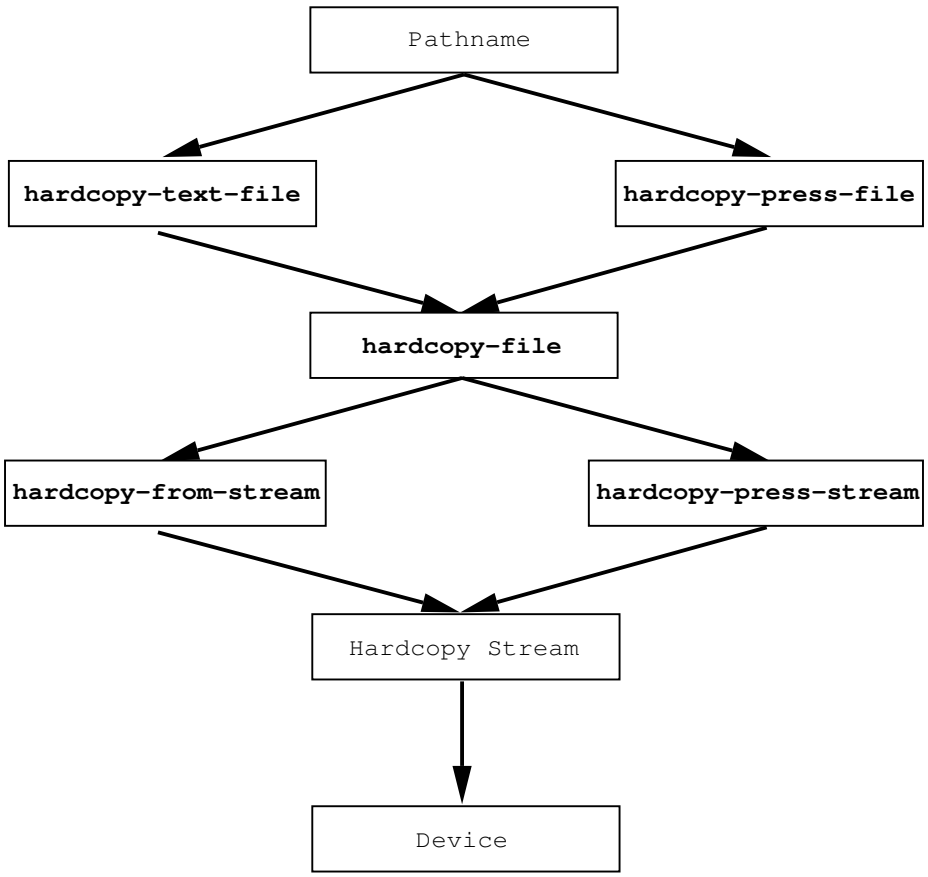


Figure 33. The Hardcopy Stream Model

Making Hardcopy Streams

The functions that create and manipulate hardcopy streams live in the package **hardcopy**, with the exception of those functions that handle press format files, which live in the package **press**.

The basic function to create a hardcopy stream is **hardcopy:make-hardcopy-stream**:

hardcopy:make-hardcopy-stream *device* &rest *options* *Function*

Returns a hardcopy stream to the given device. *options* can be any of the hardcopy option keywords. See the section "Hardcopy Options". **hardcopy:make-hardcopy-stream** creates a stream built on **hardcopy:basic-hardcopy-stream** with characteristics determined by the keyword options you specify. This stream accepts the normal output stream messages, such as **:tyo**, **:string-out**, and some specific hardcopy messages.

For example:

```
(with-open-stream
  (stream (hardcopy:make-hardcopy-stream hardcopy:*default-text-printer*))
  (send stream :string-out "this is a test
of the hardcopy system."))
```

hardcopy:get-hardcopy-device *device* &optional (*error-p* **t**) *Function*

Returns a software object named by *device* that can send data to a hardcopy device. Typical hardcopy devices are printers, files, and windows.

```
(with-open-stream
        (stream (hardcopy:make-hardcopy-stream
                  (hardcopy:get-hardcopy-device device)))
        (send stream :string-out "this is a test."))
```

device can be:

- A string treated as the name of a printer. For example: `"Tattler"`.
- A form that evaluates to a printer. For example: **hardcopy:*default-text-printer***.
- A list of **'(:window** *window***)** representing a particular window to which to send the output. For example:

  ```
  '(:window (make-window tv:window :expose-p t))
  ```

  See the section "Getting a Window to Use".
- A list of **'(:file** *pathname* &optional *canonical-type***)** that sends bytes to *pathname* in the appropriate format. For example, the following creates a file suitable for printing on an LGP2:

  ```
  '(:file "q:>kjones>my-output.text" :lgp2)
  ```
- **:debug**, which means print a description of each message sent to the hardcopy stream to **terminal-io**. For example:

  ```
  (with-open-stream
          (stream (hardcopy:make-hardcopy-stream
                    (hardcopy:get-hardcopy-device :debug)))
          (send stream :string-out "this is a test."))
  ```

produces:

```
Set font 0 (#<FONT CPTFONT 107216574>).
Set cursorpos X=0. micas, Y= 26940 micas.
this is a test.
Eject page (eof).
NIL
```

- **:window**, which means create a special window to accept the output.

Using Hardcopy Streams

hardcopy:basic-hardcopy-stream *Flavor*

The basic flavor upon which all hardcopy streams are built. Any hardcopy stream handles the operations defined by the methods of **hardcopy:basic-hardcopy-stream**. A diagram illustrating an instance of a hardcopy stream appears in!.

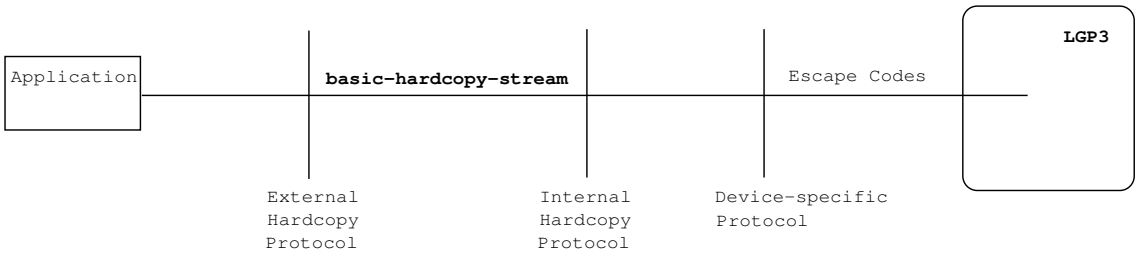


Figure 34. An Instance of a Hardcopy Stream

Messages to hardcopy streams often take a *units* argument. This argument tells the stream how other arguments connoting size and distance should be interpreted. There are several types of units, but by and large they can be grouped into device-specific units and device-independent units. Two common device-specific units are the *pixel* and the *device unit*.

A pixel is the smallest dot the device can image; a typical laser printer like the LGP3 has a resolution of 300 pixels to the inch. A device unit expreses the unit distance of the coordinate system supported by the device; for the LGP3, this is 1/72 of an inch. Device-dependent distances are expressed in device units.

The most common device-independent unit is the *mica*, defined as 10 microns. There are 2540 micas to the inch. Micas are used to express absolute distances in a way that is independent of any particular device.

The operations handled by hardcopy streams are:

(flavor:method :show-rectangle hardcopy:basic-hardcopy-stream) *width height* *Method*

Draws a filled-in rectangle on the page with the lower left corner at the current cursor position of size *width* by *height*. If you are not sure of the current cursor position, use **:set-cursorpos** before **:show-rectangle**. You should not depend on the cursor position after using **:show-rectangle**. If you need the cursor position, do a **:set-cursorpos** after this operation.

width and *height* are always in device-dependent units. Use **:convert-to-device-units** to convert from other units.

(flavor:method :show-line hardcopy:basic-hardcopy-stream) *to-x to-y* *Method*

Draws a line on the page from the current position to the position designated by *to-x*, *to-y*. You should not depend on the cursor position after using **:show-line**. If you need the cursor position, do a **:set-cursorpos** after this operation.

The coordinates given to this message are absolute coordinates. If you have coordinates relative to the page margins, for instance arguments to **:set-cursorpos**, use **:un-relative-coordinates** to convert.

(flavor:method :read-cursorpos hardcopy:basic-hardcopy-stream) &optional (*units* **':device**) *Method*

Returns the current position of the cursor in *units*, either **:micas** or **:device**. The default is **:device**, meaning the units are device dependent.

(flavor:method :set-cursorpos hardcopy:basic-hardcopy-stream) *x y* &optional (*units* **':device**) *Method*

Moves the place where printing occurs on the page to a new position. Unlike the Symbolics console display, the 0,0 point of hardcopy streams is in the lower left corner, the first (*x*) coordinate increasing toward the right of the page, the second (*y*) coordinate increasing toward the top of the page. The coordinates are relative to the margins of the page. If you need absolute coordinates, use **:un-relative-coordinates** to convert.

units specifies the format of *x* and *y*. **:device** means that the interpretation is device dependent. **:micas** means *x* and *y* are in micas.

A value of **nil** for a coordinate means do not set that coordinate. For example,

```
(send stream :set-cursorpos nil 10)
```

sets the cursor position 10 device units above the bottom of the page and leaves its horizontal position unchanged.

(flavor:method :increment-cursorpos hardcopy:basic-hardcopy-stream) *dx dy* &optional *units* *Method*

Changes the point on the page where printing occurs with respect to the current position. Unlike the Symbolics console display, the 0,0 point of hardcopy streams is in the lower left corner, the first (*x*) coordinate increasing toward the right of the page, the second (*y*) coordinate increasing toward the top of the page.

units specifies the format of *x* and *y*. The default is **:device**.

A value of **0** for a coordinate means do not change that coordinate. For example,

```
(send stream :increment-cursorpos 0 10)
```

raises the cursor position 10 device units above where it was and leaves its horizontal position unchanged.

(flavor:method :un-relative-coordinates hardcopy:basic-hardcopy-stream) *x y* &optional (*units* **':device**) *Method*

Converts the point *x,y* given to messages like **:set-cursorpos** that take coordinates relative to the page margins to absolute coordinates for use with messages like **:show-line**.

(flavor:method :convert-to-device-units hardcopy:basic-hardcopy-stream) *quantity units direction* *Method*

Converts *quantity* in *units* into the corresponding quantity in device-dependent units. *units* can be **:micas** or **:pixel**. *direction* is either **:horizontal** or **:vertical**.

(flavor:method :convert-from-device-units hardcopy:basic-hardcopy-stream) *quantity units direction* *Method*

Converts *quantity* from device units into *units*. *direction* is either **:horizontal** or **:vertical**.

(flavor:method :home-cursor hardcopy:basic-hardcopy-stream) *Method*

Positions the cursor at the upper left hand corner of the page.

(flavor:method :size hardcopy:basic-hardcopy-stream) &optional (*units* **':device**) *Method*

Returns width and height, the size of the paper in *units*. *units* can be **:micas**, **:pixel**, or **:device**. **:device** (device-dependent units) is the default.

(flavor:method :inside-size hardcopy:basic-hardcopy-stream) &optional (*units* **':device**) *Method*

Returns the size of the area on the paper (the *box*) within which printing can occur, in *units*. **:device** means the units are device dependent.

(flavor:method :allocate-margin hardcopy:basic-hardcopy-stream) *size margin* &optional (*units* **':device**) *Method*

Adds the amount of space specified by *size* to the margin specified by *margin* in *units*. **:device** means the units are device dependent. Use of **:allocate-margin** increases the margin, making **:inside-size** smaller.

(flavor:method :string-length hardcopy:basic-hardcopy-stream) *string* &optional (*start* **0**) *end* *Method*

Returns the length of *string*, which is the horizontal distance the cursor would have to move to print *string*, in device units.

Example of a hardcopy stream

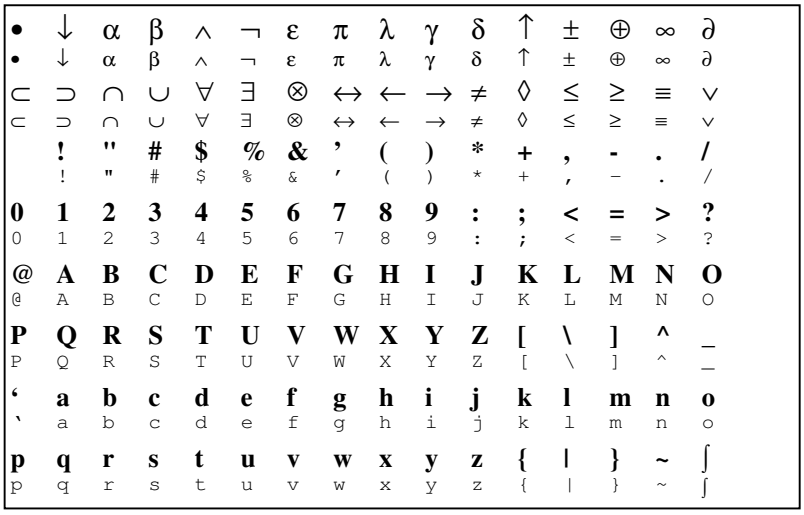
```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: USER; Base: 10 -*-

;;; Print characters in the character set,
;;; alternating roman and some other font.

(defun font-catalog-page (font &optional
                                (printer hardcopy:*default-text-printer*))
  (setq printer (hardcopy:get-hardcopy-device printer))
  (with-open-stream (stream (hardcopy:make-hardcopy-stream
                              printer))
    (let ((fix-font (send stream :maybe-add-font "FIX9"))
          (catalog-font (send stream :maybe-add-font font)))
      (flet ((send-to-stream-in-font (new-font message &rest args)
               (send stream :set-font new-font)
               (lexpr-send stream message args))
             (draw-line (from-x from-y to-x to-y)
               (send stream :set-cursorpos from-x from-y)
               (multiple-value-bind (x y)
                   ;; Note: :SHOW-LINE takes outside coordinates while
                   ;; :SET-CURSORPOS takes inside coordinates.
                   (send stream :un-relative-coordinates to-x to-y)
```

```
        (send stream :show-line x y))))
(multiple-value-bind (x-size y-size) (send stream :inside-size)
  (decf x-size) (decf y-size)           ;Leave room for drawing box
  (let* ((line-height-0 (send stream :convert-to-device-units
                              1 :character :vertical))
         (line-height-both (* 2 line-height-0))
         (x 10)
         (y (- y-size (* 1.3 line-height-both)))
         (max-x x)
         (device-units-rounded?
           ;;If the device units are bigger than 0.01 inch, assume they
           ;;are flonums
           (> (send stream :convert-to-device-units 2540. :micas :vertical)
              100.0)))
    (labels
      ((round-device-units (y)
         (if device-units-rounded? (round y) y))
       (draw-box ()
         (decf y line-height-0)
         (draw-line 0     y      0     y-size)
         (draw-line 0   y      max-x y)
         (draw-line max-x y      max-x y-size)
         (draw-line 0   y-size max-x y-size)
         (send stream :set-cursorpos 0 (- y line-height-both))
         (send-to-stream-in-font
           fix-font
           :string-out (format nil "Font ~A catalog" font)))
       (new-page ()
         (send stream :new-page)
         (setq x 10 y (- y-size line-height-both)
               max-x x))
       (new-line ()
         (setq y (round-device-units
                   (- y (* 1.3 line-height-both))))
         (setq max-x (max x max-x))
         (setq x 10)
         (when (< y line-height-both)
           (draw-box)
```

```
      (new-page)))
  (new-character (character)
    (send stream :set-cursorpos x y)
    (send-to-stream-in-font fix-font :tyo character)
    (send stream :set-cursorpos x (+ y line-height-0))
    (send-to-stream-in-font catalog-font :tyo  character)
    (incf x
      (+ (max (send-to-stream-in-font fix-font
                :character-width character)
              (send-to-stream-in-font catalog-font
                :character-width character))
         10))
    (when (> x (- x-size 10)) (new-line))))
 (setq y (round-device-units y))
 (loop for char from 32 below 127
       and character = (code-char char)
       do
   (new-character character)
       when (= 15 (mod char 16))
         do (new-line))
 (draw-box)))))))
```

```
(zl-user:font-catalog-page "centuryschoolbook105")
```

Output:



The code for this example can be found in SYS:EXAMPLES;HARDCOPY-STREAM-EXAMPLE.LISP.

Hardcopy Streams Reference Information

Hardcopy Front End and Formatting Functions

hardcopy:hardcopy-text-file *file-name device* &rest *options* *Function*

Called by the various hardcopy commands when the file to be hardcopied is just text (as opposed to press format or other format produced by a text formatting program) or is in an unspecified format.

For a discussion of the *options* this command takes, see the section "Hardcopy Options".

hardcopy:hardcopy-text-file calls **hardcopy:hardcopy-file** and **hardcopy:hardcopy-from-stream** to do its work.

press:hardcopy-press-file *filename device* &rest *options* *Function*

Called by the various hardcopy commands when the file to be hardcopied is in press format.

For a discussion of the *options* this command takes, see the section "Hardcopy Options".

hardcopy:hardcopy-file *file-name device* &rest *options* &key *title format formatter file-open-options* &allow-other-keys *Function*

Determines the format of the input file, opens it, and passes the input stream to the appropriate formatter function, **hardcopy:hardcopy-from-stream** or **press:hardcopy-press-stream**.

hardcopy:hardcopy-file takes the following keywords that indicate the *format* of the file:

:ascii Straight ASCII text files, which cannot contain character styles, to be printed on an ASCII printer.

:text Text files, which can contain character styles.

:lgp2, **:lgp3**, **:press** Binary files that are recorded output streams, that is, bits to be sent verbatim to a printer of the proper type.

:postscript Character files containing programs to be interpreted by a PostScript printer.

The variable **hci:*hardcopy-formats*** holds the list of valid hardcopy formats.

For a discussion of the formatting and spooling *options* this command takes, see the section "Hardcopy Options".

hardcopy:hardcopy-from-stream *format stream device* &rest *options* &key (*page-headings* **t**) *starting-page ending-page* &allow-other-keys *Function*

The formatting function for text files. It recognizes character styles in files written by Zmacs. If the file has a `-*- Default Character Style ...` attribute, that style is used as the base for character style merging.

For a discussion of the *options* this command takes, see the section "Hardcopy Options".

press:hardcopy-press-stream *format stream device* &rest *options* &key *starting-page ending-page copies* &allow-other-keys *Function*

The formatting function for press files. It recognizes press format, that is, a description of formatted text, and generates the appropriate escape codes for the printing device specified.

For a discussion of the *options* this command takes, see the section "Hardcopy Options".

Hardcopy Options

The functions **hardcopy:hardcopy-text-file**, **hardcopy:hardcopy-file**, **hardcopy:hardcopy-from-stream**, and **hardcopy:make-hardcopy-stream** that do the actual work of hardcopying files, creating hardcopy streams and sending characters to those streams, share a number of keyword options. Each function handles some keywords and passes the remainder along to the function that it calls. Some of the keyword options determine formatter options, some are handled directly by the hardcopy stream, and others are passed along to the spooler.

