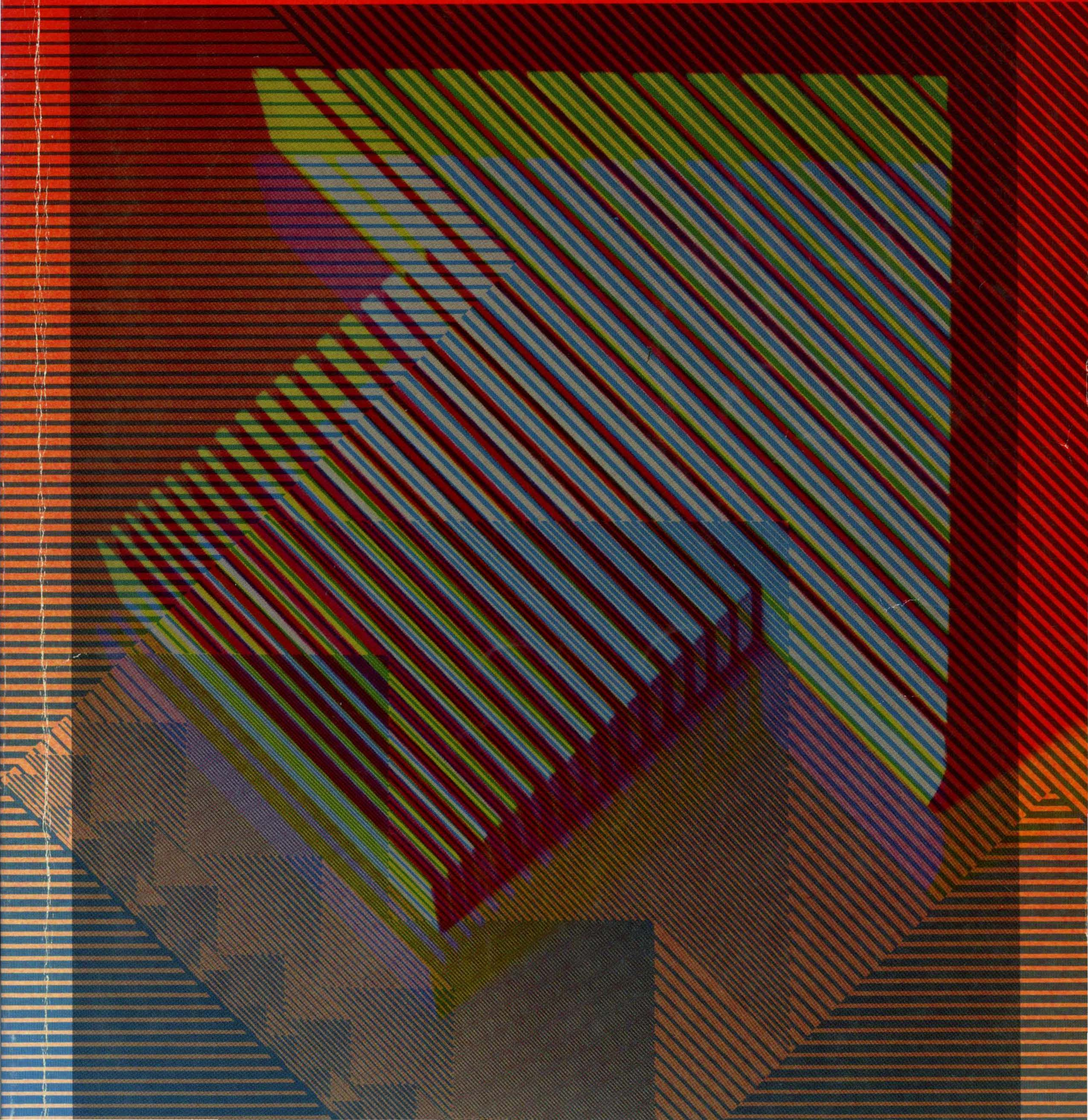
8 Internals, Processes, and Storage Management

symbolics



8 Internals, Processes, and Storage Management

symbolics

Cambridge, Massachusetts

Internals, Processes, and Storage Management

 # 996085

March 1985

This document corresponds to Release 6.0 and later releases.

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.

Text typography: Century Schoolbook and Helvetica produced on Symbolics 3600-family computers from Bitstream, Inc., outlines; text masters printed on Symbolics LGP-1 Laser Graphics Printers.
Cover design: Schafer/LaCasse
Cover printer: W.E. Andrews Co., Inc.
Text printer: ZBR Publications, Inc.

Printed in the USA.

Printing year and number: 87 86 85 9 8 7 6 5 4 3 2 1

Table of Contents

List of Tables

PART I.

Internals

1. Stack Groups

A *stack group* (usually 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 "Processes", page 73.) A stack group represents a computation and its internal state, including the Lisp stack.

At any time, the computation being performed by the Lisp Machine 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 also contains the binding (environment) stack. This contains all of the values saved by **lambda**-binding of special variables. 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. (See the special form **let-globally** in *Reference Guide to Symbolics-Lisp*.)

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 lambda-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.

1.1 Resuming of Stack Groups

The interesting thing that happens to stack groups is that they 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 into 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 gets to run 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. **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, typically a one-second clock tick, a *sequence break* occurs. This forces the current stack group to resume a special stack group called the *scheduler*. (See the section "The Scheduler", page 77.) The scheduler implements processes by resuming, one after another, the stack group of each process that is ready to run.

sys:sg-previous-stack-group *stack-group* *Function*

Returns the resumer of *stack-group*.