Keyword Options for Formatting

Keyword	*Explanation*
:margins	A list of left margin, top margin, right margin, and bottom margin in *micas*.
:page-headings	Whether to put headings on each page. The default is **t**, which puts headings on each page.
:page-heading	The heading to put on the top of each page. The default is the value of **:title**.
:page-heading-date	The date to put in the heading. The default is the value of **:data-creation-date**.
:output-stream	The destination of bytes for the output device. The formatting function creates an output stream by looking at the options it is given.
:keep-output-stream-open-p	Whether to suppress closing the output stream.
:landscape-p	How to orient the output. Landscape means with the long axis of the paper horizontal. Portrait means with the long axis of the paper vertical. **:landscape-p** defaults to **nil**, meaning portrait.

:new-page-hook A function to call at the start of each page. It receives two arguments, the hardcopy stream and the page number. It can be used to print page headings, for example.

:starting-page The first physical page to print. The default is the first page of the file. A page is defined by the presence of a Page character or form feed in the file. Thus plain text files containing no page markers are single-page files. It is important to remember for both **:starting-page** and **:ending-page** that this is a physical page and does not use the page number, if any, supplied by a text formatting program.

:ending-page The last physical page to print. The default is the last page of the file.

:page-number The number to start with when printing numbers on paper. The default is 1. This is a hardcopy stream option. The value is determined by **:starting-page** and **:ending-page**.

Keyword Options for Formatting and Spooling

Keyword	*Explanation*
:body-character-style	The character style to use in printing the main text of the file.
:heading-character-style	The character style to use in printing page headings.
:copies	How many times the request should be printed.
:data-creation-date	The creation date of the data (file or buffer) to print on the cover page and in page headings, in universal time format.
:title	A string describing the data being printed. It appears in printer status messages and on any cover page. The default is "Unnamed Request".

Keyword Options for the Spooler

Keyword	*Explanation*
:print-cover-pages	Whether or not to print a cover page. If it is **t** (the default), a cover page is printed. If it is set to **nil**, then no cover page is printed.
:requestor-user-id	The user name of the requestor (**zl:user-id**).
:requestor-host	The machine from which the request is issued (**net:*localhost***).

:recipient The person for whom the hardcopy output is being printed. It defaults to the personal name of the requestor.

Writing Programs that Use Magnetic Tape

tape:make-stream *Function*

Creates streams that read or write magnetic tape. It handles both cartridge and industry-compatible tape. With **tape:make-stream**, you can access tape on the local machine, or on any machine with a tape server.

tape:make-stream creates a stream. **with-open-stream** and other standard tools for managing streams should be used to ensure proper closing of a stream made with **tape:make-stream**.

Tape streams accept (for output) and return (as input) 8-bit characters. Normal stream messages can be used to tape streams. See the section "Streams".

There are a few other messages: See the section "Messages to Tape Streams".

tape:make-stream takes a large number of optional keyword arguments:

:host The host on which the tape drive to be used is located. This can be a string or a host object. The keyword **:local** is also accepted for the local host. If this argument is not provided, **tape:make-stream** prompts for the name of the host.

The host must already be registered in the network database for supporting TAPE service.

:unit The identifier of the tape drive on the selected host that is to be used. Hosts having only one tape drive generally do not require this information. The value of this argument is generally a character string. "" or **nil** specifies "don't care", which is the usual value.

:reel The name of the tape reel to be mounted. This information is needed by tape servers that have operators, who need to know the name of a tape in order to mount it. It is also needed by servers who have tape access control systems. Currently (Release 5.0) no such servers are supported. "" or **nil**, the usual default, means "don't care".

:direction Specifies whether reading, writing, or intermixed reading and writing are to be performed. The valid values of this argument are thus **:input**, **:output**, and **:bidirectional**, respectively.

:input-stream-mode

This argument, which is only valid if the **:direction** argument is **:input** or **:bidirectional**, controls whether record boundaries, on input, are reflected to you. The default is **t**, meaning that

they are *not*. It is not meaningful for cartridge tapes: record boundaries are never visible to the user of cartridge tape.

In *input stream mode* (a value of **t**), input bytes are transferred from the tape records to you until a file mark (tape mark, EOF) is encountered, at which time you see an end-of-file in your stream.

In *input record mode* (a value of **nil**), input bytes are transferred from the tape records to you until a record boundary, at which time you see an end-of-file in your stream. To progress beyond the record boundary, the message **:discard-current-record** must be sent to the stream.

:record-length Controls the maximum length, in bytes, of tape records. This is ignored for cartridge tape. For reading, it must provide for the largest record to be read. Not all input records need be this long, although in some cases the server decides whether to allow records of other than this size. See also the keywords **:minimum-record-length** and **:minimum-record-length-granularity**. The default is 4096.

:density Density of the tape in bits per inch. This is ignored for cartridge tape. The default is 1600 for servers that have the capability of multiple densities.

:pad-char A number that is the single character with which to pad records when short records are padded. (This is ignored for cartridge tape.) The default pad character is 0. For compatibility with previous releases, supplying this argument and *not* supplying a value for either **:minimum-record-length** or **:minimum-record-length-granularity** implies a value of **:full** for **:minimum-record-length**.

:minimum-record-length

A number that is the minimum record length, in bytes, to which all output records will be padded. (This is ignored for cartridge tape.) This ability is present because many tape controllers cannot read records shorter than some minimum. Arguments to this keyword can be:

not supplied	If this argument is not supplied, a value of 64 is assumed.
integer	Some number smaller than the value of the **:record-length** argument. Short records are padded with 0, or the value of the **:pad-char** argument, if that is supplied.
:full	All records are padded to their maximum length, namely, the value of the **:record-length** argument. Short records are padded

with 0, or the value of the **:pad-char** argument, if that is supplied.

nil — Genera does not enforce any minimum record length. The tape server and/or the tape hardware on that server might enforce some minimum of its own.

:minimum-record-length-granularity

An integer, or **nil**, establishing a *granularity*, or enforced integral divisor, for the length of all tape records written. If non-**nil**, all records written are padded (with 0, or the value of the **:pad-char** argument, if that is supplied) to be multiples of this number in length. This value is ignored for cartridge tape. It is also ignored if short records are not to be written, that is, **:minimum-record-length** is given as **:full** or the same as **:record-length**.

All Genera tape applications (LMFS and distribution dumpers and carry tape) enforce a granularity of 4.

:prompt — This is an optional string that is formatted into **tape:make-stream**'s prompt for a host name, if one is issued. It should describe the tape to be mounted in terms of the application program running. For instance, if this string is supplied as **"billing master"**, **tape:make-stream** might prompt

```
Type name of tape host for billing master:
```

:no-bot-prompt — Normally, **tape:make-stream** notices if the tape is offline, or not at BOT (beginning-of-tape) when it is called. If the tape is offline, **tape:make-stream** queries you to wait for it to become ready. If the tape is not at BOT, **tape:make-stream** queries you about rewinding it. Supplying a non-**nil** value for **:no-bot-prompt** suppresses these checks, allowing you to handle these exigencies in any way you choose. The message **:bot-p** can be sent to a tape stream to determine if it is at BOT, and **:check-ready** to wait for a tape to become ready.

:norewind — Normally, **tape:make-stream** rewinds the tape at the time the stream is closed. Supplying a non-**nil** value for **:norewind** suppresses this behavior.

:lock-reason — Another optional string describing the application. This string is used in error messages sent to other users who try to access the tape drive you are using. For instance, if it is supplied as **"daily billing run"**, another user might see a message like:

```
Cannot mount tape:
Drive 0 in use by daily billing run.
```

Messages to Tape Streams

The following messages to tape streams are important. Tape streams, of course, also support standard stream messages appropriate to input or output streams. See the section "Streams".

These are the messages relevant to any kind of tape stream:

:close (&optional (*abort-p* **nil**))
Closes the stream. Normally, causes a rewind, and all the operations associated with **:rewind** (see the description of **:rewind**) to take place. The **:norewind** argument suppresses this rewind, although, for an output stream, buffered output is written, along with two EOFs. The tape is left positioned between the two EOFs, for industry-compatible tape, or after them, for cartridge tape.

:rewind Rewinds the tape. For input streams, buffered input is discarded before the rewind. For output streams, buffered output is written out, possibly padded, according to the current padding parameters, and then two EOFs written, before the rewind. No read-ahead is performed. This message does *not* wait for the rewind to complete.

:await-rewind Waits for a previously started rewind to complete.

:set-offline A **:rewind** is done, and the tape is set offline, or unloaded, as befits the controller and drive. The setting of the tape offline does not wait for the rewind to complete.

:clear-error If a tape error occurs, and is handled by you, you must send this message before attempting to continue using the stream. Otherwise, it remains in the error state, where it can only be closed.

:skip-file (&optional (*n* **1**))
Skips to, and past, a file mark (EOF). *n* is how many to skip, and can be negative, indicating backward motion. For input streams, all buffered input is discarded before the motion. For output streams, this operation is not valid unless the last thing written was an EOF, not a data record. Cartridge tape cannot skip backward. Forward motion is not allowed immediately after output.

:host-name The name of the host on which the tape is mounted.

:bot-p Returns **t** if the tape is at BOT (beginning of tape), and **nil** if not.

:check-ready Checks to make sure the tape drive is ready, and informs you, waiting interactively, if not.

These are the messages specifically relevant to tape input streams. Most of them are relevant only to input record mode, which is the mode requested by a value of **nil** for **:input-stream-mode**. See the description of the **:input-stream-mode** argument to the function **tape:make-stream**.

:clear-eof This clears the EOF state that results from reading an EOF mark. When an EOF is encountered, all character-reading operations encounter an end-of-file indication until **:clear-eof** is sent. This is needed in input stream mode as well as input record mode.

:discard-current-record
This discards the remainder of the current record, when in input record mode, and allows reading the next record. This message must be issued to progress past a record boundary in input record mode, even if all of the bytes in the record have been read. This is meaningless for cartridge tape.

:record-status (&optional (*error-p* **t**))
This is only valid in input record mode, and meaningless for cartridge tape. This call is only valid at the beginning of a record, that is, if no bytes have been read from the current record. It describes, via its return value, the record that is about to be read by the user. Here are the possible values:

an error object	The next record cannot be read, due to error. An error object is returned. If *error-p* is **t**, which is the default, an error is signalled in this case, instead of an error object being returned.
integer	The length of a good record, in bytes.
:eof	The next record is not a record at all, but an EOF (a file mark).

These are the messages relevant to tape output streams:

:write-eof Writes an EOF (a file mark). If a record is being built, it is written out. Whether or not it is padded depends upon the values of the arguments **:minimum-record-length** and **:minimum-record-length-granularity**.

:force-output Writes out any record being buffered. Whether or not it is padded depends upon the values of the arguments **:minimum-record-length** and **:minimum-record-length-granularity**. This is the normal way to end a record when record boundaries are significant, or short records are written. Otherwise, records are written when they are full.

:write-error-status (&optional *error-p*)
Verifies that all records have been written correctly. Tape

streams often buffer many records ahead. **:write-error-status** waits for all buffered I/O to complete. If there was no error, **nil** is returned. If there was an error, an error object is returned describing the error. If *error-p* is non-**nil**, an error is signalled instead. If the error is end of tape, however, and **error-p** is **nil**, **:end-of-tape** is returned.

Tape Error Flavors

tape:tape-error *Flavor*

This set includes all tape errors. This flavor is built on **error**.

tape:mount-error *Flavor*

A set of errors signalled because a tape could not be mounted. This includes problems such as "no ring" and "drive not ready". Normally, **tape:make-stream** handles these errors, and manages mount retry. This flavor is built on **tape:tape-error**.

tape:tape-device-error *Flavor*

A hardware data error, such as a parity error, controller error, or interface error, occurred. This flavor has **tape:tape-error** as a **:required-flavor**.

tape:end-of-tape *Flavor*

The end of the tape was encountered. When this happens on writing, the tape usually has a few more feet left, in which the program is expected to finish up and write two end-of-file marks. Normally, closing the stream does this automatically. Whether or not this error is ever seen on input depends on the tape controller. Most systems do not see the end of tape on reading, and rely on the software that wrote the tape to have cleanly terminated its data, with EOFs.

This flavor is built on **tape:tape-device-error** and **tape:tape-error**.

Stack Groups and Subprimitives

Subprimitives for 3600-family and Ivory Based Machines

Subprimitives are functions that are not intended to be used by the average program, only by "system programs". They allow you to manipulate the environment at a level lower than normal Lisp. Subprimitives usually have names that start with a % character. The "primitives" described elsewhere typically use subprimitives to accomplish their work. The subprimitives take the place of machine language in other systems, to some extent. In most cases, subprimitive operations have been hand-coded in microcode by Symbolics.

Subprimitives, by their very nature, cannot do full checking. Improper use of subprimitives can destroy the environment. Subprimitives come in varying degrees of dangerousness. Those without a % sign in their name cannot destroy the environment, but are dependent on "internal" details of the Lisp implementation. The ones whose names start with a % sign can violate system conventions if used improperly. Note that this chapter does not document all the things you need to know in order to use them. Still other subprimitives are not documented here because they are very specialized. Most of these are never used explicitly by a programmer; the compiler inserts them into the program to perform operations that are expressed differently in the source code.

The most common problem you can cause using subprimitives, though by no means the only one, is to create invalid pointers: pointers that, because of one storage convention or another, are not allowed to exist. The storage conventions are not documented; as we said, you have to be an expert to correctly use a lot of the functions in this chapter. If you create such an invalid pointer, it probably will not be detected immediately, but later on parts of the system might see it, notice that it is invalid, and (probably) halt the machine.

In a certain sense **car**, **cdr**, **rplaca**, and **rplacd** are subprimitives. If these are given a locative instead of a list, they access or modify the cell addressed by the locative without regard to what object the cell is inside. Subprimitives can be used to create locatives to strange places.

Many subprimitives that are used only for effect also return values. A few look like functions but are really macros; they do not evaluate their arguments in left-to-right order.

Names of subprimitives are currently in a variety of packages, but all of them are exported by the **system** package. The best way to reference a subprimitive is to use a **system:** prefix, which can be abbreviated **sys:**. You can also make your own package *use* the **system** package.

Additional information can be found in the Symbolics Supplemental Sources package.

SYS: L-SYS; SYSDEF.LISP Data structure definitions

SYS: L-SYS; SYSDF1.LISP Communication areas, escape routines

SYS: L-SYS; OPDEF.LISP Instruction set definition

For Ivory based machines, the corresponding files are:

SYS: I-SYS; SYSDEF.LISP Data structure definitions

SYS: I-SYS; SYSDF1.LISP Communication areas, escape routines

SYS: I-SYS; OPDEF.LISP Instruction set definition

Data Type Subprimitives

sys:data-type *x* *Function*

Returns a symbol that is the name for the internal data type of the "pointer" that represents *x*. Note that some types as seen by the user are not distinguished from each other at this level, and some user types can be represented by more than one internal type. For example, on 3600-family machines, **sys:dtp-extended-number** is the symbol that **sys:data-type** would return for a double-precision floating-point number, a bignum, a complex number, or a rational number even though those types are quite different. The **type-of** function is a higher-level primitive that is more useful in most cases; normal programs should always use **type-of** rather than **sys:data-type**.

Some of these type codes are internal tag fields that are never used in pointers that represent Lisp objects at all, but they are listed here anyway.

sys:dtp-symbol The object is a symbol.

sys:dtp-nil **nil** has a data type of **dtp-nil**, rather than **sys:dtp-symbol**, and does not have a pointer field of zero. **symbolp** of **nil** is true, and the address field points to the same storage representation as all other symbols.

sys:dtp-fix The object is a fixnum; the numeric value is contained in the address field of the pointer.

sys:dtp-float The object is a single-precision floating-point number.

sys:dtp-extended-number The object is a double-precision floating-point, rational, or complex number, or a bignum. This value will also be used for future numeric types.

sys:dtp-list The object is a cons.

sys:dtp-locative The object is a locative pointer.

sys:dtp-array The object is an array.

sys:dtp-compiled-function The object is a compiled function.

sys:dtp-closure The object is a dynamic closure. See the section "Dynamic Closures".

sys:dtp-lexical-closure The object is a lexical closure. See the section "Lexical Scoping".

sys:dtp-instance The object is an instance of a flavor, that is, an "active object". See the section "Flavors".

sys:dtp-generic-function The object is a generic function. See the section "Generic Functions".

sys:dtp-character The object is a character. See the section "Characters".

sys:dtp-null Nothing to do with **nil**. This is used in unbound value and function cells.

sys:dtp-external-value-cell-pointer
An invisible pointer used for external value cells, which are part of the closure mechanism. See the section "Dynamic Closures".

sys:dtp-header-forward An invisible pointer used to indicate that the structure containing it has been moved elsewhere. The "header word" of the structure is replaced by one of these invisible pointers.

sys:dtp-element-forward An invisible pointer used to indicate that the structure containing it has been moved elsewhere. This points to the new location of the word containing it.

sys:dtp-one-q-forward An invisible pointer used to indicate that the single cell containing it has been moved elsewhere.

sys:dtp-logic-variable An invisible pointer used by Symbolics Prolog.

sys:dtp-monitor-forward An invisible pointer used by the debugging facilities such as the Command Processor command Monitor Variable. See the section "Debugger".

sys:dtp-gc-forward This is used by the garbage collector to flag the obsolete copy of an object; it points to the new copy.

sys:dtp-odd-pc, sys:dtp-even-pc
The object is a program counter and points to macroinstructions.

sys:dtp-header-i, sys:dtp-header-p
Internal markers in storage, found at the base of the storage of structures.

sys:*data-types* *Variable*

A list of all of the symbolic names for data types described above under **sys:data-type**. These are the symbols whose print names begin with **dtp-**. The values of these symbols are the internal numeric data-type codes for the various types.

si:data-types *type-code* *Function*

Given the internal numeric data-type code, returns the corresponding symbolic name.

sys:%instance-flavor

instance *Function*

Gets the flavor structure of *instance*.

sys:%change-list-to-cons

list *Function*

Changes the two-element cdr-coded *list* to a dotted pair by altering the cdr codes.

sys:%flonum

number *Function*

Sets the data type field to convert a fixnum to a flonum. It is not the function **zl:float**, but instead provides direct access to the internal bit representation of single-precision floating-point numbers.

sys:%fixnum

number *Function*

Sets the data type field to convert a flonum to a fixnum. It is not the function **zl:fix**, but instead provides direct access to the internal bit representation of single-precision floating-point numbers.

Forwarding Words in Memory

An *invisible pointer* is a kind of pointer that does not represent a Lisp object, but just resides in memory. There are several kinds of invisible pointers and various rules about where they can appear. The basic property of an invisible pointer is that if the machine reads a word of memory and finds an invisible pointer there, instead of seeing the invisible pointer as the result of the read, it does a second read, at the location addressed by the invisible pointer, and returns that as the result instead. Writing behaves in a similar fashion. When the machine writes a word of memory it first checks to see if that word contains an invisible pointer; if so it goes to the location pointed to by the invisible pointer and tries to write there instead. Many subprimitives that read and write memory do not do this checking.

The simplest kind of invisible pointer has the data type code **sys:dtp-one-q-forward**. It is used to forward a single word of memory to another location. The invisible pointers with data types **sys:dtp-header-forward** and **sys:dtp-element-forward** are used for moving whole Lisp objects (such as cons cells or arrays) another location. The **sys:dtp-external-value-cell-pointer** is very similar to the **sys:dtp-one-q-forward**; the difference is that it is not "invisible" to the operation of binding. If the (internal) value cell of a symbol contains a **sys:dtp-external-value-cell-pointer** that points to some other word (the external value cell), then **symbol-value** or **set** operations on the symbol consider the pointer to be invisible and use the external value cell, but binding the symbol saves away the **sys:dtp-external-value-cell-pointer** itself, and stores the new value into the internal value cell of the symbol. This is how dynamic closures are implemented.

sys:dtp-gc-forward is not an invisible pointer at all; it only appears in old space and is never seen by any program other than the garbage collector. When an object is found not to be garbage, and the garbage collector moves it from old space to copy space, a **sys:dtp-gc-forward** is left behind to point to the new copy of the object. This ensures that other references to the same object get the same new copy.

structure-forward *old new* &optional *(old-header-size* **1***) (new-header-size* **1***)*
Function

Causes references to *old* to reference *new*, by storing invisible pointers in *old*. It returns *old*.

An example of the use of **structure-forward** is **zl:adjust-array-size**. If the array is being made bigger and cannot be expanded in place, a new array is allocated, the contents are copied, and the old array is structure-forwarded to the new one. This forwarding ensures that pointers to the old array, or to cells within it, continue to work. When the garbage collector goes to copy the old array, it notices the forwarding and uses the new array as the copy; thus the overhead of forwarding disappears eventually if garbage collection is in use.

follow-structure-forwarding *structure* *Function*

Normally returns *object*, but if *object* has been **structure-forward**ed, returns the object at the end of the chain of forwardings. If *object* is not exactly an object, but a locative to a cell in the middle of an object, a locative to the corresponding cell in the latest copy of the object is returned.

forward-value-cell

from-symbol to-symbol *Function*

Alters *from-symbol* so that it has the same value as *to-symbol*, by sharing its value cell. A **sys:dtp-one-q-forward** invisible pointer is stored into *from-symbol*'s value cell. **forward-value-cell** is careful to never move a cell that is already forwarded.

To forward one arbitrary cell to another (rather than specifically one value cell to another), given two locatives, do:

```
(sys:%p-store-tag-and-pointer locative1
        sys:dtp-one-q-forward locative2)
```

follow-cell-forwarding *loc evcp-p* *Function*

Normally returns *loc*, a locative to a cell, but if the cell has been forwarded, this follows the chain of forwardings and returns a locative to the final cell. If the cell is part of a structure that has been forwarded, the chain of structure forwardings is followed, too. If *evcp-p* is **t**, external value cell pointers are followed; if it is **nil** they are not.

Pointer Manipulation

It should be emphasized that improper use of these functions can damage or destroy the Lisp environment. It is possible to create pointers with illegal data type, to create pointers to nonexistent objects, and to completely confuse the garbage collector.

sys:%pointerp *object* *Function*

Returns **t** when *object* has an address (as opposed to being an immediate object).

sys:%pointer-type-p *data-type-number* *Function*

Returns **t** if the argument is a data type code that has an associated address (rather than an associated immediate field). The argument comes from **sys:%data-type** or **sys:%p-data-type.**

For example:

```
(sys:%pointer-type-p (sys:%data-type 'symbol))
```

sys:%pointer-lessp *p1 p2* *Function*

Compares two addresses. Returns **t** if *p1* has a pointer field lower in the address space than *p2*'s pointer field; returns **nil** otherwise.

sys:%data-type *x* *Function*

Returns the data-type field of *x*, as a fixnum.

sys:%pointer *x* *Function*

Returns the pointer field of *x*, as a fixnum. For most types, this is dangerous since the garbage collector can copy the object and change its address.

sys:%make-pointer *data-type pointer* *Function*

Makes up a pointer, with *data-type* in the data-type field and the pointer field of *pointer* in the pointer field, and returns it.

data-type should be an internal numeric data-type code; these are the values of the symbols that start with **dtp-**.

pointer can be any object; its pointer field is used. This is most commonly used for changing the type of a pointer. Do not use this to make pointers that are not allowed to be in the machine, such as **sys:dtp-null** or invisible pointers.

sys:%make-pointer-offset *new-dtp pointer offset* *Function*

Returns a pointer with *new-dtp* in the data-type field, and *pointer* plus *offset* in the pointer field. The *new-dtp* and *pointer* arguments are like those of **sys:%make-pointer**; *offset* can be any object but is usually a fixnum. The types of the arguments are not checked; their pointer fields are simply added together. This is useful for constructing locative pointers into the middle of an object, although **sys:%p-structure-offset** can be more appropriate.

sys:%pointer-difference *pointer-1 pointer-2* *Function*

Returns a fixnum that is *pointer-1*'s pointer field minus *pointer-2*'s pointer field. No type checks are made. For the result to be meaningful, the two pointers must point into the same object, so that their difference cannot change as a result of garbage collection.

Analyzing Structures

sys:%find-structure-header *pointer* *Function*

Finds the structure into which *pointer* points, by searching backward for a header. It is a basic low-level function used by such things as the garbage collector. *pointer* is normally a locative, but its data-type is ignored.

In structure space, the "containing structure" of a pointer is well-defined by system storage conventions. In list space, it is considered to be the contiguous, cdr-coded segment of list surrounding the location pointed to. If a cons of the list has been copied out by **rplacd**, the contiguous list includes that pair and ends at that point.

sys:%find-structure-leader *pointer* *Function*

Always returns the lowest address in the structure (as a locative).

sys:%structure-total-size *pointer* *Function*

Returns the total number of words occupied by the representation of the indicated object.

sys:%find-structure-extent *pointer* *Function*

Roughly a combination of **sys:%find-structure-header**, **sys:%find-structure-leader**, and **sys:%structure-total-size**. It returns three values:

1. The structure into which *pointer* points.

2. A locative to the base of the structure. This is almost the same as **sys:%find-structure-leader**, but **sys:%find-structure-extent** always returns a locative.

3. The total number of words occupied by the object (the same thing **sys:%structure-total-size** returns).

Example:

```
(defun page-in-structure (obj &optional
                          (hang-p *default-page-in-hang-p*)
                          (normalize-p *default-page-in-normalize-p*))
  (setq obj (follow-structure-forwarding obj))
  (multiple-value-bind (nil leader size)
      (sys:%find-structure-extent obj)
    (page-in-words leader size
                   hang-p normalize-p)))
```

Basic Locking Subprimitive

store-conditional *pointer old new* *Function*

Takes three arguments: *pointer* (a locative which addresses some cell), *old* (any Lisp object), and *new* (any Lisp object). It checks to see whether the cell contains *old*, and, if so, it stores *new* into the cell. The test and the set are done as a single atomic operation. **store-conditional** returns **t** if the test succeeded and **nil** if the test failed. It behaves like **sys:%p-store-contents** in that it leaves the cdr code of the location that is being stored into undisturbed. You can use **store-conditional** to do arbitrary atomic operations to variables that are shared between processes. For example, to atomically add 3 into a variable *x*:

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

The first argument is a locative so that you can atomically affect any cell in memory; for example, you could atomically add 3 to an element of an array or structure.

store-conditional locks out microtasks but cannot lock out the FEP or external-DMA devices. Protocols for communicating with such devices must use locking methods that do not depend on atomic read-modify-write, such as those based on cells that are only written by one party and only read by the other party.

The old name for this function, **sys:%store-conditional**, is still accepted, but should not be used in new programs.

Storage Layout Definitions

The following special variables have values that define the most important attributes of the way Lisp data structures are laid out in storage. In addition to the variables documented here, there are many others that are more specialized. They are not documented here since they are in the **system** package rather than the **global** package. The variables whose names start with %% are byte specifiers, in-

tended to be used with subprimitives such as **sys:%p-ldb**. If you change the value of any of these variables, you will probably bring the machine to a crashing halt.

The byte specifiers **sys:%%q-fixnum** and **sys:%%q-high-type** reflect the fact that the number of bits in a fixnum does not equal the number of bits in a pointer.

For details about byte specifiers, field values, and accessor macros for the internal data structures, see the files SYS:L-SYS;SYSDEF.LISP (for 3600-family machines) or SYS:I-SYS;SYSDEF.LISP (for Ivory-based machines). This file is part of the Supplemental Source package available from Symbolics.

sys:%%q-cdr-code *Variable*

The field of a memory word that contains the cdr-code. See the section "Cdr-Coding".

sys:%%q-data-type *Variable*

The field of a memory word that contains the data type code. See the section "Data Types".

sys:%%q-pointer *Variable*

The field of a memory that contains the pointer address, or immediate data.

sys:%%q-pointer-within-page *Variable*

The field of a memory word that contains the part of the address that lies within a single page.

sys:%%q-typed-pointer *Variable*

The concatenation of the **sys:%%q-data-type** and **sys:%%q-pointer** fields.

sys:%%q-all-but-typed-pointer *Variable*

The field of a memory word that contains the tag field **sys:%%q-cdr-code**.

sys:%%q-all-but-pointer *Variable*

The concatenation of all fields of a memory word except for **sys:%%q-pointer**.

sys:%%q-all-but-cdr-code *Variable*

The concatenation of all fields of a memory word except for **sys:%%q-cdr-code**.

sys:cdr-normal *Variable*

One of the numeric values that go in the cdr-code field of a memory word. This value means that the cdr is stored in the next location. See the section "Cdr-Coding".

sys:cdr-next *Variable*

One of the numeric values that go in the cdr-code field of a memory word. This value means that the cdr is the next location. See the section "Cdr-Coding".

sys:cdr-nil *Variable*

One of the numeric values that go in the cdr-code field of a memory word. The cdr is **nil**. See the section "Cdr-Coding".

Special Memory Referencing

These subprimitives reference and manipulate memory contents. There is no subprimitive called **%p-contents**, since the function **location-contents** performs that operation.

sys:%p-structure-offset *x offset* *Function*

Does **follow-structure-forwarding** on *x*, then **sys:%make-pointer-offset sys:dtp-locative** of that and *offset*. This operation captures the inherent primitive underlying **sys:%p-ldb-offset** and the like.

sys:%p-contents-offset *pointer offset* *Function*

Checks the cell pointed to by *pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting forwarded *pointer* and returns the contents of that location.

sys:%p-contents-as-locative *x* *Function*

Returns the contents of the location as a **sys:dtp-locative**, given a pointer to a memory location containing a pointer that is not allowed to be "in the machine" (typically an invisible pointer). It changes the disallowed data type to **sys:dtp-locative** so that you can safely look at it and see what it points to. You must be sure that the location really contains a pointer data type.

sys:%p-contents-as-locative-offset *pointer offset* *Function*

Checks the cell pointed to by *pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting

forwarded *pointer*, fetches the contents of that location, and returns it with the data type changed to **sys:dtp-locative** in case it was a type that is not allowed to be "in the machine" (typically an invisible pointer).

sys:%p-store-contents *pointer x* *Function*

x is stored into the data-type and pointer fields of the location addressed by *pointer*. The cdr-code field remains unchanged. *x* is returned.

sys:%p-store-contents-offset *value pointer offset* *Function*

Checks the cell pointed to by *pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting forwarded *pointer*, and stores *value* into the data-type and pointer fields of that location. The cdr-code field remains unchanged. *value* is returned.

sys:%p-store-type-and-pointer *pointer type-field pointer-field* *Function*

The location addressed by pointer is written, without following invisible pointers, such that the type field of the location contains *type-field* and the pointer field contains *pointer-field*. The cdr-field of *pointer* is preserved. This is a good way to store a forwarding pointer from one cell to another.

sys:%p-store-tag-and-pointer *pointer tag-fields pointer-field* *Function*

The location addressed by *pointer* is written, without following invisible pointers, such that the tag fields of the location contain *tag-fields* and the pointer field contains *pointer-field*. **sys:%p-store-tag-and-pointer** will overwrite the cdr-code field of *pointer* with that in *type-fields*, and may not be suitable for storing a forwarding pointer from one cell to another. To preserve the cdr-code, see the function **sys:%p-store-type-and-pointer**. Also, see the function **sys:%p-store-cdr-type-and-pointer**.

sys:%p-store-cdr-and-contents *pointer x cdr* *Function*

Stores *cdr* and the object *x* into a memory location identified by *pointer*, without reading the previous contents of that location or following invisible pointers. Use this subprimitive to store fixnums and single-precision floating-point numbers; **sys:%p-store-tag-and-pointer** cannot do so, because the tag overlaps the value.

This function can be used to write to hardware registers in Symbolics machines.

sys:%p-store-cdr-type-and-pointer *pointer cdr-field type-field pointer-field* *Function*

A more general form of **sys:%p-store-tag-and-pointer**.

sys:%p-ldb *bytespec pointer* *Function*

A read operation like **ldb** ("load byte"), but it fetches a byte specified by *bytespec* from the location addressed by *pointer*. Note that you can load bytes out of the data type, not just the pointer field, and that the source word need not be a fixnum. It always returns a fixnum.

The size of *bytespec* must be 32 or less, and the sum of the size and position must be less than or equal to 36 on 3600-family machines, 40 on Ivory based-machines.

This function can be used for reading hardware registers.

sys:%p-ldb-offset *ppss pointer offset* *Function*

Checks the cell pointed to by *pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, the byte specified by *ppss* is loaded from the contents of the location addressed by the forwarded *pointer* plus *offset*, and returned as a fixnum.

The size of *ppss* must be 32 or less, and the sum of the size and position must be less than or equal to 36 on 3600-family machines, 40 on Ivory based-machines.

sys:%p-dpb *newbyte bytespec pointer* *Function*

value, a fixnum, is stored into the byte selected by *bytespec* in the word addressed by *pointer*. **nil** is returned. You can use this to alter data types, cdr codes, and so on.

The size of *bytespec* must be 32 or less, and the sum of the size and position must be less than or equal to 36 on 3600-family machines, 40 on Ivory-based machines.

sys:%p-dpb-offset *value ppss pointer offset* *Function*

Checks the cell pointed to by *pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, *value* is stored into the byte specified by *ppss* in the location addressed by the forwarded *pointer* plus *offset*. **nil** is returned. The size of *ppss* must be 32 or less, and the sum of the size and position must be less than or equal to 36 on 3600-family machines, 40 on Ivory based-machines.

sys:%p-pointer *pointer* *Function*

Extracts the pointer field of the contents of the location addressed by *pointer* and returns it as a fixnum.

sys:%p-data-type *pointer* *Function*

Extracts the data-type field of the contents of the location addressed by *pointer* and returns it as a fixnum.

sys:%p-cdr-code *pointer* *Function*

Extracts the cdr-code field of the contents of the location addressed by *pointer* and returns it as a fixnum.

sys:%p-store-pointer *pointer value* *Function*

Replaces the pointer field of the location addressed by *pointer* with *value*, and returns *value*.

sys:%p-store-data-type *pointer value* *Function*

Replaces the data-type field of the location addressed by *pointer* with *value*, and returns *value*.

sys:%p-store-cdr-code *pointer value* *Function*

Replaces the cdr-code field of the location addressed by *pointer* with *value*, and returns *value*.

sys:%stack-frame-pointer *Function*

Returns a locative pointer to its caller's stack frame. This function is not defined in the interpreted Lisp environment; it works only in compiled code.

sys:%block-store-cdr-and-contents *address count cdr contents increment* *Function*

The contiguous region of memory specified by the beginning *address* and *count* of words is efficiently filled with the object *contents* and the cdr-code (*cdr*). The addresses to be initialized must not be mapped to A memory. The *increment* to *contents* is typically 0. The increment is added to the address field (**sys:%%q-pointer**) of *contents*. If *increment* is nonzero, it must not be used to increment a pointer across the boundaries of a garbage collector "space"; otherwise, the garbage collector tags will be set incorrectly.

sys:%block-store-tag-and-pointer *address count tag pointer increment* *Function*

The contiguous region of memory specified by the beginning *address* and *count* of words is efficiently filled with a word assembled from the *tag* and *pointer* fields, allowing the construction of invisible pointers. The addresses to be initialized must not be mapped to A memory. The *increment* to *contents* is usually 0. If *increment* is nonzero, it must not be used to increment a pointer across the boundaries of a garbage collector "space"; otherwise, the garbage collector tags will be set incorrectly.

sys:%block-search-eq *object address count* *Function*

The contiguous region of memory specified by the beginning *address* and *count* of words is searched for the specified *object*. The comparison uses the *eq* function. If it does not find anything it returns **nil**; otherwise, it returns the address of the word it found.

sys:%unsynchronized-device-read *address* *Function*

Reads registers from the revision 2 I/O board on 3600-family machines. It allows data that are not properly synchronized to the Lbus clock to be read without causing a parity error.

Special Variable Binding Subprimitive

sys:%bind-location *pointer value locative value* *Function*

Binds the cell pointed to by *locative* to *value*, in the caller's environment. This function is not defined in the interpreted Lisp environment; it works only from compiled code. Since it turns into an instruction, the "caller's environment" really means "the binding block for the stack frame that executed the **sys:%bind-location** instruction". The preferred higher-level primitives that turn into this are **let-if**, **zl:progv**, **progw**, and **letf**.

Function-Calling Subprimitives

Except for **sys:%push** and **sys:%pop**, the subprimitives for calling with a run-time-variable number of arguments, without consing a list, are the **sys:%start-function-call** and **sys:%finish-function-call** special forms.

sys:%start-function-call *function destination n-arguments lexpr* *Function*

Calls a function with a variable number of arguments at run time, without consing a list. See the section "Function-Calling Subprimitives".

sys:%finish-function-call *function destination n-arguments lexpr* *Function*

Finishes a call to a function with a variable number of arguments at run time, without consing a list. See the section "Function-Calling Subprimitives".

sys:%start-function-call and **sys:%finish-function-call** each take the same four subforms. The subforms are:

function A form evaluated to yield the function to be called.

destination The disposition of its results. Not evaluated. It takes these values:

Value	*Meaning*
nil	Call for effect.
t	Receive one value on the stack. You must use **sys:%pop** to fetch the value off the stack. You should not use the value of the "call" to **sys:%finish-function-call**.
return	Return all values from the function in which it is being used.

There is no provision for receiving multiple values.

n-arguments A form evaluated to yield the number of times **sys:%push** has to be done.

lexpr True if the last **sys:%push** is a list of arguments rather than a single argument; false in the normal case. Not evaluated.

Follow these steps:

1. Do a **sys:%start-function-call**.
2. Do a **sys:%push** on each argument.
3. Do a **sys:%finish-function-call**.

The order of evaluation of the subforms is not guaranteed, and you must make certain to pass the same subform values to the **sys:%start-function-call** and the **sys:%finish-function-call**. Generally it is best to use variables and not do computations in these subforms.