1.2 Stack Group Functions

make-stack-group *name* &rest *options* *Function*

This creates and returns a new stack group. *name* may 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 options are:

:sg-area

The area in which to create the stack group structure itself. Defaults to the default area (the value of **permanent-storage-area**).

:regular-pdl-area

The area in which to create the stack group's control stack. The default is **stack-area**.

:special-pdl-area

The area in which to create the binding (environment) stack. Defaults to the default area (the value of **stack-area**).

: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*

This 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 may 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-protects**.

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.

symeval-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 **symeval**. 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 the variable has no value this gets an unbound-variable error. 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.

There are a large number of functions in the **sys:** and **dbg:** packages 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.

1.3 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** in *Reference Guide to Streams, Files, and I/O*. 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.

1.4 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. A better way of saying this is, 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-returns** 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 special form **defresource**, page 132. 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.

2. Subprimitives

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. Subprimitives are normally hand-coded in microcode.

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.

Additional information can be found in the system definition files:

sys: l-sys; sysdef.lisp
: Data structure definitions

sys: l-sys; sysdfl.lisp
: Communication areas, escape routines

sys: l-sys; opdef.lisp
: Instruction set definition

2.1 Data Type Subprimitives

data-type *arg* *Function*

data-type returns a symbol that is the name for the internal data type of the "pointer" that represents *arg*. 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, **dtp-extended-number** is the symbol that **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 **typep** function is a higher-level primitive that is more useful in most cases; normal programs should always use **typep** rather than **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.

dtp-symbol	The object is a symbol.
dtp-nil	**nil** has a data type of **dtp-nil**, rather than **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.
dtp-fix	The object is a fixnum; the numeric value is contained in the address field of the pointer.
dtp-float	The object is a single-precision floating-point number.
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.
dtp-list	The object is a cons.
dtp-locative	The object is a locative pointer.
dtp-array	The object is an array.
dtp-compiled-function	The object is a compiled function.
dtp-closure	The object is a dynamic closure. See the section "Closures" in *Reference Guide to Symbolics-Lisp.*
dtp-lexical-closure	The object is a lexical closure. See the section "Closures" in *Reference Guide to Symbolics-Lisp.*
dtp-instance	The object is an instance of a flavor, that is, an "active object". See the section "Flavors" in *Reference Guide to Symbolics-Lisp.*
dtp-null	Nothing to do with **nil**. This is used in unbound value and function cells.

dtp-external-value-cell-pointer
An "invisible pointer" used for external value cells, which are part of the closure mechanism. See the section "Closures" in *Reference Guide to Symbolics-Lisp*.

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.

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.

dtp-one-q-forward An "invisible pointer" used to indicate that the single cell containing it has been moved elsewhere.

dtp-gc-forward This is used by the garbage collector to flag the obsolete copy of an object; it points to the new copy.

dtp-odd-pc,dtp-even-pc The object is a program counter and points to macroinstructions.

dtp-header-i,dtp-header-p
Internal markers in storage, found at the base of the storage of structures.

sys:*data-types* *Variable*
The value of **sys*data-types*** is a list of all of the symbolic names for data types described above under **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. This "function" is actually an array.

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*
This function sets the data type field to convert a fixnum to a flonum. It is not the function **float**, but instead provides direct access to the internal bit representation of single-precision floating-point numbers.

sys:%fixnum *number* *Function*

This function sets the data type field to convert a flonum to a fixnum. It is not the function **fix**, but instead provides direct access to the internal bit representation of single-precision floating-point numbers.

2.2 Forwarding

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 there are various rules about where they can or cannot 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 **dtp-one-q-forward**. It is used to forward a single word of memory to someplace else. The invisible pointers with data types **dtp-header-forward** and **dtp-element-forward** are used for moving whole Lisp objects (such as cons cells or arrays) somewhere else. The **dtp-external-value-cell-pointer** is very similar to the **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 **dtp-external-value-cell-pointer** that points to some other word (the external value cell), then **symeval** 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 **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.

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 **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*

This causes references to *old* to actually reference *new*, by storing invisible pointers in *old*. It returns *old*.

An example of the use of **structure-forward** is **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 *object* *Function*

Normally returns *object*, but if *object* has been **structure-forwarded**, 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*

This alters *from-symbol* so that it always has the same value as *to-symbol*, by sharing its value cell. A **dtp-one-q-forward** invisible pointer is stored into *from-symbol*'s value cell.

To forward one arbitrary cell to another (rather than specifically one value cell to another), given two locatives do

```
(%p-store-tag-and-pointer locative1 dtp-one-q-forward locative2)
```

follow-cell-forwarding *loc evcp-p* *Function*

loc is a locative to a cell. Normally *loc* is returned, 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.

2.3 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*

sys:%pointerp returns **t** when *object* has an address (as opposed to being an immediate object).

sys:%pointer-type-p *data-type-number* *Function*

sys:%pointer-type-p 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 **%data-type** or **%p-data-type.**

For example:

```
(sys:%pointer-type-p (%data-type 'symbol))
```

sys:%pointer-lessp *p1 p2* *Function*

Compares two addresses.

%data-type *x* *Function*

Returns the data-type field of *x*, as a fixnum.

%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.