Also, you must not allocate or deallocate any local variables between the **sys:%start-function-call** and the **sys:%finish-function-call**, because they will get in the way of the **sys:%push** subprimitives. Thus, the following *will not work*:

```
(sys:%start-function-call ...)
(dolist (x l) (sys:%push x))
(sys:%finish-function-call ...)
```

Instead, write:

```
(let ((x l))
   (sys:%start-function-call ...)
   (do () ((null x)) (sys:%push (pop x)))
   (sys:%finish-function-call ...))
```

Once you have done **sys:%start-function-call**, you cannot return from the function until after you have done a **sys:%finish-function-call**; the return instructions do not operate correctly until the **sys:%finish-function-call**.

sys:%push *value* *Function*

Pushes *value* onto the stack. Use this to push the arguments. See the section "Function-Calling Subprimitives".

sys:%pop *Function*

Pops the top value off of the stack and returns it as its value. See the section "Function-Calling Subprimitives".

The Paging System

Note that it is futile to page-in sections of virtual memory that are larger than physical memory. Be especially wary of **sys:page-in-area** and **sys:page-in-region**.

For a table of related functions and methods for raster operations:See the section "Operations on Rasters".

sys:page-in-structure *structure* &rest *page-in-words-keywords obj* &rest *page-in-words-keywords* *Function*

Makes sure that the storage that represents *obj* is in main memory. Any pages that have been swapped out to disk are read in, using as few disk operations as possible. Consecutive disk pages are transferred together, taking advantage of the full speed of the disk. If *obj* is large, this is much faster than bringing the pages in one at a time on demand. The storage occupied by *obj* is defined by the **sys:%find-structure-extent** subprimitive.

These are the keywords accepted by **sys:page-in-words**:

If *hang-p* is **t**, the function waits for the disk reads to finish before returning. Otherwise, the function returns immediately after requesting the disk reads, which might still be in progress. Thus, *hang-p* causes the process to hang until the input/output is complete, that is, until all the requested pages are there. The default value, **si:*default-page-in-hang-p***, is **t** by default.

normalize-p specifies whether the pages are "normal" (not flushable from main memory). *normalize-p* causes the paged-in pages to receive the "normal" page age rather than the "page-in" age. Its default value, **si:*default-page-in-normalize-p***, is **t** by default.

sys:page-in-array *array* &optional *from to (hang-p* **storage::*default-page-in-hang-p***) *(normalize-p* **t**) *array* &optional *from to (hang-p* **si:*default-page-in-hang-p***) *(normalize-p* **t**) *Function*

This is a version of **sys:page-in-structure** that can bring in a portion of an array. *from* and *to* are lists of subscripts; if they are shorter than the dimensionality of *array*, the remaining subscripts are assumed to be zero.

If *hang-p* is **t**, the function waits for the disk reads to finish before returning. Otherwise, the function returns immediately after requesting the disk reads, which might still be in progress. Thus, *hang-p* causes the process to hang until the input/output is complete, that is, until all the requested pages are there. The default value, **storage::*default-page-in-hang-p***, is **t** by default.

normalize-p specifies whether the pages are "normal" (not flushable from main memory). *normalize-p* causes the paged-in pages to receive the "normal" page age rather than the "page-in" age. Its default value, **si:*default-page-in-normalize-p***, is **t** by default.

sys:page-in-words *address n-words* &key *(:type* **storage::*default-page-in-type***) *(:hang-p* **storage::*default-page-in-hang-p***)* *Function*

Reads in any pages in the range of address space starting at *address* and continuing for *n-words* that have been swapped out to disk with as few disk operations as possible.

If *hang-p* is **t**, the function waits for the disk reads to finish before returning. Otherwise, the function returns immediately after requesting the disk reads, which might still be in progress. Thus, *hang-p* causes the process to hang until the input/output is complete, that is, until all the requested pages are there. The default value, **storage::*default-page-in-hang-p***, is **t** by default.

sys:page-in-area *area* &rest *page-in-words-keywords* *Function*

Brings into main memory all swapped-out pages of the specified area.

sys:page-in-region *region* &rest *page-in-words-keywords* *Function*

Brings into main memory all swapped-out pages of the specified region.

sys:page-out-structure *structure* &rest *page-out-words-keywords* *Function*

Similar to **sys:page-in-structure**, but takes pages out of main memory rather than bringing them in. Any modified pages are written to disk, using as few disk operations as possible. The pages are then made flushable; if they are not touched again soon, their memory is reclaimed for other pages. Use this operation when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:page-out-array *array* &optional *from to hang-p* *Function*

Similar to **sys:page-in-array**, but takes pages out of main memory rather than bringing them in. Any modified pages are written to disk, using as few disk operations as possible. The pages are then made flushable; if they are not touched again soon their memory is reclaimed for other pages. Use this operation when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

If *hang-p* is **t**, the function waits for the disk reads to finish before returning. Otherwise, the function returns immediately after requesting the disk reads, which might still be in progress. Thus, *hang-p* causes the process to hang until the input/output is complete, that is, until all the requested pages are there. The default value, **si:*default-page-in-hang-p***, is **t** by default.

sys:page-out-words *address n-words* &key *(:write-modified* **storage::*default-page-out-write-modified***) (:reuse* **storage::*default-page-out-reuse***)* *Function*

Similar to **sys:page-in-words**, but takes pages out of main memory rather than bringing them in. Any modified pages are written to disk, using as few disk operations as possible. The pages are then made flushable; if they are not touched again soon their memory is reclaimed for other pages. Use this operation when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:page-out-area *area* &rest *page-out-words-keywords* *Function*

Similar to **sys:page-in-area**, but takes pages out of main memory rather than bringing them in. Any modified pages are written to disk, using as few disk operations as possible. The pages are then made flushable; if they are not touched again soon their memory is reclaimed for other pages. Use this operation when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:page-out-region *region* &rest *page-out-words-keywords* *Function*

Similar to **sys:page-in-region**, but takes pages out of main memory rather than bringing them in. Any modified pages are written to disk, using as few disk operations as possible. The pages are then made flushable; if they are not touched again soon their memory is reclaimed for other pages. Use this operation when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:page-in-raster-array *raster* &optional *from-x from-y to-x to-y (hang-p* **si:*default-page-in-hang-p***) *(normalize-p* **t***)*

Function

Ensures that the storage that represents *raster* is in main memory. *from-x* and *from-y* can be specified as **nil**, meaning the lower limit for that item. *to-x* and *to-y* can be specified as **nil**, meaning the upper limit for that item.

This, rather than **sys:page-in-array**, should be used on rasters.

For a table of related items: See the section "Operations on Rasters".

sys:page-out-raster-array *array* &optional *from-x from-y to-x to-y* (*hang-p* **si:*default-page-in-hang-p***) *Function*

Takes the pages that represent *raster* out of main memory. *from-x* and *from-y* can be specified as **nil**, meaning the lower value for that item. *to-x* and *to-y* can be specified as **nil**, meaning the upper limit for that item.

This, rather than **sys:page-out-array**, should be used on rasters.

For a table of related items: See the section "Operations on Rasters".

storage:page-array-calculate-bounds *array to from* *Function*

Calculates the bounds of a page-in or page-out array. *from* and *to* are either fixnums or a list of subscripts. If they are fixnums then they are the flattened (coerced to one dimensional) array indices. If they are lists and the lists are shorter than the number of dimensions, zero is used for each missing element of *from* and the maximum index for the corresponding dimension is used for each missing element of *to*. (Therefore, **nil** for *from* means the start of the array and **nil** for *to* means the end of the array.) In all cases, *from* is inclusive and *to* is exclusive.

If the array is eventually displaced to an absolute address then **nil** is returned. Otherwise three values are returned: the array (after chasing indirect pointers), the starting address of data, and the number of words of data. Indirect arrays and indirect arrays with the element size changing are both supported.

Note: **sys:%find-structure-header** and **sys:%structure-total-size** are used to find the virtual memory location and extent of whole arrays or other structures to be wired. **storage:page-array-calculate-bounds** can be used to calculate the virtual memory location and extent of portions of an array that are to be be wired, when **storage:wire-words** or **storage:wire-consecutive-words** is used. **sys:%pointer-difference** can also be used to determine the length of the extent, in words, between two addresses obtained via these primitives or the **zl:aloc** function or the **locf** macro.

Wiring Memory

It is possible to *wire* objects in memory, in other words, lock them into physical memory. Wiring prevents them from being paged out or moved by the Genera system. This can greatly improve the response time of certain time-critical operations and references.

storage:wire-words *address number-of-words* *Function*

Wires at least *number-of-words* starting at the specified *address*. **storage:wire-words** wires any extent of virtual memory into physical memory, although the page frames into which successive pages are wired might not be contiguous.

storage:wire-consecutive-words *address number-of-words* *Function*

Wires at least *number-of-words* consecutively starting at the specified virtual memory address (*address*). **storage:wire-consecutive-words** wires any extent of virtual memory into physical memory. Successive pages are guaranteed to be stored in successive page frames in physical memory.

storage:unwire-words *address number-of-words* *Function*

Unwires at least *number-of-words* starting at the specific *address*. The first or last page of the range can stay wired if its wired-count does not go to zero because other words on that page are wired.

storage:with-wired-structure *structure* &body *body* *Macro*

Evaluates the body with the specified object wired in main memory. **storage:with-wired-structure** wires an entire structure (a convenience device to avoid having to calculate the location and extent of the virtual memory occupied by a structure).

storage:wire-structure *object* *Function*

Wires the *object* in main memory, in the manner of **storage:wire-words**. The preferred way to do this is with **storage:with-wired-structure**

storage:unwire-structure *object* *Function*

Unwires the structure *object*.

Ivory-Only Subprimitives

This section describes subprimitives available on all Ivory-based processors, such as the XL400, UX-family, and MacIvory machines. These subprimitives are not available on 3600-family machines. Use these subprimitives in inner loops of memory-intensive programs where maximizing efficiency is more important than machine independence.

Ivory Memory and Processor Architecture

Overview of Ivory Memory and Processor Architecture

The performance of modern microprocessor-based computers is often bounded not by the speed of the processor, but rather by the speed of its memory system. Typically, a large memory array will have an access time several times longer than the cycle time of the accompanying processor, because the power, space and cost requirements of the system necessitate the use of denser, slower memory technologies. The processor incurs a certain amount of memory traffic just to fetch the instructions in a given program, and executing those instructions incurs further traffic, depending on the details of the program. Without an architectural or technological solution to the processor/memory performance imbalance, the processor may spend a considerable fraction of its time idly waiting for its memory system.

A popular technological solution to the processor/memory performance mismatch is simply to use a faster memory array. Often, increasing the speed of the entire memory array is infeasible due to power, space, or cost limitations, but some of the advantages of this approach can be realized by using a *cache memory*. A cache

memory is a small, fast memory inserted in between the processor and its primary memory; locations in the slow primary memory are temporarily relocated to the fast cache memory whenever they are referenced, and subsequent references to that location may be processed at speeds comparable to that of the processor cycle time.

In a cache-based system, the cache memory is by definition substantially smaller than the primary memory, so only a small set of memory locations may reside in the cache simultaneously. While small, well-localized programs may reside entirely in the cache and benefit greatly from it, large programs that manipulate lots of data will waste some time migrating data between the cache and the memory, and therefore benefit less. In addition, the programmer, the language compiler, and the processor architecture all attempt to eliminate redundant memory references by caching information on chip (in the processor's register file or stack cache, for example); this obviously improves performance, but reduces the value of a cache which is optimizing the same thing. And finally, some memory traffic patterns derive no benefit at all from a cache; for example, graphics, image processing, and vector algorithms that require reading and writing large portions of memory.

An architectural solution to the processor/memory performance mismatch is to increase parallelism in the memory system. If, for example, a memory system is four times slower than a processor, then connecting four such memory systems to a single processor ought to balance things out some. This technique is called *memory interleaving*, and can only succeed if in fact the processor can effectively use all four memory systems at the same time. One means of doing this is to use *pipelining*, in which the processor/memory interface is partitioned into several stages which operate independently, so the interface can be working on several requests in different stages simultaneously. This is conceptually similar to, but not the same as, pipelining instruction execution, as is common in most modern microprocessors.

The Symbolics Ivory processor contains a pipelined memory interface that can track up to four simultaneous outstanding requests, and when supported with an interleaved memory system, can achieve sustained transfer rates of one memory word per processor cycle. The Ivory processor uses this memory interface to great advantage internally for fetching instructions, filling its map cache, filling and emptying its stack cache, looking up methods for a generic function, and operating on Lisp lists. The Genera system software uses this memory interface to great advantage in **bitblt** and other graphics operations, garbage collection and other object maintenance, process switching, and throughout the I/O system.

User programs may take direct advantage of the Ivory memory interface by using the documented subprimitives to optimize any inner loops containing sequential memory traffic. Subprimitives are available for:

- Reading or writing consecutive words between memory and the stack cache at maximum memory bandwidth (for example, the contents of an array may be copied to another at speeds asymptotically approaching two cycles per element)

- Reading consecutive words from memory at maximum memory bandwidth, simultaneously performing an arbitrary ALU operation on each word (for example, the sum of an array of integers may be computed at speeds asymptotically approaching one element per processor cycle)

- Reading consecutive words from memory at maximum memory bandwidth, simultaneously performing an arbitrary ALU operation on each word and evaluating a simple condition on the result, branching to specified code if the condition is true (for example, an array may be searched for a given element at speeds asymptotically approaching one element per processor cycle)

- Reading consecutive words from memory at maximum memory bandwidth, shifting them by a prespecified number of bits (this is useful for graphics operations)

Note that all these operators are subprimitives, meaning they operate below the level of the Lisp virtual machine, and can damage the Lisp environment if misused. Often the system cannot provide its normal consistency checks, such as array-bounds checking, when using these operations. If Genera's storage conventions are violated, system failure or unexpected behavior may ensue; quite often the garbage collector will find the inconsistency later and halt the system for debugging. Therefore, the recommended approach is to first use conventional Lisp operations such as **car** and **aref** when accessing memory. If the performance of the resulting code is insufficient, then careful use of the Ivory subprimitives can dramatically improve the speed of inner loops.

Correcting Memory Errors: the ECC Scrubber

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 rewriting 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 bitten 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".

Ivory Memory Addresses

The Ivory processor supports 32-bit virtual addresses and 32-bit physical addresses. The Genera operating system provides a demand-paged virtual memory of up to 4 gigawords, using as little as 1/2 megaword of main memory to page in. Although the Ivory processor supports a physical address space of 4 gigawords, it is not intended to support main memories that large. Rather, the large physical address space is used to accommodate large memory-mapped I/O devices such as high resolution color frame buffers and interfaces to other peripheral busses such as the VMEbus.

The Ivory processor operates on three different kinds of addresses: mapped virtual addresses, unmapped virtual addresses, and physical addresses. Normal Lisp objects always reside in virtual memory and their object references use mapped virtual addresses. Unmapped virtual addresses are untranslated virtual addresses used throughout the FEP, the Genera virtual memory software, and the Genera I/O software.

Physical addresses are Lisp object references with a special data type identifying them; when printed, they look like #<DTP-PHYSICAL-ADDRESS 760020220>. They can be created with **sys:%make-physical-address**, and identified using the type **sys:physical-address**. (Note that a physical address has the *data type* of DTP-PHYSICAL-ADDRESS, and it has the *type* **sys:physical-address**.) Although technically physical addresses are Lisp objects, they cannot be operated on using normal Lisp functions such as **car**. They may only be referenced using subprimitives such as **sys:%p-dpb**, **sys:%p-ldb**, **sys:%memory-read** and related functions, and **sys:%block-read** and related functions. However, arrays may be displaced to physical addresses, and all the normal array operators will work.

The subprimitives that reference memory do not do any kind of type checking on their address argument. When that address is presented to the Ivory memory mapping hardware, if the data type is **sys:dtp-physical-address**, then the address is passed through untranslated. Otherwise, the address is treated as a virtual address (note that although fixnums will be interpreted as virtual addresses, locatives should be used instead, so that if the garbage collector moves the object it points to the reference will be updated). If the byte field **sys:%%vma-equals-pma** indicates that the virtual address is in the high 1/32nd of the address space, then it is an unmapped address which is translated directly to the low 1/32nd of the physical address space as shown in Figure ! .

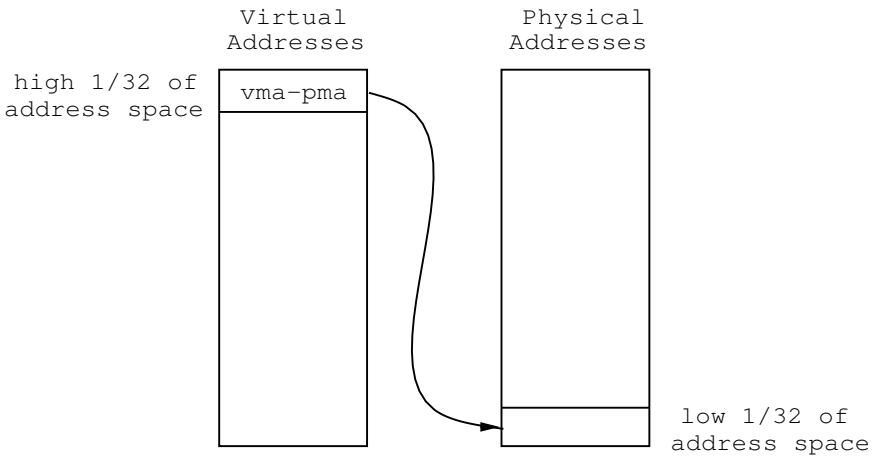


Figure 35. Mapping of Virtual to Physical Addresses

Ivory Subprimitives for Handling Memory Addresses

This section documents Ivory subprimitives that deal with physical and virtual memory addresses.

sys:%%vma-equals-pma *Constant*

A byte specifier for the byte in a virtual address which is used to determine if the address is mapped or unmapped. See the constant **sys:%vma-equals-pma**.

sys:%%vma-pma *Constant*

A byte specifier for the byte in an unmapped virtual address which corresponds to the physical address.

sys:%make-physical-address *pma* *Macro*

pma must be a fixnum. Returns a physical address.

sys:%vma-equals-pma *Constant*

The value of the **sys:%%vma-equals-pma** field in a virtual address when the address is unmapped.

storage:%vma-to-pma *vma* *Function*

vma is a virtual address (a pointer to an object, of any type). **storage:%vma-to-pma** determines whether that virtual address is resident in main memory. If so, it returns the corresponding physical address (a pointer with data type **dtp-physical-address**); if not, it returns **nil**. Note that the address translation is valid only at the instant it is made, because the virtual memory system may swap that page out immediately upon return. **storage:%vma-to-wired-pma** is generally more useful.

storage:%vma-to-wired-pma *vma* *Function*

vma is a virtual address (a pointer to an object, of any type). **storage:%vma-to-wired-pma** determines whether that virtual address is resident and wired in main memory. If so, it returns the corresponding physical address (a pointer with data type **dtp-physical-address**); if not, it returns **nil**. The address translation is valid until the address in question is unwired.

sys:physical-address *Type Specifier*

sys:physical-address is the type specifier for a physical address.

Ivory Memory Operations

The Ivory processor implements a number of subprimitive instructions to read and write memory. There are operations to read and write individual memory locations (**sys:%memory-read** and **sys:%memory-write**), follow forwarding pointers (**sys:%memory-read-address**), and operations for more efficiently reading and writing blocks of consecutive memory locations (**sys:%block-read** and related functions). These operations are useful for implementing and circumventing the high-level Lisp data model (for example, Genera uses them to create, initialize, and garbage-collect Lisp objects such as lists and arrays), for I/O operations that involve memory locations outside Lisp virtual memory, and for improving performance in critical inner loops.

The subprimitive memory operations may be used on any memory addressible by Ivory, including virtual, unmapped, and physical addresses. See the section "Ivory Memory Addresses". Most of them have options to control the data type checking, invisible pointer following, and garbage-collection features of Ivory's memory interface. The following options are common to all the subprimitives for reading memory locations:

:cycle-type
: Controls the datatype checking and invisible-pointer following performed on the contents of the memory location. The default is to use the same kind of memory cycle that **car** uses, which should be suitable for most usage.

:fixnum-only
: When **:fixnum-only** is true, an error trap occurs if the data read is

not a fixnum. This is a useful defensive measure when operating on bit arrays or I/O devices, for example.

:set-cdr-next
When true, the cdr-code of the data read is set to cdr-next, otherwise it is preserved. The Ivory function calling architecture requires that passed arguments always have cdr-next cdr-codes, so the default is to do that.

See the section "Forwarding Words in Memory" for more information about invisible pointers. See the section "Memory Cycle Types" for more information about the **:cycle-type** option.

The subprimitive memory operations work only in compiled code, not in interpreted code.

Ivory Subprimitives for Memory Operations

sys:%memory-read *address* &key *(:cycle-type* **sys:%memory-data-read***) :fixnum-only (:set-cdr-next* **t***)* *Macro*

Reads and returns the word at *address* with the specified *cycle-type*. If *fixnum-only* is **t**, an error is signaled if the word is not a fixnum. If *set-cdr-next* is **t**, the cdr-code of the result will be cdr-next.

In most cases where *address* is a locative, location-contents is preferable.

sys:%memory-read-address *address* &key *(:cycle-type* **sys:%memory-data-read***) :fixnum-only (:set-cdr-next* **t***)* *Macro*

Reads the word at *address* with the specified *cycle-type*. If *fixnum-only* is **t**, an error is signaled if the word is not a fixnum. If *set-cdr-next* is **t**, the cdr-code of the result will be cdr-next.

This macro returns the resulting address with the same data type as its argument. If the *cycle-type* follows invisible pointers, then the result of **sys:%memory-read-address** is the address reached after any invisible pointers are followed. If there are no invisible pointers, then the result is *address*.

This subprimitive supports the higher-level functions **follow-cell-forwarding** and **follow-structure-forwarding**. See the function **follow-cell-forwarding**. See the function **follow-structure-forwarding**.

sys:%memory-write *address data* *Macro*

Writes *data* into the memory address specified by *address*. This operation does not follow invisible pointers, and does not preserve the cdr code in memory at *address*.

Ivory Block Memory Operations

The Ivory processor implements a number of subprimitive instructions for efficiently reading and writing blocks of consecutive memory locations, with variants that can shift, test, and/or perform ALU operations on the result. These operations use the pipelined Ivory memory interface to achieve very high data transfer bandwidths, and when carefully applied can significantly improve the performance of algorithms that operate on blocks of memory. See the section "Overview of Ivory Memory and Processor Architecture" for an explanation of the Ivory memory system.

Each block memory instruction may use one of four *block address registers* to indicate the address to operate on. See the section "Ivory Block Address Registers". The operation of the **sys:%block-write** instruction is fairly straightforward: it writes a value into the location indicated by the specified block address register, then increments that register. The block instructions that read memory (**sys:%block-read**, **sys:%block-read-alu**, **sys:%block-read-shift**, and **sys:%block-read-test**) are more complex; their basic behavior is to:

- Read the memory location indicated by the specified block address register.
- Perform the specified data type and invisible pointer checking on the result.
- For **sys:%block-read-alu** or **sys:%block-read-shift**, operate on or shift the result.
- For **sys:%block-read-test**, test the result for a specified condition and possibly branch out of a loop.
- Push the result on the stack.
- Optionally increment the specified block address register.

In addition, the block read instructions do some behind-the-scenes negotiating with the Ivory memory interface to prefetch subsequent memory words so that they will be ready for consumption at the exact cycle in which they are needed. After an initial transient, the memory pipeline and instruction pipeline synchronize, and all the above operations take place in a single processor cycle.

To achieve optimal performance using block operations, care must be taken to order the instructions properly, to avoid stalling the memory pipeline. We recommend that a sequence of several (four to eight) block read instructions be executed consecutively, although this is not absolutely necessary and other strategies will provide good performance. In particular, when using block read and block write instructions together, it is best to read a number of words, write those out, read some more, write those out, and so on, instead of simply reading and writing a single word at a time. The **sys:unroll-block-forms** macro can help structure block memory loops. See the section "Loop Unrolling Technique".

The block read operations take the same **:cycle-type**, **:fixnum-only**, and **:set-cdr-next** options that the memory subprimitives do. They also take a **:no-increment** option to inhibit the post-increment of the block address register, which is useful when you want to read a word, modify it, then write it back to the same location. The block read operations also take a **:prefetch** option that is useful when you know you won't be reading further words; you would supply **nil** when reading the last word of a block. This is an optimization to reduce memory interface contention.

Note that the MacIvory processor does not fully implement pipelined memory operations, due to the limitations of the Macintosh Nubus. The documented block operations work, but may not provide substantial performance improvement.

Also note that the block memory operations work only in compiled code, not in interpreted code.

Ivory Subprimitives for Block Memory Operations

Many of the block operations work in conjunction with the ALU-CONTROL and ROTATE-LATCH registers. For information on those registers, see the section "Ivory ALU-CONTROL and ROTATE-LATCH Registers".

sys:with-block-registers *(*&rest *registers)* &body *body* *Macro*

Any use of a block register should occur within the *body* of a **sys:with-block-registers** form. This Ivory subprimitive implements the correct protocol for saving/restoring the contents of the block register. The *registers* are integers: 1 indicates BAR-1; 2 indicates BAR-2; 3 indicates BAR-3.

For information on the calling conventions of the three block registers, see the section "Ivory Block Address Registers".

For examples, see the section "Examples of Using Ivory Subprimitives".

sys:%block-register *n* *Macro*

Returns the contents of the block register numbered *n*. Use **setf** to set a block register.

sys:%block-write *bar value* *Macro*

Writes the *value* into the memory location that the *bar* points at, and then increments the *bar* to point at the next word in memory. This macro does not follow invisible pointers, and does not preserve the cdr code in memory.

This is a block operation that can increase performance when you are writing data at consecutive memory locations. If you are only writing data at one address, then you should use **sys:%memory-write**, which does not require setting up and using a block register.

sys:%block-read *bar* &key *(:cycle-type* **sys:%memory-data-read***) :fixnum-only (:set-cdr-next* **t***) (:prefetch* **t***) :no-increment* *Macro*

Reads the word from memory that the *bar* points at, and then increments the *bar* to point at the next word in memory.

This is a block operation that can increase performance when you are reading data at consecutive memory locations. If you are only reading data from one address, then you should use **sys:%memory-read**, which does not require setting up and using a block register.

:cycle-type *cycle-type*

Controls the datatype checking and invisible-pointer following performed on the contents of the memory location. The default is to use the same kind of memory cycle that **car** uses, which should be suitable for most usage. *Cycle-type* is one of the following fixnum constants:

sys:%memory-data-read
sys:%memory-data-write
sys:%memory-raw
sys:%memory-header
sys:%memory-scavenge

For information on the semantics of the memory cycle types, see the section "Memory Cycle Types".

:fixnum-only *boolean*

When **:fixnum-only** is true, then an error trap occurs whenever the data read is *not* a fixnum. This is a useful defensive measure when operating on bit arrays or I/O devices, for example. The condition signaled is **sys:bad-data-type-in-memory**, or sometimes one of its more specialized flavors such as **sys:unbound-function**.

:set-cdr-next *boolean*

When true, the cdr-code of the data read is set to cdr-next, otherwise it is preserved. The Ivory function calling architecture requires that passed arguments always have cdr-next cdr-codes, so the default is to do that.

:prefetch *boolean* A value of **nil** can be supplied if you know you won't be reading further words; you would supply **nil** when reading the last word of a block. This is an optimization to reduce memory interface contention. The default is **t**. This option affects performance only, so if you request no prefetching and then proceed to read the next word, the word will be read without a problem. Similarly, there is no problem if a word is prefetched and then not needed. If you use **sys:unroll-block-forms**, the prefetching is taken care of automatically.

:no-increment *boolean*

A true value prevents the block register from being incremented to point at the next word in memory. This can be useful when you want to read a word from memory, modify it, then write it back to the same location. You would supply a true value for **:no-increment** for the block read operation.

sys:%block-read-alu *bar arg* *Macro*

Performs an ALU operation on two operands. OP1 is the word addressed by *bar*, which is 1, 2, or 3. OP2 is *arg*. The ALU-CONTROL register determines the operation that will be performed. The result is placed in *arg*, which must be a local lexical variable.

The cdr code of the operand is set to the cdr code from memory. The specified *bar* is incremented. This instruction always uses a memory cycle type of **sys:%memory-data-read**. This instruction traps if either operand is not a fixnum, or if arithmetic overflow occurs in an ALU function that checks for overflow.

sys:%block-read-shift *bar* &key *(:cycle-type* **sys:%memory-data-read***) :fixnum-only (:set-cdr-next* **t***) (:prefetch* **t***) :no-increment* *Macro*

Reads the word addressed by the block register specified by *bar* (an integer 1, 2 or 3) and rotates it left by the amount specified in the byte-r field of the ALU-CONTROL register.

The top (byte-s + 1) bits come from this rotated word, and the bottom bits come from the ROTATE-LATCH register, and this value is pushed onto the stack. The ROTATE-LATCH register is then loaded from the rotated memory word. The effect of this operation is to perform a **dpb** (deposit-byte) of the word from memory into the ROTATE-LATCH register, and to push the result on the stack. The specified *bar* is incremented.

For information on the keyword options, see the macro **sys:%block-read**.

sys:%block-read-test *bar* &key *(:cycle-type* **sys:%memory-data-read***) :fixnum-only (:set-cdr-next* **t***) (:prefetch* **t***) :no-increment :require-tos-valid* *Macro*

Performs a test on two operands: OP1 is the word read from memory, and OP2 is specified by **sys:%block-read-test-tagbody**. The test should be a predicate that returns true or false.

The operation specified by the ALU-CONTROL register is performed, and the condition is tested according to the condition sense in the ALU-CONTROL register. If successful, the *bar* is not incremented and a branch is taken to the tag specified by **sys:%block-read-test-tagbody**. If the test is not successful, it is as if a **sys:%block-read** had been performed (although no value is returned). This instruc-

tion should normally only be used with the **sys:%block-read-test-tagbody** special form.

This instruction is typically used for searching tables and bitmaps, and by the garbage collector.

:require-tos-valid is a boolean. It should be **t** if the test depends on OP2.

For information on the other keyword options, see the macro **sys:%block-read**.

sys:%block-read-test-tagbody *(success-tag* &key *operand-2)* &body *body* *Macro*

Like **tagbody**, except that it prepares for uses of **sys:%block-read-test**. *success-tag* should be a tag to be branched to if **sys:%block-read-test** succeeds. When the condition requires a second operand, *operand-2* is that operand.

The PC corresponding to *success-tag* will be pushed on the stack, followed by *operand-2* (or **nil** if *operand-2* is not supplied). When a **sys:%block-read-test** is executed, there must be no additional objects on the stack at the time.

Legal example:

```
(let ((count n))
  (sys:%block-read-test-tagbody (success :operand-2 x)
     loop
       (when (= count 0)
         (go ran-out))
       (sys:%block-read-test 1)
       (decf count)
       (go loop)
     success
        ...
     ran-out
        ...))
```

Illegal example:

```
(sys:%block-read-test-tagbody (success :operand-2 x)
   (loop repeat n doing
      (sys:%block-read-test 1))
   ...
   success
   ...)
```

This is illegal because **loop** will generate a temporary variable which will be located on the stack above what is pushed by **sys:%block-read-test-tagbody**.

Ivory Hardware Registers

Ivory has a number of hardware registers which control its operation. Most of these registers are only of interest to Ivory or very low levels of the system, and should not be used by users. However, a few are useful in performance-critical ap-

plications. These registers are virtual, that is, each stack group maintains its own set of values. The debugger, the scheduler, interrupt handlers, and trap handlers all ensure that they do not disturb the state of these registers.

This section describes the Block Address Registers, or BARs (see the section "Ivory Block Address Registers"), and the ALU-CONTROL and ROTATE-LATCH registers (see the section "Ivory ALU-CONTROL and ROTATE-LATCH Registers").

When using hardware registers, certain conventions must be followed to preserve their contents. One convention is "callee saves". This means that if a function modifies a register, it is required to restore the original value of the register. Another convention is "caller saves". This means that if you are using a register and call something which is "outside" of your code's dominion, then you should expect that that register will have a different value when that call has completed. "Calling" means calling a function; instructions will not change the value of a hardware register unless they are documented to do so. For information on which instructions change Ivory registers,

see the section "Instructions That Change Ivory Registers".

Ivory Block Address Registers

Ivory makes all memory references through one of four *Block Address Registers* (BARs). A BAR contains a memory address at which an operation is to be performed. Associated with each BAR is a memory data register which receives memory data when reading memory. All memory references from Ivory use one of these four BARs.

User programs may take advantage of the Ivory memory interface by using the block operations (which operate with BARs) to optimize any inner loops containing sequential memory traffic. The goal is to group the memory traffic into longer sequences so the memory pipeline can operate efficiently. For a general description of memory pipelining, see the section "Overview of Ivory Memory and Processor Architecture".

The BARs are numbered from 0 to 3 inclusive. BAR-0 is used by the processor for instruction fetches, by many instructions (such as **car** and **aref**) and by asynchronous exception handlers. It is not generally useful to software. BAR-1, BAR-2 and BAR-3 are used by a few instructions and by some system software, and can be used by application software as long as certain conventions are obeyed.

BAR-2 and BAR-3 are "callee saves", and BAR-1 is "caller saves". The form **sys:with-block-registers** should be used around any code which uses any of the BARs. You can use **sys:%block-register** to get the contents of a block register, and use **setf** with it to set a block register. These subprimitives are documented elsewhere; See the section "Ivory Subprimitives for Block Memory Operations".

For examples of using block registers, see the section "Examples of Using Ivory Subprimitives".

Ivory ALU-CONTROL and ROTATE-LATCH Registers

Ivory machines offer two special registers which can be used with the block registers: the ALU-CONTROL register and the ROTATE-LATCH register. Both of these registers are "caller saves". **Caution**: if you use these registers, you should see the section "Instructions That Change Ivory Registers".

The ALU-CONTROL register is used by these Ivory subprimitives:

sys:%alu
sys:%block-read-alu
sys:%block-read-test
sys:%block-read-shift

These operations optionally use the ROTATE-LATCH register as well. These operations use byte-fields in the ALU-CONTROL register to route data through the Ivory chip.

The ALU-CONTROL register has the following fields:

Function
Byte-R
Byte-S
Condition
Condition-Sense

Before using the ALU-CONTROL register, you must prepare the register for use by calling **sys:set-alu-and-rotate-control**. This macro sets the various fields of the register.

Ivory Subprimitives for Using the ALU-CONTROL and ROTATE-LATCH Registers

This section documents the macro that sets up the ALU-CONTROL and ROTATE-LATCH registers, and one macro that works with the ALU-CONTROL register. Most operations that use these registers are block operations. For background information on block registers, see the section "Ivory Block Address Registers".

For reference documentation on the block operations, see the section "Ivory Subprimitives for Block Memory Operations".

sys:set-alu-and-rotate-control &key *:byte-r :byte-s :function :condition :condition-sense* *Macro*

Prepares the ALU-CONTROL register for use. The keyword arguments are:

:byte-r *bits*
Specifies the value for the BYTE-R field of the ALU-CONTROL register. See below for details.

:byte-s *bits*
Specifies the value for the BYTE-S field of the ALU-CONTROL register.

A complete byte field is specified by the BYTE-R and BYTE-S fields of the ALU-CONTROL register. This byte field is similar to (BYTE (+ BYTE-S 1) BYTE-R), except that it can wraparound to the low-order bits. For example, in the following table, the shaded bits (indicated by #) are in the specified fields:

```
 S  R  | 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
       | 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
-------------------------------------------------------------------------
 5  3                                                  # # # # # #
10 28    # # # #                                          # # # # # # # #
```

OP2 is rotated left by R bits. In the **ldb** case, the low S+1 bits of the result come from the rotated OP2, and the remaining bits come from *background*. In the **dpb** case, the specified field comes from the rotated OP2, and the remainder comes from *background*. If the ROTATE-LATCH is set, it is set to the rotated OP2.

:function *operation*

Specifies the operation to be performed. It can take on the following values:

BOOLE operations

If the **:function** argument is an integer from 0 through 15, then **(boole** *op op1 op2***)** will be performed. The integer specifies the *op*, just as it does in the **boole** function. See the function **boole**.

(**sys:%alu-function-ldb** *background rotate-latch*)

Specifies that an **ldb** operation should be performed.

background is one of:

sys:%alu-byte-background-op1
sys:%alu-byte-background-rotate-latch
sys:%alu-byte-background-zero

rotate-latch is one of:

sys:%alu-byte-hold-rotate-latch
sys:%alu-byte-set-rotate-latch

For information on the semantics of *background* and *rotate-latch*, see the description of the **:byte-s** option.

(**sys:%alu-function-dpb** *background rotate-latch*)

Specifies that a **dpb** operation should be performed. *background* and *rotate-latch* can take on the values documented in the **ldb** case. For information on the semantics of *background* and *rotate-latch*, see the description of the **:byte-s** option.

(**sys:%alu-function-add** *2nd-input-source 2nd-input-invert carry-in*)

The adder is a 32-bit adder with some additional logic useful for performing a number of comparison operations (see ALU conditions). The 32-bit adder takes one input from OP1. The other input can be 0, -1,

OP2, or (LOGNOT OP2). This is specified with the following argument values:

2nd-input-source	*2nd-input-invert*	2nd input value
sys:%alu-add-op-2	0	OP2
sys:%alu-add-op-2	1	(LOGNOT OP2)
sys:%alu-add-zero	0	0
sys:%alu-add-zero	1	-1

The result will be the 32-bit sum of the two inputs and *carry-in* (which must be 0 or 1).

2nd-input-invert does not affect *carry-in*, so when using **sys:%alu-function-add** for subtraction, *carry-in* should be 1 if there is no borrow and 0 if there is a borrow.