%make-pointer *data-type pointer* *Function*

This 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 **dtp-null**, invisible pointers, etc.

%make-pointer-offset *new-dtp pointer offset* *Function*

This 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 **%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 **%p-structure-offset** may be more appropriate.

%pointer-difference *pointer-1 pointer-2* *Function*

Returns a fixnum that is *pointer-1* minus *pointer-2*. 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.

2.4 Analyzing Structures

%find-structure-header *pointer* *Function*

This subprimitive 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.

%find-structure-leader *pointer* *Function*

The result of **%find-structure-leader** is always the lowest address in the structure (as a locative).

%structure-total-size *pointer* *Function*

Returns the total number of words occupied by the representation of the indicated object.

%find-structure-extent *pointer* *Function*

This is roughly a combination of **%find-structure-header**, **%find-structure-leader**, and **%structure-total-size**.

%find-structure-extent 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 **%find-structure-leader**, but **%find-structure-extent** always returns a locative.

3. The total number of words occupied by the object (the same thing **%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)
      (%find-structure-extent obj)
    (page-in-words leader size
                   hang-p normalize-p)))
```

2.5 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 **%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, **%store-conditional**, is still accepted, but should not be used in new programs.

2.6 Accessing Arrays Specially

sys:array-column-span *array* *Function*

sys:array-column-span, given a two-dimensional **array**, returns the number of array elements spanned by one of its columns. Normally, this is just equal to the length of a column (that is,. the number of rows), but for conformally displaced arrays, the length and the span are not equal. This function is primarily for users of **sys:%1d-aref** and the **sys:array-register-1d** declaration who need to perform their own subscript calculations and do special loop optimizations.

A column is the sequence of elements that have the same value in the second subscript and varying values in the first subscript. Currently, with column-major order, the screen displays a column horizontally.

2.7 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, intended to be used with subprimitives such as **%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 **%%q-fixnum** and **%%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 file sys:l-sys;sysdef.lisp.

%%q-cdr-code *Variable*

The field of a memory word that contains the cdr-code. See the section "Cdr-coding" in *Reference Guide to Symbolics-Lisp.*

%%q-data-type *Variable*

The field of a memory word that contains the data type code. See the section "Data Types" in *Reference Guide to Symbolics-Lisp.*

%%q-pointer *Variable*

The field of a memory that contains the pointer address, or immediate data.

%%q-pointer-within-page *Variable*

The field of a memory word that contains the part of the address that lies within a single page.

%%q-typed-pointer *Variable*

The concatenation of the **%%q-data-type** and **%%q-pointer** fields.

%%q-all-but-typed-pointer *Variable*

The field of a memory word that contains the tag field **%%q-cdr-code**.

%%q-all-but-pointer *Variable*

The concatenation of all fields of a memory word except for **%%q-pointer**.

%%q-all-but-cdr-code *Variable*

The concatenation of all fields of a memory word except for **%%q-cdr-code**.

cdr-normal *Variable*

The value of this variable is one of the numeric values that go in the cdr-code field of a memory word. See the section "Cdr-coding" in *Reference Guide to Symbolics-Lisp.*

cdr-next *Variable*

The value of this variable is one of the numeric values that go in the cdr-code field of a memory word. See the section "Cdr-coding" in *Reference Guide to Symbolics-Lisp.*

cdr-nil *Variable*

The value of this variable is one of the numeric values that go in the cdr-code field of a memory word. See the section "Cdr-coding" in *Reference Guide to Symbolics-Lisp.*

2.8 Special Memory Referencing

sys:%p-structure-offset *x offset* *Function*

Does **follow-structure-forwarding** on *x*, then **%make-pointer-offset dtp-locative** of that and *offset*. This operation captures the inherent primitive underlying **%p-ldb-offset** and the like.

%p-contents-offset *pointer offset* *Function*

This 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.

There is no **%p-contents**, since **location-contents** performs that operation.

%p-contents-as-locative *x* *Function*

Given a pointer to a memory location containing a pointer that is not allowed to be "in the machine" (typically an invisible pointer) this function returns the contents of the location as a **dtp-locative**. It changes the disallowed data type to **dtp-locative** so that you can safely look at it and see what it points to.

%p-contents-as-locative-offset *pointer offset* *Function*

This 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 **dtp-locative** in case it was a type that is not allowed to be "in the machine" (typically an invisible pointer).

%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.

%p-store-contents-offset *value pointer offset* *Function*

This 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.

%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*. This is a good way to store a forwarding pointer from one structure to another (for example).

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, because **%p-store-tag-and-pointer** cannot be reasonably used to do so, because the tag overlaps the value.

sys:%p-store-cdr-type-and-pointer *pointer cdr-field type-field pointer-field* *Function*
This is a more general form of **%p-store-tag-and-pointer**.

%p-ldb *ppss pointer* *Function*
This is like **ldb** but gets a byte specified by *ppss* 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. The result returned is always a positive fixnum. The size of *ppss* must be 31 or less, and the sum of the size and position must be less than or equal to 36.

%p-ldb-offset *ppss pointer offset* *Function*
This 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 31 or less, and the sum of the size and position must be less than or equal to 36.

%p-dpb *value ppss pointer* *Function*
The *value*, a fixnum, is stored into the byte selected by *ppss* 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 *ppss* must be 31 or less, and the sum of the size and position must be less than or equal to 36.

%p-dpb-offset *value ppss pointer offset* *Function*
This checks the cell pointed to by *pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, the *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 31 or less, and the sum of the size and position must be less than or equal to 36.

%p-pointer *pointer* *Function*
Extracts the pointer field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%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.

%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.

%p-store-pointer *pointer value* *Function*

Clobbers the pointer field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-data-type *pointer value* *Function*

Clobbers the data-type field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-cdr-code *pointer value* *Function*

Clobbers the cdr-code field of the location addressed by *pointer* to *value*, and returns *value*.

%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 only works 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 the object should be 0 if the object is not a fixnum. The increment is added to the address field (**%%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 the object should be 0 if the object is not a fixnum. 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:%unsynchronized-device-read *address* *Function*

Reads registers from the revision 2 I/O board. It allows data that are not properly synchronized to the Lbus clock to be read without causing a parity error.

2.9 Lambda-binding Subprimitive

bind *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 only works 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 **bind** instruction". The preferred higher-level primitives that turn into this are **let-if**, **progv**, **progw**, and **letf**.

2.10 Function-calling Subprimitives

Except for **%push** and **%pop**, the subprimitives for calling with a run-time-variable number of arguments, without consing a list, are the **%start-function-call** and **%finish-function-call** special forms.

%start-function-call and **%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.
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 **%push** has to be done.

lexpr — True if the last **%push** is a list of arguments rather than a single argument; false in the normal case. Not evaluated.

Follow these steps:

1. Do a **%start-function-call.**
2. Do a **%push** on each argument.
3. Do a **%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 **%start** and the **%finish**. 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 **%start** and the **%finish**, because they will get in the way of the **%push** subprimitives. Thus, the following *will not work*:

```
(%start-function-call ...)
(dolist (x l) (%push x))
(%finish-function-call ...)
```

Instead, write:

```
(let ((x l))
  (%start-function-call ...)
  (do () ((null x)) (%push (pop x)))
  (%finish-function-call ...))
```

%push *value* *Function*

Pushes *value* onto the stack. Use this to push the arguments.

%pop *Function*

Pops the top value off of the stack and returns it as its value.

2.11 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 **page-in-area** and **page-in-region.**

sys:page-in-structure *obj* &optional (*hang-p* **si:*default-page-in-hang-p***) (*normalize-p* **si:*default-page-in-normalize-p***) *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 **%find-structure-extent** subprimitive. 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 may still be in progress. 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); its default value, **si:*default-page-in-normalize-p***, is **t** by default.

sys:page-in-array *array* &optional *from to* (*hang-p* **si:*default-page-in-hang-p***) (*normalize-p* **si:*default-page-in-normalize-p***) *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.

sys:page-in-words *address n-words* &optional (*hang-p* **si:*default-page-in-hang-p***) (*normalize-p* **si:*default-page-in-normalize-p***) *Function*

Any pages in the range of address space starting at *address* and continuing for *n-words* that have been swapped out to disk are read in with as few disk operations as possible.

sys:page-in-area *area* &optional (*hang-p* **si:*default-page-in-hang-p***) (*normalize-p* **si:*default-page-in-normalize-p***) *Function*

All swapped-out pages of the specified area are brought into main memory.

sys:page-in-region *region* &optional (*hang-p* **si:*default-page-in-hang-p***) (*normalize-p* **si:*default-page-in-normalize-p***) *Function*

All swapped-out pages of the specified region are brought into main memory.

sys:page-out-structure *obj* &optional (*hang-p* **si:*default-page-out-hang-p***) *Function*

Similar to **sys:page-in-structure**, but take 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. **hang-p** specifies whether the function waits for the disk writes to complete before returning; its default value, **si:*default-page-out-hang-p***, is **nil** by default.

sys:page-out-array *array* &optional *from to* (*hang-p* **si:*default-page-out-hang-p***) *Function*

Similar to **sys:page-in-array**, but take 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-words *address n-words* &optional (*hang-p* **si:*default-page-out-hang-p***) *Function*

Similar to **sys:page-in-words**, but take 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* &optional (*hang-p* **si:*default-page-out-hang-p***) *Function*

Similar to **sys:page-in-area**, but take 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* &optional (*hang-p* **si:*default-page-out-hang-p***) *Function*

Similar to **sys:page-in-region**, but take 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.

2.12 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 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 *(variable* &rest *list-elements) body ...* *Special Form*

with-stack-list is used to bind a variable to a list and evaluate 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.

```
(with-stack-list (var element1 element2...elementn)
  body)
```

is like

```
(let ((var (list element1 element2...elementn)))
  body)
```

with-stack-list* *(variable* &rest *list-elements) body ...* *Special Form*

with-stack-list* simulates **list*** instead of **list**. (See the function **list*** in *Reference Guide to Symbolics-Lisp*.)

```
(with-stack-list* (var element1 element2...elementn)
  body)
```

is like

```
(let ((var (list* element1 element2...elementn)))
  body)
```

stack-let *clauses* &body *body* *Macro*

stack-let 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:

```
(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.

stack-let* *clauses* &body *body* *Macro*

stack-let* 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***.

2.13 The Data Stack

sys:with-stack-array (*var length* &key *type displaced-to displaced-index-offset displaced-conformally leader-list leader-length named-structure-symbol initial-value fill-pointer*) &body *body* *Special Form*

This form is like **with-stack-list** but makes an array. The array has dynamic lifetime and becomes garbage when this form is exited, just as for **with-stack-list**. The array is created on the data stack, which is part of a stack group. Only arrays can be allocated on the data stack.

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.

More information is available about stack arrays and the data stack. See the function **sys:make-stack-array**, page 28. See the function **sys:with-data-stack**, page 28.

The function **return-array** is another tool for manually freeing array storage.

sys:with-data-stack &body *body* *Special Form*

This primitive special form takes care of cleaning up the data stack when the body is exited. You sometimes want to optimize for extra speed by putting a **sys:with-data-stack** 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* &key *type displaced-to displaced-index-offset displaced-conformally leader-list leader-length named-structure-symbol initial-value fill-pointer* *Function*

This function is a special version of **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.

Currently, you cannot make anything but arrays on the data stack.

3. 3600-family Disk System User Interface

This chapter describes the portions of the 3600 family's 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 "3600-family Disk System Definitions and Constants", page 29.

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", page 31. See the section "Disk Events", page 32.

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", page 35.

To learn about the error-handling mechanism: See the section "Disk Error Handling", page 36.

For a discussion of the FEP file system and disk streams: See the section "FEP File System", page 42.

For a discussion of disk performance, along with some basic approaches for achieving high performance: See the section "Disk Performance", page 50.

For examples that illustrate concepts introduced in all the sections mentioned above: See the section "Examples of High Disk Performance", page 52.

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", page 58.

3.1 Definitions and Constants

The 3600-family disk system is capable of transferring data in either *32-bit mode* or *36-bit mode*. In 32-bit mode data are packed 32 bits per memory word, with a fixnum data type automatically supplied, making the data all integers. In 36-bit mode the data are packed into all 36 bits of a memory word, so the data type is supplied by the disk's data. 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. 32-bit mode is referred to as *user mode* and is handled by the *disk system user interface* described in this document. This document does not describe 36-bit mode, also called *system mode*, since it 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 the 3600 family, such as the Fujitsu M2284 and M2351 and the Maxtor XT-1140, have nonremovable disk packs.

Each disk drive is assigned a unique small positive number, called the *unit number*, that addresses 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 decimal. The term *disk unit* refers to both 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 between the disk and virtual memory. It consists of 64 bits called *checkwords* and 9216 bits of data. In user mode the data bits are packed into **si:disk-sector-data-size32** (288) fixnums. The two checkword fixnums are used by the FEP file system for identifying the block. If the disk block is not part of a FEP file system, the checkwords are available for use by the user program.

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.

sys:%%dpn-unit *Constant*

A byte specifier for accessing the unit number field in a disk address.

sys:%%dpn-page-num *Constant*

A byte specifier for accessing the block number field in a disk address. Block numbers are relative to a disk unit, with zero addressing the first disk block, and successive integers addressing consecutive blocks. The first disk block resides at cylinder zero, head zero, sector zero, with consecutive blocks being ordered by increasing sector numbers, then head numbers, and finally cylinder numbers.

si:disk-sector-data-size32 *Variable*

The value of this special variable is the number of data cells available in a disk block, excluding checkwords.

si:disk-block-length-in-bytes *Variable*

The number of bytes available in a disk block, excluding checkwords.

3.2 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; the blocks' checkwords are stored in the array's leader.

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", page 131.)

si:disk-array &optional *length* &rest *make-array-options* *Resource*

The **si:disk-array** resource is 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 **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 into the disk array. The default type is **art-q**.
:initial-value	The initial value to fill the array with, which must be a fixnum. The default value is zero.

The **si: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 value of this variable is the default area to allocate disk arrays in.

si:disk-array-block-count *disk-array* *Function*

This function accesses the slot in *disk-array* describing the number of disk blocks that the disk array can contain.

si:disk-array-checkwords *disk-array* *checkword-index* *Function*

This function accesses the checkwords stored in *disk-array*'s leader. The value of *checkword-index* specifies which checkword in *disk-array* is being accessed. For example, if *checkword-index* is 0, the first checkword of the first block stored in *disk-array* is accessed. A *checkword-index* value of 3 accesses the second checkword of the second block, since there are two checkwords per disk block.

3.3 Disk Events

Disk events are **art-2b** arrays 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", page 131.)

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.

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", page 36.) A disk event is associated with a disk transfer by the **si:disk-read** and **si:disk-write** functions.

si:disk-event *Resource*

The **si:disk-event** resource is 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.

3.3.1 Synchronization Functions

The following functions manipulate disk event tasks for synchronizing disk transfers:

si:with-disk-event-task *variable disk-event* &body *body* *Special Form*

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-number* *Function*
Deallocates the *task-number* task in *disk-event*.

si:disk-event-task-done-p *disk-event task-number* *Function*
Returns true if the *task-number* task in *disk-event* has completed. **nil** is returned if it has not completed.

si:wait-for-disk-event-task *disk-event task-number* *Function*
Waits for the *task-number* task in *disk-event* to complete.

si:wait-for-disk-event *disk-event* *Function*
Waits for all outstanding disk transfers associated with *disk-event* to complete.

si: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.