Examples:

```
(sys:set-alu-and-rotate-control
  (:function
    ;Adds OP1 and OP2
    (sys:%alu-function-add sys:%alu-add-op-2 0 0)))

(sys:set-alu-and-rotate-control
  (:function
    ;Adds 1 to OP1
    (sys:%alu-function-add sys:%alu-add-zero 0 1)))

(sys:set-alu-and-rotate-control
  (:function
    ;Subtracts OP2 from OP1
    (sys:%alu-function-add sys:%alu-add-op-2 1 1)))

(sys:set-alu-and-rotate-control
  (:function
    ;Subtracts 1 from OP1
    (sys:%alu-function-add sys:%alu-add-zero 1 1)))
```

:condition *condition*

Sets the condition field. The condition field is used only by the **sys:%block-read-test** instruction. *condition* can be one of the named conditions below. The arithmetic condition descriptors refer to the following signals:

`Invert`	The value of *2nd-input-invert* in **sys:%alu-function-add**
`Sign1`	Bit 31 of input 1
`Sign2`	Bit 31 of input 2
`Zero`	Bits 31:0 of the result are 0

Cout	The carry from the 32-bit adder
Sign	Bit 31 of the result
XSign	Cout XOR Sign1 XOR Sign2
Overflow	XSign XOR Sign

sys:%alu-condition-signed-less-than-or-equal

This condition is true if the possibly overflowing operation on twos complement fixnum inputs was less than or equal to 0, that is. if the two inputs are sign-extended to 33 bits, this condition is true if the twos complement 33 bit result is less than or equal to 0.

Zero OR XSign=1.

sys:%alu-condition-signed-less-than

This condition is true if the possibly overflowing operation on twos complement fixnum inputs was less than 0, that is. if the two inputs are sign-extended to 33 bits, this condition is true if the twos complement 33 bit result is less than 0.

XSign=1.

sys:%alu-condition-negative

This condition is true if the two's complement result is negative.

Sign=1.

sys:%alu-condition-signed-overflow

This condition is true if the operation on the two's complement inputs overflowed.

Overflow.

sys:%alu-condition-unsigned-less-than-or-equal

This condition is true if the inputs are 32-bit unsigned integers, a subtract was performed, and the result was less than or equal to 0.

Zero OR (NOT(Invert) XOR Cout).

sys:%alu-condition-unsigned-less-than

This condition is true if the inputs are 32-bit unsigned integers, a subtract was performed, and the result was less than 0.

NOT(Invert) XOR Cout.

sys:%alu-condition-zero

The condition will be true when the low 32 bits of the result are 0.

Zero.

sys:%alu-condition-eq

The condition will be true when the result is 0 and the data types of OP1 and OP2 are the same. The following example shows how to write an **eq** test:

```
(sys:set-alu-and-rotate-control
      :condition %alu-condition-eq
      :condition-sense %alu-condition-sense-true
      :function cl:boole-xor)
```

sys:%alu-condition-false

The condition is always false.

sys:%alu-condition-result-cdr-low

The condition will be true if the cdr-code of OP1 is either CDR-NIL or 3. Note that **:set-cdr-next** must be **nil** in **sys:%block-read-test** for this to work.

:condition-sense *condition-sense*

Sets the condition sense field, which is used only by the **sys:%block-read-test** instruction. *condition-sense* can be:

sys:%alu-condition-sense-true
Causes **sys:%block-read-test** to branch if the condition is true.

sys:%alu-condition-sense-false
Causes **sys:%block-read-test** to branch if the condition is false.

sys:%alu *op1 op2* *Macro*

Performs the operation specified by the ALU-CONTROL register on the operands *op1* and *op2*, and returns the result.

Instructions That Change Ivory Registers

The following table shows which instructions change the BAR, ALU-CONTROL, and ROTATE-LATCH registers. In general, BAR-2 and BAR-3 are not changed by instructions; however, each one can be changed by a Joshua instruction. Care must be taken if you program both with block registers and with Joshua.

Instruction	BAR	ALU-CONTROL	ROTATE-LATCH
%allocate-list-block	1		
%lshc-bignum-step		*	*
aref-1		*	
aset-1		*	
ash		*	
assoc	1		
bind-locative	1		
bind-locative-to-value	1		
fast-aref-1		*	
fast-aset-1		*	
lsh		*	
push-global-logic-variable	2		
rot		*	
unify	3		

Memory Cycle Types

When reading from memory, it is often desirable to specify the expected format of the data being read. The processor can then check for this format in parallel with instruction execution, and trap appropriately if the data does not match the expected format. This relieves software from explicitly checking memory data, and from interlocking with other memory references. The ability to have hardware perform data-type checks in parallel with memory operations is one of the major advantages of Lisp processors.

For example, suppose we wanted to read the value cell of a symbol, and return its value to an application. The symbol could be unbound, in which case we should signal an error. We therefore issue the memory request such that unbound-memory-location markers are detected by the hardware. On the other hand, we may want to read the value cell of a symbol simply to check if it is bound. In that case, we don't want the hardware to trap on unbound-memory-location markers.

The memory cycle type controls:

- Invisible pointer following
- GC transporting of references to oldspace
- Invalid data type trapping (unbound variable, malformed memory)
- Location monitoring

On read operations, the parallel data-type checking hardware is controlled via the **:cycle-type** and **:fixnum-only** options to **sys:%memory-read**, **sys:%memory-read-address**, **sys:%block-read**, **sys:%block-read-shift**, and **sys:%block-read-test**.

The memory cycle types are:

sys:%memory-data-read *Constant*

This is the default memory cycle type, and it is usually the most appropriate one to use for block register read operations. This is used for most operations that read ordinary data from memory, such as **car** and **aref**. This memory cycle type reads the word located at the requested memory address. If an invisible pointer is obtained, then that invisible pointer is followed to its end. Block operations when invisible pointers are present are confusing and not recommended. The block register is reset to wherever the invisible pointer points to.

Note that using **sys:%memory-read** and **sys:%memory-read-address** should not be confusing when invisible pointers are present.

This memory cycle type traps on the usual situations, including on unbound variables and monitor.

sys:%memory-data-write *Constant*

This memory cycle type is used with read operations when the goal is to read a word, then write that word. Invisible pointers are followed, so the next write operation writes to the correct location.

sys:%memory-raw *Constant*

This memory cycle type reads anything. A raw memory reference has all the indirection (invisible pointer following), trapping, and transporting possibilities disabled. Using this memory cycle type can be dangerous because the GC transporter is turned off; if you store the value somewhere, you can subvert the garbage collector.

sys:%memory-header *Constant*

This memory cycle type is used by the system when referencing headers of some data structures, including arrays and instances. It doesn't follow any forwarding pointers, but enables transporting of objects condemned by the garbage collector.

sys:%memory-scavenge *Constant*

This memory cycle type is used by the garbage collector. This is primarily an internal operation, and not generally useful to application programs.

Ivory Array Registers

An Ivory array register is a decoded form of an array, consisting of four consecutive words on the stack which cache the information in an array's header. They permit faster access because no reference to the header is required.

On Ivory machines, you can use the subprimitives underlying the array registers. The higher-level array register facility is documented elsewhere: See the section "Array Registers".

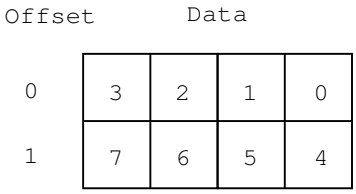
For examples of using the subprimitives underlying array registers, see the section "Examples of Filling Arrays Using Block Registers".

Reasons for Using Array Registers

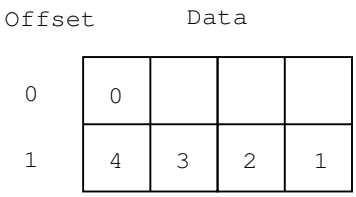
When you use a block register, you need to know the address, actual length, and byte-packing of the data. If your data are in an array, you can use the Ivory subprimitives for dealing with array registers to obtain this information efficiently for an arbitrary array. Note that if you know this information about arrays used in your application, it is not necessary to use array registers.

The following situations show how the suprimitives can be useful:

- When there is more than one array element per word, the elements are stored from the low bytes to the high bytes:



- If the array is displaced with an index offset, then the array elements may be shifted in the words.



- If an array A is displaced to B, the actual number of usable elements in A may be shorter than the length of A because B may be shorter than the length of A plus the index offset:

```
(setq b (make-array 50.))
(setq a (make-array 100. :displaced-to b
                         :displaced-index-offset 10.))
```

  The array A really can use only 40 of its elements.

- If your array is not an object array, **locf** cannot be used to find the address of the first element.

By using the array register subprimitives, your application does not depend upon the current Symbolics implementation of arrays on Ivory (note, however, that array registers are different on 3600-family machines). The array register subprimitives enable you to conveniently determine the address of the first array element, the offset of the first element in that word, the number of elements per word, and the usable length of the array (in array elements). In addition, an event count is maintained which can be used to tell if the array register values may need to be updated (you need be concerned about this only if the array may have been adjusted since you got the array register values). For an example of getting this information about an array, see the section "Decoding an Array with Array Register Subprimitives".

Components of an Ivory Array Register

The four components of an array register are:

Array A reference to the array object itself.

Control word
A fixnum containing the following fields:

sys:array-register-element-type
Describes the element type of the array. The possible values are:

sys:array-element-type-fixnum
sys:array-element-type-character
sys:array-element-type-boolean
sys:array-element-type-object

sys:array-register-byte-packing
This is Log_2(elements per word).

sys:array-register-byte-offset
The number of unused elements preceding the array in the first word. This is a number to be added to the specified index in **sys:fast-aref-1** and **sys:fast-aset-1**. It is non-zero only for displaced arrays, and always represents an offset less than one word.

sys:array-register-event-count
A snapshot of a count which is incremented whenever an event which might have invalidated the cached array information has occurred. If this value differs from the value returned by **(sys:%read-internal-register sys:%register-event-count)** then it is possible that the array register is stale (due to an array being adjusted, for example, but not due to garbage collection). If you know that your array has not been adjusted since you set up the array register, you do not need to worry about this.

Base address
The address of the first element in the array. If the array is displaced, it does not necessarily include all the bits in the word at this address.

Array length

The number of elements in the array. In the case of a displaced array, this may be smaller than the value returned by **array-total-size** since each indirection imposes its own length restrictions.

Creating an Ivory Array Register

To create an Ivory array register, use one of the following:

sys:setup-1d-array *array*

Creates an array register describing *array*, which must be a one-dimensional array.

sys:setup-force-1d-array *array*

Creates an array register describing *array*, which can be any array. This function causes multidimensional arrays to be accessed as if they were one-dimensional arrays, with the order of elements in row-major order.

These return four values, which are the components of the array register. For information on these components, see the section "Components of an Ivory Array Register".

Note that the values returned must be kept in the order in which they are returned; that is, there must be four consecutive words on the stack containing an array, a control word, a base address, and a length.

When you create an array register using one of these subprimitives, you cannot use the normal Lisp array functions to manipulate it. To read or write an element of the array, use **sys:fast-aref-1** or **sys:fast-aset-1**.

Decoding an Array with Array Register Subprimitives

This function illustrates how to "decode" an array register; that is, to get information about the array, such as its:

- Base address
- Element type
- Byte packing
- Byte offset

```
(defun quick-describe-array (array)
  (multiple-value-bind (array control base-address length)
      (sys:setup-1d-array array)
    (declare (ignore array))
    (format t "~&The first element of the array is at location ~s"
            base-address)
    (format t "~&The array elements are ~a"
            (select (ldb sys:array-register-element-type control)
                (sys:array-element-type-fixnum "fixnums")
                (sys:array-element-type-character "characters")
                (sys:array-element-type-boolean "boolean")
                (sys:array-element-type-object "objects")))
    (let ((byte-packing (ldb sys:array-register-byte-packing control)))
      (format t "~&The byte-packing is ~s, so there are ~s elements per word"
              byte-packing
              (rot 1 byte-packing)))
    (format t "~&Array element 0 is ~s elements into the first word"
            (ldb sys:array-register-byte-offset control))
    (format t "~&There are ~s usable elements in the array" length)))
```

This function uses Ivory subprimitives for dealing with array registers. For information on the subprimitives, see the section "Ivory Subprimitives for Handling Array Registers".

Caveats Regarding Ivory Array Registers

When you use the higher-level interface for array registers (the **sys:array-register** declaration), the system ensures that the array register is valid as it is being used. That is, if the array is adjusted, the array register re-encaches the new state of the array.

The subprimitives **sys:fast-aref-1** and **sys:fast-aset-1** also re-encache the state of the array if it changes. However, if you copy values out of the array register (such as its length), these values will not be updated if the array is adjusted.

When you use the Ivory subprimitives for array registers, you have two choices of how to deal with this potential problem. In some situations, you know that the array will not be adjusted, so the problem should not occur. If, however, you anticipate that the array might be adjusted, then you will have to establish a protocol such as locking to protect the integrity of the array.

For example, the window system locks a lock whenever it draws on a window, and whenever it adjusts a screen array, so that these two operations cannot interfere with each other.

Ivory Subprimitives for Handling Array Registers

sys:array-register-byte-offset *Constant*

This constant is a byte specifier for the **sys:array-register-byte-offset** field in the control word of an array register. See the section "Components of an Ivory Array Register".

sys:array-register-byte-packing *Constant*

This constant is a byte specifier for the **sys:array-register-byte-packing** field in the control word of an array register. See the section "Components of an Ivory Array Register".

sys:array-register-element-type *Constant*

This constant is a byte specifier for the **sys:array-register-element-type** field in the control word of an array register. See the section "Components of an Ivory Array Register".

sys:array-register-event-count *Constant*

This constant is a byte specifier for the **sys:array-register-event-count** field in the control word of an array register. See the section "Components of an Ivory Array Register".

sys:array-element-type-boolean *Constant*

Indicates that the elements of the array described by a given array register are boolean values (**t** or **nil**). The words in memory are of type **dtp:fixnum**. Each bit is 0 for **nil** or 1 for **t**.

This constant is one of four possible values to be returned by the **sys:array-register-element-type** macro, and one of four possible values for the **sys:array-register-element-type** field in the control-word component of an array register. See the section "Components of an Ivory Array Register".

sys:array-element-type-character *Constant*

Indicates that the elements of the array described by a given array register are characters. The words in memory are of type **dtp:fixnum**.

This constant is one of four possible values to be returned by the **sys:array-register-element-type** macro, and one of four possible values for the **sys:array-register-element-type** field in the control-word component of an array register. See the section "Components of an Ivory Array Register".

sys:array-element-type-fixnum *Constant*

Indicates that the elements of the array described by a given array register are fixnums. The words in memory are of type **dtp:fixnum**.

This constant is one of four possible values to be returned by the **sys:array-register-element-type** macro, and one of four possible values for the **sys:array-register-element-type** field in the control-word component of an array register. See the section "Components of an Ivory Array Register".

sys:array-element-type-object *Constant*

Indicates that the elements of the array described by a given array register can be any kind of Lisp object. The words in memory can be of any type.

This constant is one of four possible values to be returned by the **sys:array-register-element-type** macro, and one of four possible values for the **sys:array-register-element-type** field in the control-word component of an array register. See the section "Components of an Ivory Array Register".

sys:array-event-count *object* *Macro*

Returns the current event count for array registers. See the section "Components of an Ivory Array Register".

sys:fast-aref-1 *index array-register-control-word* *Macro*

Like **sys:%1d-aref**, but intended for use with an array register. Note that the control word (*array-register-control-word*), not the array, is used to specify the array register. This ensures that the array register is not stale; if it is, the array register will be updated.

index specifies which element in the array to return.

sys:fast-aset-1 *value index array-register-control-word* *Macro*

Like **sys:%1d-aset**, only the array register is used and no value is returned. Note that the control word (*array-register-control-word*), not the array, is used to specify the array register. This ensures that the array register is not stale; if it is, the array register will be updated.

index specifies which element in the array to set. *value* is the value to which it should be set.

sys:setup-1d-array *array* *Macro*

Creates an array register describing *array*, which must be a one-dimensional array.

Returns four values, which are the components of the array register. These components are the array, control word, base address, and array length. For detailed in-

formation on these components, see the section "Components of an Ivory Array Register".

Note that this is an Ivory subprimitive. There is a function of the same name defined on 3600-family machines, but it is incompatible with the Ivory subprimitives, and not documented for use.

sys:setup-force-1d-array *array* *Macro*

Creates an array register describing *array*, which can be any array. This function causes multidimensional arrays to be accessed as if they were one-dimensional arrays, with the order of elements in row-major order.

Returns four values, which are the components of the array register. These components are the array, control word, base address, and array length. For detailed information on these components, see the section "Components of an Ivory Array Register".

Note that this is an Ivory subprimitive. There is a function of the same name defined on 3600-family machines, but it is incompatible with the Ivory subprimitives, and not documented for use.

Examples of Using Ivory Subprimitives

Loop Unrolling Technique

Many loops in programs are characterized by a small body which is performed a large number of times. Loops that add or do **bitblt** are examples. The overhead of such a loop is more time-consuming than the body, for a single iteration. Such loops can be optimized by a technique known as *loop unrolling*; you reconstruct the code to have a larger body with fewer iterations. The goal is not to have a larger body per se, but rather to perform more than one iteration step on each trip through the body.

When using block registers, you can do loop unrolling by hand or by using **sys:unroll-block-forms** (documented later in this section). For examples of loop unrolling by hand, see the section "Examples of Vector Addition Using Block Registers". For examples of using **sys:unroll-block-forms**, see the section "Examples of Filling Arrays Using Block Registers".

When using block operations, the goal is to regularize the memory traffic into longer sequences so the memory pipeline can operate efficiently. We recommend structuring code to do a set of reads followed by a set of writes, instead of interleaving block read and write operations. Interleaving reads and writes introduces substantial slowdowns.

Recommended pattern of block operations:

```
read read read / write write write
```

Less efficient pattern:

```
read/write read/write read/write
```

On a MacIvory, when data is in the cache, the MacIvory implements the memory pipeline. However, when you access the NuBus because of a cache miss, or because of using an uncached physical address such as one might use to access a frame buffer, there can be only one cycle outstanding at a time. In this case, the memory pipelining does no good, and the right way and the wrong way to write memory processing loops perform the same.

sys:unroll-block-forms *(n blocking)* &body *body* *Macro*

Used for "loop unrolling", reconstructing Lisp code that uses a loop to have a larger body of the loop which requires fewer iterations; this is done to increase performance.

n is the number of times to perform the loop. *blocking* must be a power of 2 no greater than 128. *body* is the body of the loops.

Using this macro is equivalent to:

```
(loop repeat n doing ,@body)
```

except that the loop is unrolled. If *n* is not a multiple of *blocking*, then the body is optionally executed in groups of 1, 2, 4, ... times (up to *blocking*/2). It is then executed in a group of *blocking*, **(floor** *n block-size***)** times, something like:

```
(when (ldb-test (byte 1 0) ,n)
  ,@body)
(when (ldb-test (byte 1 1) ,n)
  ,@body
  ,@body)
...
(loop repeat (floor n ,block-size) doing
  ,@body
  ,@body
  ...
  ,@body)
```

However, since **sys:%block-read**s from a particular BAR are considerably faster when they occur consecutively, the macro rearranges the code to try to make the reads consecutive. The body should be kept relatively simple, because the macro does not do real flow-analysis on it.