3.3.2 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", page 39. This accessor function is the predicate for determining if an error has occurred for a disk transfer associated with *disk-event*.

si:disk-event-error-unit *disk-event* *Function*

Accesses the slot in *disk-event* containing the unit number on which the error occurred. This slot contains a **nil** if the unit number was unrelated to the error.

si:disk-event-error-cylinder *disk-event* *Function*

Accesses the slot in *disk-event* containing the cylinder number on which the error occurred. This slot contains a **nil** if the cylinder number was unrelated to the error.

si:disk-event-error-head *disk-event* *Function*

Accesses the slot in *disk-event* containing the head number on which the error occurred. This slot contains a **nil** if the head number was unrelated to the error.

si:disk-event-error-sector *disk-event* *Function*

Accesses the slot in *disk-event* containing the sector number on which the error occurred. This slot contains a **nil** if the sector number was unrelated to the error.

si:disk-event-error-string *disk-event* *Function*

Accesses the slot in *disk-event* containing the error string supplied by the recovery routine.

si:disk-event-error-flushed-transfer-count *disk-event* *Function*

Accesses the slot in *disk-event* containing the total number of transfers aborted or removed from the disk queue due to the disk error.

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", page 39. Disk error codes are discussed in that section.

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))
```

si:disk-event-error-dcw *disk-event* *Function*

Accesses the slot in *disk-event* containing the first word of the disk command word block of the failed transfer.

3.4 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 System", page 42.

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 it enqueues the writes it specifies a *hang-p* argument of **nil** to **si: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 will also be aborted. (For more details on error handling: See the section "Disk Error Handling", page 36.) Thus, if the write of the file's page table fails and is aborted, the write of the directory page will also be 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.)

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", page 32. 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**) *Function*

si:disk-read 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), **si:disk-read** waits for all the reads to complete before returning. If *hang-p* is false **si:disk-read** returns immediately upon enqueuing the disk reads without waiting for completion. When *hang-p* is false all of the *disk-arrays* and the *disk-event* must be wired before calling **si: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.

sys:disk-write *disk-arrays disk-event dpn* &optional *n-blocks* (*hang-p* **t**) *Function*

si:disk-write 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 **si:disk-write** are identical to those of **si:disk-read**.

3.5 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 selectively suppress the automatic error recovery for a set of disk error types.) After **si:*n-disk-retries*** 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 retries the failed disk transfer **si:*n-disk-retries*** times, depending on the type of error. If one of the retries 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 failed disk transfer is aborted.

4. If the failed disk transfer does not have an associated disk event the unrecoverable error becomes fatal and halts the machine. (Most system mode disk transfers do not have an associated disk event.) Otherwise the information describing the error is stored in the disk event.

5. The disk system removes from the disk queue any remaining pending transfers that are associated with 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 (unless **si:*signal-disk-errors-from-enqueue-p*** is true, in which case a disk error is signalled from the disk transfer function, incrementing the disk event's **si:disk-event-error-flushed-transfer-count** slot).

8. When **si: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.

3.5.1 Disk Error Variables

si:*n-disk-retries* *Variable*

The value of **si:*n-disk-retries*** is the number of times to retry the failing disk operation before declaring it unrecoverable.

si:*signal-disk-errors-from-enqueue-p* *Variable*

This variable controls whether enqueuing a disk transfer associated with a disk event that is already associated with an failed transfer will signal an error or discard the enqueue request. If the value is true, an **si:disk-error-event** condition is signalled. If the value is false, which is the default, an error is not signalled and the transfer is discarded, incrementing the disk event's **si:disk-event-error-flushed-transfer-count** slot.

A false value is useful when multiple disk transfers are being enqueued without waiting for completion and it is not desirable to provide an error handler for each enqueue. In this case, the condition handler needs to be provided only for the final synchronization function.

The enqueue function still signals an error if it waits for completion of an failed transfer. For example, **si:disk-read** signals an error regardless of the value of **si:*signal-disk-errors-from-enqueue-p*** when its *hang-p* argument is true.

si:*automatically-recover-from-hung-disks* *Variable*

When this variable is false, the machine halts when the disk stops responding to transfer requests. A true value causes the disk system to attempt to recover from a hung disk. By default the value of the variable is true.

3.5.2 Disk Error Conditions

si:disk-error-event *Flavor*

This condition flavor is 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 **si:disk-error** condition; condition handlers should use the **si:disk-error** condition.

:disk-event of **si:disk-error-event** *Method*

This 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.

:error-type of **si:disk-error-event** *Method*

This method returns the error type code number, which is also stored in the disk event's **si:disk-event-error-code** slot. For a list of the possible disk error code numbers: See the section "Disk Error Codes", page 39.

:flushed-transfer-count of **si:disk-error-event** *Method*

This method returns the number of disk transfers that were not performed because of the error, including the failed transfer. The value is the same as is stored in the disk event's **si:disk-event-flushed-transfer-count** slot.

3.5.3 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* *Constant*

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's causes.

sys:%disk-error-select *Constant*

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 *Constant*

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.

The error recovery logic waits 60 seconds for the unit to be ready before signalling this error.

sys:%disk-error-device-check *Constant*

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. Otherwise the error recovery logic clears the fault condition and retries the disk transfer for **si:*n-disk-retries*** times before signalling this error.

sys:%disk-error-seek *Constant*

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 seeks to generate an error. The error recovery logic recalibrates the drive and retries the disk seek for **si:*n-disk-retries*** times before signalling this error.

sys:%disk-error-search *Constant*

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. The disk system recalibrates the disk drive and retries the disk transfer for **si:*n-disk-retries*** times before signalling this error.

sys:%disk-error-overrun *Constant*

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. The error recovery logic retries the disk transfer **si:*n-disk-retries*** times before signalling this error.

sys:%disk-error-ecc *Constant*

The data read from the disk has at least one invalid bit. The disk error recovery logic first attempts to correct the data, followed by a retry of the read transfer if the correction failed, for **si:*n-disk-retries*** 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. If a multiple blocks transfer had been requested, the disk array will not be modified for the blocks following the failed block.

sys:%disk-error-state-machine *Constant*

The disk hardware detected an error that was not already listed above. This can be caused by a number of disk system malfunctions. The error recovery logic resets the disk state and retries the disk transfer for **si:*n-disk-retries*** times before signalling this error.

sys:%disk-error-misc *Constant*

The disk microcode detected an error, but no error flags were set in the disk's status register. The error recovery logic resets the disk state and retries the disk transfer **si:*n-disk-retries*** times before signalling this error.

3.5.4 Disk Error Meters

These meters are updated when the disk system detects an error, including errors that are automatically recovered from. Meters that are primarily affected by system mode transfers are not included here. Most of these meters can be inspected with the Peek utility, too; type 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-not-ready* *Variable*