Examples of Vector Addition Using Block Registers

In this example, we start with a function written in high-level Lisp, using **aref**. The function does vector addition: setting the elements of *array3* to the sum of each corresponding element in *array1* and *array2*.

```
(defun vector-add (array1 array2 array3 n-elements)
  (loop for i below n-elements do
    (setf (aref array3 i)
          (+ (aref array1 i)
             (aref array2 i)))))
```

If we know that the performance of this function has a great impact on the program as a whole, then we might try to optimize it by using block registers. (Actually, before trying block registers, we would use **sys:array-register** declarations. For further information, see the section "Array Registers".)

In the examples that follow, we gradually add techniques that increase performance.

First, we modify the function to perform the bounds checking outside the inner loop, and to abstract out the memory operation. Note that this version works only for unpacked arrays (where the **:element-type** is either **t** or **fixnum**).

```
;; Abstraction of the memory operation
(defun vector-add (array1 array2 array3 n-elements)
  (unless (and (< n-elements (length array1))
               (< n-elements (length array2))
               (< n-elements (length array3)))
    (error "Array out of bounds."))
  (%block-add (locf (aref array1 0))
              (locf (aref array2 0))
              (locf (aref array3 0))
              n-elements))
```

We now define **%block-add** to use block register operations:

```
;; Example of first cut inner loop.
(defun %block-add (source-1 source-2 destination n-elements)
  (sys:with-block-registers (1 2 3)
    (setf (sys:%block-register 1) source-1)
    (setf (sys:%block-register 2) source-2)
    (setf (sys:%block-register 3) destination)
    ;; Do most of it in blocks of eight.
    (dotimes (ignore (floor n-elements 8))
      (let* ((a1 (sys:%block-read 1))
             (a2 (sys:%block-read 1))
             (a3 (sys:%block-read 1))
             (a4 (sys:%block-read 1))
             (a5 (sys:%block-read 1))
             (a6 (sys:%block-read 1))
             (a7 (sys:%block-read 1))
```

```
            (a8 (sys:%block-read 1 :prefetch nil))
            (b1 (sys:%block-read 2))
            (b2 (sys:%block-read 2))
            (b3 (sys:%block-read 2))
            (b4 (sys:%block-read 2))
            (b5 (sys:%block-read 2))
            (b6 (sys:%block-read 2))
            (b7 (sys:%block-read 2))
            (b8 (sys:%block-read 2 :prefetch nil)))
       (sys:%block-write 3 (+ a1 b1))
       (sys:%block-write 3 (+ a2 b2))
       (sys:%block-write 3 (+ a3 b3))
       (sys:%block-write 3 (+ a4 b4))
       (sys:%block-write 3 (+ a5 b5))
       (sys:%block-write 3 (+ a6 b6))
       (sys:%block-write 3 (+ a7 b7))
       (sys:%block-write 3 (+ a8 b8))))
  ;; Do the last few words inefficiently.
  (dotimes (ignore (mod n-elements 8))
    (sys:%block-write 3
      (+ (sys:%block-read 1 :prefetch nil)
         (sys:%block-read 2 :prefetch nil))))))
```

Each time we make a change, we disassemble the code and examine it. In this case, a portion of the disassembled code suggested another optimization, to remove eight PUSH instructions:

```
43  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A1
44  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A2
45  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A3
46  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A4
47  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A5
50  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A6
51  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A7
52  %BLOCK-1-READ DATA-READ SET-CDR-NEXT INHIBIT-PREFETCH ;Creating A8
53  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B1
54  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B2
55  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B3
56  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B4
57  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B5
60  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B6
61  %BLOCK-2-READ DATA-READ SET-CDR-NEXT       ;Creating B7
62  %BLOCK-2-READ DATA-READ SET-CDR-NEXT INHIBIT-PREFETCH ;Creating B8
63  PUSH FP|14                                 ;A1
64  ADD FP|22                                  ;B1
65  %BLOCK-3-WRITE SP|POP
66  PUSH FP|15                                 ;A2
67  ADD FP|23                                  ;B2
70  %BLOCK-3-WRITE SP|POP
```

```
 71  PUSH FP|16                    ;A3
 72  ADD FP|24                     ;B3
 73  %BLOCK-3-WRITE SP|POP
 74  PUSH FP|17                    ;A4
 75  ADD FP|25                     ;B4
 76  %BLOCK-3-WRITE SP|POP
 77  PUSH FP|18                    ;A5
100  ADD FP|26                     ;B5
101  %BLOCK-3-WRITE SP|POP
102  PUSH FP|19                    ;A6
103  ADD FP|27                     ;B6
104  %BLOCK-3-WRITE SP|POP
105  PUSH FP|20                    ;A7
106  ADD FP|28                     ;B7
107  %BLOCK-3-WRITE SP|POP
110  PUSH FP|21                    ;A8
111  ADD FP|29                     ;B8
112  %BLOCK-3-WRITE SP|POP
```

The following example removes the eight PUSH instructions:

```
(defun %block-add (source-1 source-2 destination n-elements)
  (sys:with-block-registers (1 2 3)
    (setf (sys:%block-register 1) source-1)
    (setf (sys:%block-register 2) source-2)
    (setf (sys:%block-register 3) destination)
    ;; Do most of it in blocks of eight.
    (dotimes (ignore (floor n-elements 8))
      (let* ((a1 (sys:%block-read 1))
             (a2 (sys:%block-read 1))
             (a3 (sys:%block-read 1))
             (a4 (sys:%block-read 1))
             (a5 (sys:%block-read 1))
             (a6 (sys:%block-read 1))
             (a7 (sys:%block-read 1))
             (a8 (sys:%block-read 1 :prefetch nil))
             (b1 (+ (sys:%block-read 2) a1))
             (b2 (+ (sys:%block-read 2) a2))
             (b3 (+ (sys:%block-read 2) a3))
             (b4 (+ (sys:%block-read 2) a4))
             (b5 (+ (sys:%block-read 2) a5))
             (b6 (+ (sys:%block-read 2) a6))
             (b7 (+ (sys:%block-read 2) a7))
```

```
            (b8 (+ (sys:%block-read 2 :prefetch nil) a8)))
        (sys:%block-write 3 b1)
        (sys:%block-write 3 b2)
        (sys:%block-write 3 b3)
        (sys:%block-write 3 b4)
        (sys:%block-write 3 b5)
        (sys:%block-write 3 b6)
        (sys:%block-write 3 b7)
        (sys:%block-write 3 b8)))
    ;; Do the last few words inefficiently.
    (dotimes (ignore (mod n-elements 8))
      (sys:%block-write 3
        (+ (sys:%block-read 1 :prefetch nil)
           (sys:%block-read 2 :prefetch nil))))))
```

The corresponding portion of disassembled code now is:

```
 43  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A1
 44  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A2
 45  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A3
 46  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A4
 47  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A5
 50  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A6
 51  %BLOCK-1-READ DATA-READ SET-CDR-NEXT       ;Creating A7
 52  %BLOCK-1-READ DATA-READ SET-CDR-NEXT INHIBIT-PREFETCH      ;Creating A8
 53  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 54  ADD FP|14                          ;A1    Creating B1
 55  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 56  ADD FP|15                          ;A2    Creating B2
 57  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 60  ADD FP|16                          ;A3    Creating B3
 61  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 62  ADD FP|17                          ;A4    Creating B4
 63  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 64  ADD FP|18                          ;A5    Creating B5
 65  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 66  ADD FP|19                          ;A6    Creating B6
 67  %BLOCK-2-READ DATA-READ SET-CDR-NEXT
 70  ADD FP|20                          ;A7    Creating B7
 71  %BLOCK-2-READ DATA-READ SET-CDR-NEXT INHIBIT-PREFETCH
 72  ADD FP|21                          ;A8    Creating B8
 73  %BLOCK-3-WRITE FP|22               ;B1
 74  %BLOCK-3-WRITE FP|23               ;B2
 75  %BLOCK-3-WRITE FP|24               ;B3
 76  %BLOCK-3-WRITE FP|25               ;B4
 77  %BLOCK-3-WRITE FP|26               ;B5
100  %BLOCK-3-WRITE FP|27               ;B6
101  %BLOCK-3-WRITE FP|28               ;B7
102  %BLOCK-3-WRITE FP|29               ;B8
```

In the following example, we restrict the arguments and results to be fixnums, and we illustrate the use of **%block-read-alu**:

```
;; Using %BLOCK-READ-ALU
(defun %block-add (source-1 source-2 destination n-elements)
  (sys:with-block-registers (1 2 3)
    (setf (sys:%block-register 1) source-1)
    (setf (sys:%block-register 2) source-2)
    (setf (sys:%block-register 3) destination)
    (sys:set-alu-and-rotate-control
      :function (sys:%alu-function-add sys:%alu-add-op2 0 0))
    ;; Do most of it in blocks of eight.
    (dotimes (ignore (floor n-elements 8))
      (let* ((a1 (sys:%block-read 1 :fixnum-only t))
             (a2 (sys:%block-read 1 :fixnum-only t))
             (a3 (sys:%block-read 1 :fixnum-only t))
             (a4 (sys:%block-read 1 :fixnum-only t))
             (a5 (sys:%block-read 1 :fixnum-only t))
             (a6 (sys:%block-read 1 :fixnum-only t))
             (a7 (sys:%block-read 1 :fixnum-only t))
             (a8 (sys:%block-read 1 :fixnum-only t :prefetch nil)))
        (sys:%block-read-alu 2 a1)
        (sys:%block-read-alu 2 a2)
        (sys:%block-read-alu 2 a3)
        (sys:%block-read-alu 2 a4)
        (sys:%block-read-alu 2 a5)
        (sys:%block-read-alu 2 a6)
        (sys:%block-read-alu 2 a7)
        (sys:%block-read-alu 2 a8)
        (sys:%block-write 3 a1)
        (sys:%block-write 3 a2)
        (sys:%block-write 3 a3)
        (sys:%block-write 3 a4)
        (sys:%block-write 3 a5)
        (sys:%block-write 3 a6)
        (sys:%block-write 3 a7)
        (sys:%block-write 3 a8)))
    ;; Do the last few words inefficiently.
    (dotimes (ignore (mod n-elements 8))
      (let ((a (sys:%block-read 1 :fixnum-only t :prefetch nil)))
        (sys:%block-read-alu 2 a)
        (sys:%block-write 3 a)))))
```

Examples of Filling Arrays Using Block Registers

```
;;; Filling an array with constant values (the array must have
;;; one element per word).
```

```
(defun simple-fill-array (array value)
  (multiple-value-bind (array control base-address length)
      (sys:setup-1d-array array)
    (declare (ignore array))
    ;; Make sure the array contains one element per word by checking
    ;; the byte-packing, which is Log2 the elements per word.
    (unless (= (ldb sys:array-register-byte-packing control) 0)
      (error "Attempt to use SIMPLE-FILL-ARRAY on a non-32-bit array"))

    ;; The word which gets stored in the array is not necessarily the
    ;; value, so get the appropriate word for the initial value.
    ;; In other words, the value might be a character but
    ;; array-initial-word returns fixnum with the same bit pattern.
    (let ((initial-word (si:array-initial-word control value)))
      (sys:with-block-registers (1)
        (setf (sys:%block-register 1) base-address)
        ;; Write LENGTH words, doing 4 at a time
        (sys:unroll-block-forms (length 4)
          (sys:%block-write 1 initial-word))))))

;;; Filling a possibly displaced array with arbitrary byte-size.
;;; This does not worry about conformal arrays or ART-Q-LIST arrays.
(defun general-fill-array (array value)
  (multiple-value-bind (array control base-address length)
      (sys:setup-force-1d-array array)
    (declare (ignore array))
    ;; The word which gets stored in the array is not necessarily the
    ;; value, so get the appropriate word for the initial value.
    (let ((initial-word (si:array-initial-word control value)))
      (sys:with-block-registers (1)
        (setf (sys:%block-register 1) base-address)
        (let* ((byte-packing (ldb sys:array-register-byte-packing control))
               (offset (ldb sys:array-register-byte-offset control))
               (final (+ length offset))
               (final-word-offset (lsh final (- byte-packing))))
          (when (= length 0)
            (return-from general-fill-array nil))
          ;; BYTE-PACKING is the number of array elements per word.  If
          ;; ARRAY is displaced to another array and the total offset in
          ;; all the indirections between ARRAY and the actual storage
          ;; is not an integral number of words, then OFFSET is the
          ;; number of array elements units in the word at BASE-ADDRESS
          ;; which precede the part of the word where ARRAY begins.  To
          ;; make things simpler, we will think of ARRAY as being
          ;; displaced to an array with the same element size which
          ;; begins at the beginning of BASE-ADDRESS, i.e. the
          ;; displaced-index-offset is OFFSET.  Then FINAL is length of
          ;; this array, and FINAL-WORD-OFFSET is the first word
```

```
;; containing elements after ARRAY.
(cond ((≠ offset 0)
       (let ((bit-offset (rot offset (- 5 byte-packing))))
         ;; We have to read the first word, and deposit our word
         ;; into the portion used by our array.  This starts at
         ;; bit BIT-OFFSET and is some portion of the
         ;; higher-order bits above BIT-OFFSET, depending upon
         ;; the length of ARRAY.  Note that BYTE-S is one less
         ;; than the size.  The function class specifies a DPB
         ;; into OP1, ignoring the rotate-latch.
         (cond ((= final-word-offset 0)
                ;; array is only one word long, and there may
                ;; be unused elements after it in this word
                (sys:set-alu-and-rotate-control
                  :byte-r bit-offset
                  :byte-s (1- (rot length byte-packing))
                  :function (sys:%alu-function-dpb
                              sys:%alu-byte-background-op1
                              sys:%alu-byte-hold-rotate-latch))
                (sys:%block-write
                  1 (sys:%alu (sys:%block-read 1 :no-increment t)
                              initial-word))
                (return-from general-fill-array))
               (t         ;; array uses all the rest of this word
                (sys:set-alu-and-rotate-control
                  :byte-r bit-offset
                  :byte-s (- 31. bit-offset)
                  :function (sys:%alu-function-dpb
                              sys:%alu-byte-background-op1
                              sys:%alu-byte-hold-rotate-latch))
                (sys:%block-write
                  1 (sys:%alu (sys:%block-read 1 :no-increment t)
                              initial-word))))))
      ((> final-word-offset 0)      ;; array uses all of first word
       (sys:%block-write 1 initial-word)))
;; fill the fully-occupied words in the middle of the array
(sys:unroll-block-forms ((1- final-word-offset) 4)
  (sys:%block-write 1 initial-word))
(let ((final-bit-offset (- final
                           (lsh final-word-offset byte-packing))))
  (unless (zerop final-bit-offset)
    ;; We only have part of the last word.  Similar to the
    ;; treatment of the first word.
    (sys:set-alu-and-rotate-control
      :byte-r 0
      :byte-s (1- final-bit-offset)
      :function (sys:%alu-function-dpb
                  sys:%alu-byte-background-op1
```

```
                        sys:%alu-byte-hold-rotate-latch))
              (sys:%block-write 1 (sys:%alu (sys:%block-read 1 :no-increment t)
                                            initial-word)))))))))
```

Example of Testing Array Elements Using Block Registers

```
;;; Locative should point to a list of ordered fixnums, and N should
;;; be the number of fixnums.  Starting at LOCATIVE, this compares KEY
;;; with the value at that location.  If KEY ≥ VALUE, then the
;;; location and its contents are returned.  If no value in the first
;;; N locations satisfies this condition, NIL is returned.
(defun %block-search-≤ (key locative n)
  (sys:with-block-registers (1)
    ;; Set up the ALU-CONTROL register
    ;;
    ;; The ALU Function is a subtract:  OP1 is the value read from
    ;; memory, and OP2 is one more than the ones complement (i.e. the
    ;; twos complement) of KEY.  Test for KEY not less than the word
    ;; read.
    (sys:set-alu-and-rotate-control
      :condition sys:%alu-condition-signed-less-than
      :condition-sense sys:%alu-condition-sense-false
      :function
      (sys:%alu-function-add sys:%alu-add-op2 1 1))
    (setf (sys:%block-register 1) locative)
    (let ((bound (sys:%pointer-plus locative (1- n))))
      ;; BOUND is the last word in the block of data.  This is careful
      ;; not to create a pointer past the block of data.
      (sys:%block-read-test-tagbody (success :operand-2 key)
               ;; If the test passes, a branch to SUCCESS will be
               ;; taken; otherwise SYS:%BLOCK-READ-TEST will fall
               ;; through to the next instruction.
               ;; If n is not a multiple of 8, first take care of
               ;; the extras, then handle blocks of 8.
               (case (ldb (byte 3. 0.) n)
                 (0 (go do-0))
                 (1 (go do-1))
                 (2 (go do-2))
                 (3 (go do-3))
                 (4 (go do-4))
                 (5 (go do-5))
                 (6 (go do-6))
                 (otherwise (go do-7)))
           do-8 (sys:%block-read-test 1 :fixnum-only t)
           do-7 (sys:%block-read-test 1 :fixnum-only t)
           do-6 (sys:%block-read-test 1 :fixnum-only t)
           do-5 (sys:%block-read-test 1 :fixnum-only t)
```

```
do-4 (sys:%block-read-test 1 :fixnum-only t)
do-3 (sys:%block-read-test 1 :fixnum-only t)
do-2 (sys:%block-read-test 1 :fixnum-only t)
do-1 (sys:%block-read-test 1 :fixnum-only t)
do-0 (if (sys:%pointer-lessp (sys:%block-register 1) bound)
         (go do-8)        ;; not finished, do 8 more words
         (return-from %block-search-≤ nil))  ;; search failed
;; %block-read-test jumps here when the test is satisfied
success
    (return-from %block-search-≤
      (values (sys:%block-register 1)
              ;; re-read the matching location
              (sys:%block-read 1
                               :no-increment t
                               :fixnum-only t
                               :prefetch nil)))))))
```

Stack Groups

A *stack group* (abbreviated "SG") is a type of Lisp object useful for implementation of certain advanced control structures such as coroutines and generators. Processes, which are a kind of coroutine, are built on top of stack groups. (See the section "Using Processes for Computations".) A stack group represents a computation and its internal state, including the Lisp stack.

At any time, the computation being performed by a Symbolics computer is associated with one stack group, called the *current* or *running* stack group. The operation of making some stack group be the current stack group is called a *resumption* or a *stack group switch*; the previously running stack group is said to have *resumed* the new stack group. The *resume* operation has two parts: first, the state of the running computation is saved away inside the current stack group, and secondly the state saved in the new stack group is restored, and the new stack group is made current. Then the computation of the new stack group resumes its course.

The stack group itself holds a great deal of state information. It contains the *control stack*. The control stack is what you are shown by the Debugger's backtracing commands (`c-B`, `m-B`, and `c-m-B`); it remembers the function that is running, its caller, its caller's caller, and so on, and the point of execution of each function (the "return addresses" of each function). A stack group contains the *binding* (environment) *stack*. This contains all of the values saved by binding of special variables. A stack group also contains structures allocated on the *data stack* by such operations as **sys:make-stack-array**. See the special form **sys:make-stack-array**. The name "stack group" derives from the existence of these stacks. Finally, the stack group contains various internal state information (contents of machine registers and so on).

When the state of the current stack group is saved away, all of its bindings are undone, and when the state is restored, the bindings are put back. Note that although bindings are temporarily undone, unwind-protect handlers are *not* run by a stack-group switch.

Each stack group is a separate environment for purposes of function calling, throwing, dynamic variable binding, and condition signalling. All stack groups run in the same address space, thus they share the same Lisp data and the same global (not bound) variables.

When a new stack group is created, it is empty: it doesn't contain the state of any computation, so it cannot be resumed. In order to get things going, the stack group must be set to an initial state. This is done by *presetting* the stack group. To preset a stack group, you supply a function and a set of arguments. The stack group is placed in such a state that when it is first resumed, this function calls those arguments. The function is called the *initial function* of the stack group.

Resuming of Stack Groups

Stack groups *resume* each other. When one stack group resumes a second stack group, the current state of Lisp execution is saved away in the first stack group, and is restored from the second stack group. Resuming is also called *switching* stack groups.

At any time, there is one stack group associated with the current computation; it is called the *current stack group*. The computations associated with other stack groups have their states saved away in memory, and they are not computing. So the only stack group that can do anything at all, in particular resuming other stack groups, is the current one.

You can look at things from the point of view of one computation. Suppose it is running along, and it resumes some stack group. Its state is saved away in the current stack group, and the computation associated with the one it called starts up. The original computation lies dormant in the original stack group, while other computations go around resuming each other, until finally the original stack group is resumed by someone. Then the computation is restored from the stack group and runs again.

There are several ways that the current stack group can resume other stack groups. This section describes all of them.

Associated with each stack group is a *resumer*. The resumer is **nil** or another stack group. Some forms of resuming examine and alter the resumer of some stack groups.

Resuming has another ability: it can transmit a Lisp object from the old stack group to the new stack group. Each stack group specifies a value to transmit whenever it resumes another stack group; whenever a stack group is resumed, it receives a value.

In the descriptions below, let *c* stand for the current stack group, *s* stand for some other stack group, and *x* stand for any arbitrary Lisp object.

Stack groups can be used as functions. They accept one argument. If *c* calls *s* as a function with one argument *x*, then *s* is resumed, and the object transmitted is *x*. When *c* is resumed (usually — but not necessarily — by *s*), the object transmitted by that resumption is returned as the value of the call to *s*. This is one of the

simple ways to resume a stack group: call it as a function. The value you transmit is the argument to the function, and the value you receive is the value returned from the function. Furthermore, this form of resuming sets *s*'s resumer to be *c*.

Another way to resume a stack group is to use **stack-group-return**. Rather than allowing you to specify which stack group to resume, this function always resumes the resumer of the current stack group. Thus, this is a good way to resume whoever it was who resumed *you*, assuming it was done by function-calling. Note that you cannot use **stack-group-return** if the current stack group was resumed with **stack-group-resume**. **stack-group-return** takes one argument, which is the object to transmit. It returns when someone resumes the current stack group, and returns one value, the object that was transmitted by that resumption. **stack-group-return** does not affect the resumer of any stack group.

The most fundamental way to do resuming is with **stack-group-resume**, which takes two arguments: the stack group, and a value to transmit. It returns when someone resumes the current stack group, returning the value that was transmitted by that resumption, and does not affect any stack group's resumer.

If the initial function of *c* attempts to return a value *x*, the regular kind of Lisp function return cannot take place, since the function did not have any caller (it got there when the stack group was initialized). So instead of normal function returning, a "stack group return" happens. *c*'s resumer is resumed, and the value transmitted is *x*. *c* is left in a state ("exhausted") from which it cannot be resumed again; any attempt to resume it signals an error. Presetting it makes it work again.

Those are the "voluntary" forms of stack group switch; a resumption happens because the computation said it should. There are also two "involuntary" forms, in which another stack group is resumed without the explicit request of the running program.

When certain events occur, such as a 1/60th of a second clock tick, a *sequence break* occurs. Sequence breaks are handled by system code, operating below the level of stack groups. After a certain amount of time has elapsed (typically 1/10th of a second), a sequence break causes the occurrence of a *preemption*. A preemption forces the current stack group to resume a special stack group called the *scheduler*. (See the section "The Scheduler".) The scheduler implements processes by resuming, one after another, the stack group of each process that is ready to run.

Stack Group Functions

make-stack-group *name* &rest *options* &key *(:sg-area* **sys:safeguarded-objects-area***) (:regular-pdl-area* **sys:stack-area***) (:special-pdl-area* **sys:stack-area***) (:regular-pdl-size* **12288***) (:special-pdl-size* **2048***) :absolute-control-stack-limit :absolute-binding-stack-limit (:safe* **1***) :allow-unknown-keywords* &allow-other-keys *Function*

Creates and returns a new stack group. *name* can be any symbol or string; it is used in the stack group's printed representation. *options* is a list of alternating keywords and values. The options are not too useful; most calls to **make-stack-group** do not need any options at all. The useful options are:

:regular-pdl-size
How big to make the stack group's control stack. The default is large enough for most purposes.

:special-pdl-size
How big to make the stack group's special binding pdl. The default is large enough for most purposes.

:safe If this flag is 1 (the default), a strict call-return discipline among stack groups is enforced. If 0, no restriction on stack-group switching is imposed.

stack-group-preset *sg function* &rest *args* *Function*

Sets up *sg* so that when it is resumed, *function* is applied to *args* within the stack group. Both stacks are made empty; all saved state in the stack group is destroyed. **stack-group-preset** is typically used to initialize a stack group just after it is made, but it can be done to any stack group at any time. Doing this to a stack group that is not exhausted destroys its present state without properly cleaning up by running **unwind-protect**s.

stack-group-resume *sg value* *Function*

Resumes *sg*, transmitting the value *value*. No stack group's resumer is affected.

stack-group-return *value* *Function*

Resumes the current stack group's resumer, transmitting the value *value*. No stack group's resumer is affected.

sys:sg-previous-stack-group

stack-group *Function*

Returns the resumer of *stack-group*.

symbol-value-in-stack-group

sym sg &optional *frame as-if-current* *Function*

Evaluates the variable *sym* in the binding environment of *sg*. If *sg* is the current stack group, this is just **symbol-value**. Otherwise it looks inside *sg* to see if *sym* is bound there; if so, the binding is returned; if not, the global value is returned. If *frame* is specified, the value visible in that frame is returned. If *as-if-current* is

non-**nil**, a location is returned indicating where the value would be if the specified stack group were running. The value, though, is the current one, not the one stored in that location.

zl:symeval-in-stack-group

sym sg &optional *frame as-if-current* *Function*

In your new programs, we recommend that you use the function **symbol-value-in-stack-group,** which is the Symbolics Common Lisp equivalent of the function **zl:symeval-in-stack-group**.

Evaluates the variable *sym* in the binding environment of *sg*. If *sg* is the current stack group, this is just **zl:symeval**. Otherwise this function is the same as **symbol-value-in-stack-group**.

A large number of functions in the **sys:** and **dbg:** packages exist for manipulating the internal details of stack groups. These are not documented here as they are not necessary for most users or even system programmers to know about.

Input/Output in Stack Groups

Because each stack group has its own set of dynamic bindings, a stack group does not inherit its creator's value of ***terminal-io***, nor its caller's, unless you make special provision for this. See the variable ***terminal-io***. The ***terminal-io*** a stack group gets by default is a "background" stream that does not normally expect to be used. If it is used, it turns into a "background window" that requests the user's attention. Usually this is because an error printout is trying to be printed on the stream.

If you write a program that uses multiple stack groups, and you want them all to do input and output to the terminal, you should pass the value of ***terminal-io*** to the top-level function of each stack group as part of the **stack-group-preset**, and that function should bind the variable ***terminal-io***.

Another technique is to use a dynamic closure as the top-level function of a stack group. This closure can bind ***terminal-io*** and any other variables that are desired to be shared between the stack group and its creator. Note that a dynamic enclosure must be used, not a lexical enclosure. Lexical closures do not close over **special** variables. See the function **make-dynamic-closure**. See the special form **special**.

An Example of Stack Groups

The canonical coroutine example is the so-called samefringe problem: Given two trees, determine whether they contain the same atoms in the same order, ignoring parenthesis structure. In other words, given two binary trees built out of conses, determine whether the sequence of atoms on the fringes of the trees is the same, ignoring differences in the arrangement of the internal skeletons of the two trees. Following the usual rule for trees, **nil** in the cdr of a cons is to be ignored.

One way of solving this problem is to use *generator* coroutines. We make a generator for each tree. Each time the generator is called it returns the next element of the fringe of its tree. After the generator has examined the entire tree, it returns a special "exhausted" flag. The generator is most naturally written as a recursive function. The use of coroutines, that is, stack groups, allows the two generators to recurse separately on two different control stacks without having to coordinate with each other.

The program is very simple. Constructing it in the usual bottom-up style, we first write a recursive function that takes a tree and **stack-group-return**s each element of its fringe. The **stack-group-return** is how the generator coroutine delivers its output. We could easily test this function by replacing **stack-group-return** with **print** and trying it on some examples.

```
(defun fringe (tree)
  (cond ((atom tree) (stack-group-return tree))
        (t (fringe (car tree))
           (if (not (null (cdr tree)))
               (fringe (cdr tree))))))
```

Now we package this function inside another, which takes care of returning the special "exhausted" flag.

```
(defun fringe1 (tree exhausted)
  (fringe tree)
  exhausted)
```

The **samefringe** function takes the two trees as arguments and returns **t** or **nil**. It creates two stack groups to act as the two generator coroutines, presets them to run the **fringe1** function, then goes into a loop comparing the two fringes. The value is **nil** if a difference is discovered, or **t** if they are still the same when the end is reached.

```
(defun samefringe (tree1 tree2)
  (let ((sg1 (make-stack-group "samefringe1"))
        (sg2 (make-stack-group "samefringe2"))
        (exhausted (ncons nil)))        ;unique item
    (stack-group-preset sg1 #'fringe1 tree1 exhausted)
    (stack-group-preset sg2 #'fringe1 tree2 exhausted)
    (do ((v1) (v2)) (nil)
      (setq v1 (funcall sg1 nil)
            v2 (funcall sg2 nil))
      (cond ((neq v1 v2) (return nil))
            ((eq v1 exhausted) (return t))))))
```

Now we test it on a couple of examples.

```
(samefringe '(a b c) '(a (b c))) => t
(samefringe '(a b c) '(a b c d)) => nil
```

The problem with this is that a stack group is quite a large object, and we make two of them every time we compare two fringes. This is a lot of unnecessary overhead. It can easily be eliminated with a modest amount of explicit storage allocation, using the resource facility. See the function **defresource**. While we're at it,

we can avoid making the exhausted flag fresh each time; its only important property is that it not be an atom.

```
(defvar *exhausted-flag* (ncons nil))

(defresource samefringe-coroutine ()
   :constructor (make-stack-group "for-samefringe"))

(defun samefringe (tree1 tree2)
  (using-resource (sg1 samefringe-coroutine)
    (using-resource (sg2 samefringe-coroutine)
      (stack-group-preset sg1 #'fringe1 tree1 *exhausted-flag*)
      (stack-group-preset sg2 #'fringe1 tree2 *exhausted-flag*)
      (do ((v1) (v2)) (nil)
        (setq v1 (funcall sg1 nil)
              v2 (funcall sg2 nil))
        (cond ((neq v1 v2) (return nil))
              ((eq v1 *exhausted-flag*) (return t)))))))
```

Now we can compare the fringes of two trees with no allocation of memory whatsoever.

Allocation on the Stack

Consing Lists on the Control Stack

with-stack-list and **with-stack-list*** cons lists on the control stack so that when you are finished, the lists are popped off without leaving any physical garbage. This is essentially giving you access to the mechanism that **&rest** arguments use. Because these are on the control stack, you cannot return the lists that are made, use **rplacd** with them, or place references to them in permanent data structures. The special form **sys:with-stack-array** is similar, but it makes arrays on the data stack instead of lists.

The macros **stack-let** and **stack-let*** provide an alternative to **with-stack-list** and **with-stack-list*** for consing lists on the control stack. They are especially useful for building nested list structures on the stack.

with-stack-list *(var* &rest *elements)* &body *body* *Function*

Binds a variable to a list and evaluates some forms in the context of that binding. It is like **let** (in that it binds a variable), except that it conses the list on the stack.

```
(scl:with-stack-list (var element1 element2...elementn)
  body)
```

is like

```
(let ((var (list element1 element2...elementn)))
  body)
```

If you want these values to be returned, or to be made part of permanent storage, then it is necessary to copy them with the **sys:copy-if-necessary** function. This function checks whether an object is in temporary storage or on a stack, and moves it to permanent storage if it is. See the function **sys:copy-if-necessary**.

with-stack-list* *(var* &rest *elements)* &body *body* *Function*

Binds a variable to a list and evaluates some forms in the context of that binding. It is like **let** (in that it binds a variable), except that **with-stack-list*** conses the list on the stack. **with-stack-list*** simulates **list*** instead of **list**. (See the function **list***.)

```
(scl:with-stack-list* (var element1 element2...elementn)
  body)
```

is like

```
(let ((var (list* element1 element2...elementn)))
  body)
```

stack-let *bindings* &body *body* *Function*

Provides an alternative syntax for constructing lists on the control stack. It uses the same syntax (and very similar semantics) as **let**. For example, the form:

```
(stack-let ((a (list x y z))) body)
```

expands into:

```
(scl:with-stack-list (a x y z) body)
```

This syntax is convenient for complex expressions involving nested lists, such as:

```
(stack-let ((a `((:foo ,foo) (:bar ,bar)))) body)
```

which expands into three nested **with-stack-list** forms. If an expression in a **stack-let** clause is of the form:

```
(list (reverse (list ...)))
```

only the outermost list is constructed on the stack. No codewalking is performed.

It also works for arrays and instances. If the form is not recognized, it just allocates data the ordinary way.

The form

```
(stack-let ((a (list x y z))) body)
```

is similar to

```
((lambda (&rest a)
    (declare (sys:downward-rest-argument))
     body) x y z)
```

stack-let* *bindings* &body *body* *Function*

Provides an alternative syntax for constructing lists on the control stack. It is similar to **stack-let**, but it uses the same syntax and similar semantics as **let***.

The Data Stack

sys:with-stack-array *(var length* &key *:type :element-type :initial-element :initial-contents :displaced-to :displaced-index-offset :displaced-conformally :leader-list :leader-length :named-structure-symbol :initial-value :fill-pointer)* &body *body*

Special Form

Like **with-stack-list**, but makes an array. The array has a dynamic lifetime and becomes "conceptual garbage" when the form is exited, just as with **with-stack-list**. ("Conceptual garbage" means objects that are no longer in use by the program and are thus fair game for the garbage collector. "Physical garbage," in contrast, is storage that is occupied by conceptual garbage and has not yet been reclaimed for productive use.) If you have an array that becomes conceptual garbage when control exits a form, that array is a candidate for implementation by **sys:with-stack-array** so that there will not be any physical garbage.

The array is created on the data stack, which is part of a stack group. Only arrays can be allocated on the data stack.

The keyword options to **sys:make-stack-array** include options that are accepted by **make-array** and **zl:make-array**. For information on these options: See the section "Keyword Options for **make-array**".

This recognizes various special case combinations of **make-array** keywords and calls fast specialized runtime routines. It works especially well with one-dimensional indirect arrays.

Here is an example of the use of **sys:with-stack-array**.

```
(sys:with-stack-array (a n :element-type 'cl:string-char
                           :initial-element #\space) ...)
```

More information is available about stack arrays and the data stack. See the special form **sys:make-stack-array**. See the function **sys:with-data-stack**.

For rasters, use **sys:with-raster-stack-array** instead: See the function **sys:with-raster-stack-array**.

sys:with-raster-stack-array *(var width height* &key *:type :element-type :initial-element :initial-contents :displaced-to :displaced-index-offset :displaced-conformally*

:leader-list :leader-length :named-structure-symbol :initial-value :fill-pointer) &body *body* *Function*

Provides the same functionality as does **sys:with-stack-array**, but it is used for rasters. Note that **sys:with-raster-stack-array** has *width* and *height* arguments instead of the *length* argument of **sys:with-stack-array**.

See the special form **sys:with-stack-array**.

The keyword options to **sys:make-stack-array** include options that are accepted by **make-array** and **zl:make-array**. For information on these options: See the section "Keyword Options for **make-array**".

In the following example, note that in the Genera row-major implementation the height is the first dimension and width is the second:

```
(scl:make-raster-array 2 7 :element-type 'boolean)
       => #<ART-BOOLEAN-7-2 61047172>

(sys:with-raster-stack-array (array 2 7 :element-type 'boolean)
  (print array)
  nil)
 => #<ART-BOOLEAN-7-2 21400001>
    NIL
```

sys:with-data-stack &body *body* *Function*

Cleans up the data stack when the body is exited. You sometimes want to optimize for extra speed by putting a **sys:with-data-stack** primitive special form around a piece of code that calls **sys:make-stack-array** multiple times, perhaps even inside a loop that is known not to be executed more than a few times. This can be more efficient than doing **sys:with-stack-array** multiple times.

sys:make-stack-array *dimensions* &rest *keywords* *Special Form*

A special version of **make-array** and **zl:make-array** that allocates on the data stack. You should call this only when dynamically inside a **sys:with-data-stack**. This is actually a macro that expands into a call to an appropriate routine, to allocate the desired kind of array on the data stack.

The keyword options to **sys:make-stack-array** include all options that are accepted by **make-array** and **zl:make-array**. For information on these options: See the section "Keyword Options for **make-array**".

For rasters, use **sys:with-raster-stack-array** instead: See the function **sys:with-raster-stack-array**.

Currently, you cannot make anything but arrays and rasters on the data stack.

sys:make-raster-stack-array *width height* &key *keywords* *Function*

Provides the same functionality as **sys:make-stack-array**, but it is used for rasters. Note that **sys:with-raster-stack-array** has *width* and *height* arguments instead of the *dimensions* argument of **sys:make-stack-array**.

See the special form **sys:make-stack-array**.

The keyword options to **sys:make-raster-stack-array** include all options that are accepted by **make-array** and **zl:make-array**. For information on these options: See the section "Keyword Options for **make-array**".

In the following example, note that in the Genera row-major implementation the height is the first dimension and width is the second:

```
(sys:with-data-stack
  (let ((array (sys:make-raster-stack-array
                 3 5 :element-type 'character)))
    (print array))
  nil)
 => #<ART-FAT-STRING-5-3 21400001>
```