sys:%disk-error-not-ready 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.

si:*count-disk-device-checks* *Variable*
sys:%disk-error-device-check errors.

si:*count-disk-state-machine-errors* *Variable*
sys:%disk-error-state-machine errors.

si:*count-disk-other-errors* *Variable*
sys:%disk-error-misc errors.

si:*count-disk-hung-restarts* *Variable*
Times the disk was hung.

si:*count-disk-errors-lost* *Variable*
Times the disk was hung due to a disk error not waking up the disk software.

si:*count-disk-stops-lost* *Variable*
Times the disk was hung due to the disk system not waking up after the disk queue became empty.

3.6 FEP File System

The *FEP file system* manages the disk space available on a disk pack, grouping sets of data into named structures called *FEP files*. All the available space on a disk pack is described by the FEP file system. A single FEP file system cannot extend beyond a single disk pack; each disk pack has its own separate FEP file system.

The FEP file system supports all of the generic file system operations. It also supports multiple file versions, soft deletion and expunging, and hierarchical directories.

Although "FEP" is an acronym for *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.

Disk streams access FEP files. A disk stream is an I/O stream that performs input and output operations on the disk. (For information about streams: See the section "I/O Streams" in *Reference Guide to Streams, Files, and I/O*.). When disk streams are opened with a **:direction** keyword of **:input** or **:output**, the disk stream reads or writes bytes (respectively), buffering the data internally as required. When the **:direction** is **:block**, the disk stream can both read and write the specified disk blocks. Block mode disk streams address blocks with a block number relative to the beginning of the file, starting at file block number zero. This *file block number* is internally translated into the corresponding disk address.

The FEP file system is also used by the system for allocating system overhead files, such as the paging file. See the section "FEP File Types", page 49. This section lists some of these files and what they are used for.

The ability of the FEP to access FEP files and the use of FEP files by the system imposes some constraints on the design of the FEP file system. The internal data structures of the file system must be simple enough to permit the FEP to be able to read them, and a small amount of concurrent access by both the FEP and Lisp must be tolerated. A FEP file's data blocks should have a high degree of locality on the disk to minimize access times. And the FEP file system must be very reliable, since the FEP needs to use the file system for running diagnostics and for booting the machine.

Note: Because of these constraints, the FEP file system is not intended to be a replacement for LMFS. (See the section "Lisp Machine File System" in *Reference Guide to Streams, Files, and I/O*.) Allocating new blocks for FEP files is slow, so that creating many files, especially many small files, might impair the performance of the FEP file system, and ultimately the virtual memory system if paging files or world load files become highly fragmented.

3.6.1 Naming of FEP Files

See the section "Lisp Machine File System" in *Reference Guide to Streams, Files, and I/O*. The FEP filename format is similar to the LMFS filename format, with the following exceptions:

host — The name of the FEP file system host. The format for a FEP host is *host*|FEP*disk-unit*, where the *host* field specifies which machine's FEP file system is being referred to, and *disk-unit* specifies the disk unit number on the machine. The *host* field defaults to the local machine if it and the terminating vertical bar (|) are omitted. If both the *host* and *disk-unit* fields are omitted, the FEP host defaults to the disk unit the world was booted from on the local machine. For example:

`Merrimack|FEP0`	The FEP file system on Merrimack's unit 0.
`FEP2`	The FEP file system on the local machine's unit 2.
`FEP`	The FEP file system the booted world load file resides on.

directory — The name of the directory. The FEP file system supports hierarchical directories in the same format as in LMFS. Each directory name is limited to a maximum of 32 characters; there is no limit on the total length of a hierarchical directory specification.

name — The name of the FEP file, which cannot exceed 32 characters.

type — The type of the FEP file, which cannot exceed 4 characters.

version — The version number of the FEP file, which must be a positive integer or the characters "newest".

3.6.2 Accessing FEP Files

FEP files are accessed by open disk streams. A disk stream is opened by the **open** function. (See the section "Accessing Files" in *Reference Guide to Streams, Files, and I/O*. That section contains more details on accessing files.) If a FEP file system residing on a remote host is referred to, a *remote stream* is returned with limited operations as specified by the remote file protocol.

In addition to the normal **open** options, the following keywords are recognized:

:direction — Specifies the type of disk stream to open.

:input — Open a buffered input disk stream. A buffered input disk stream can only read bytes of data; write operations are not permitted. The

	number of disk blocks to buffer can be specified by the **:number-of-disk-blocks** keyword.
:output	Open a buffered output disk stream. A buffered output disk stream can only write bytes of data; read operations are not permitted. The number of disk blocks to buffer can be specified by the **:number-of-disk-blocks** keyword.
:block	Open a bidirectional block disk stream. Block disk streams are used to read and write random disk blocks of data as requested. Block disk streams do not internally buffer disk blocks. Block disk streams are not supported by the remote file protocol.
:probe	Open a probe disk stream. A probe stream can read and modify a FEP file's properties, but cannot read or modify the file's data.

:if-exists This keyword specifies the action to be taken if the specified file already exists and the **:direction** is **:output** or **:block**. This keyword is ignored when the **:direction** keyword is **:input** or **:probe**. Only the following values are supported:

:error	Signal an error. This is the default when the version component of the file name is not **:newest**.
:new-version	Create a new version of the file. This is the default when the version component of the file name is **:newest**.
:overwrite	Use the existing file.
:supersede	Supersede the existing file by deleting and expunging it.
nil	Return **nil** if the file already exists without creating a file or a stream.

:if-does-not-exist This keyword specifies the action to be taken if the specified file does not exist.

:error	Signal an error. This is the default if the **:direction** is **:input** or **:probe**, or if the **:if-exists** argument is **:overwrite**.
:create	Create a new file with the specified file name. This is the default if the **:direction** is **:output**

		or **:block**, and the **:if-exists** argument is not **:overwrite**.
	nil	Return **nil** if the file does not already exist without creating a file or a stream.

:if-locked This keyword specifies the action to be taken if the specified file is locked. This keyword is not supported by the remote file protocol.

:error	Signal an error. This is the default.
:share	Open the specified file even if it is already locked, incrementing the file's lock count. This mode permits multiple processes to simultaneously write to the same file. (See the section "FEP File Locks", page 48. That section contains more information on file locks.)

:estimated-length

The value of this keyword is the minimum number of bytes to preallocate for the file. If the file's block length is not large enough to accommodate **:estimated-length** bytes of data, disk blocks are allocated and appended to the file. If the file's block length is greater than is required to satisfy **:estimated-length**, its size is not adjusted. This keyword is ignored if the **:direction** keyword is **:input** or **:probe**.

:number-of-disk-blocks

The value of this keyword is the number of disk blocks to buffer internally if the **:direction** keyword is **:input** or **:output**. This keyword is ignored for other values of **:direction** or for files on remote hosts. The default **:number-of-disk-blocks** is two.

3.6.3 Operating on Disk Streams

All disk streams to a local FEP file system handle the following messages:

:grow &optional *n-blocks* &key **:map-area** **:zero-p** *Message*

This 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 described in **:create-data-map**'s description below). 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*

This 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 described in **:create-data-map**'s description below). **: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*

This message returns the disk stream's file access path.

For example, you can find out what unit number a FEP file resides on as follows:

```
(send (send stream :file-access-path) :unit)
```

:map-block-no *block-number grow-p* *Message*

This 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 if 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 appended 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*

This 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, and 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*

This 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.

3.6.4 Input and Output Disk Streams

Input and output disk streams are buffered streams. In addition to the standard buffered stream messages, local input and output disk streams also support the messages described elsewhere: See the section "Operating on Disk Streams", page 45.

Input disk streams read bytes of data starting at the current byte position in the FEP file, updating the byte position as the data is read. Output disk streams write bytes of data in the same way.

The bytes of data are stored in buffers internal to the stream. The **:number-of-disk-blocks open** keyword 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 in from the file for an input stream. Output streams also write out all the buffered disk blocks when the stream is sent a **:close** message without an **:abort** option.

3.6.5 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. They are not supported by the remote file protocol.

See the section "Operating on Disk Streams", page 45. In addition to the messages described in that section, block disk streams support the following messages:

:block-length *Message*

The **: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*

The **:block-in** message causes the disk to start reading data from the disk into the disk arrays in *disk-arrays* starting with the file block number *block-number* for *n-blocks*. *disk-arrays* can be a disk array or a list of 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 disk array in *disk-arrays*. Unused disk arrays or portions of disk arrays remain unmodified.

When the value of **:hang-p** is true, which it is by default, the **:block-in** message 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*

The **:block-out** 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* for *n-blocks*. The arguments to the **:block-out** message are identical to those of the **:block-in** message.

3.6.6 FEP File Properties

In addition to having a name and containing data, FEP files also have properties. These properties store information about the file itself, such as when it was last written and whether it can be deleted or not. File properties are read by the **fs:file-properties** function, and modified by the **fs:change-file-properties** function. The **fs:directory-list** function also returns the file properties of several files at once. (See the section "Accessing Directories" in *Reference Guide to Streams, Files, and I/O*.)

The following file properties can be both read and modified:

:creation-date The universal time the file was last written to. Universal times are integers. (See the section "Dates and Times" in *Programming the User Interface*.)

:author The user-id of the last writer. The user-id must be a string.

:length-in-bytes The length of the file expressed as an integer.

:deleted When **t** the file is marked as being deleted. A deleted file can then be marked as being undeleted by changing this property to be **nil**. The disk space used by a deleted file is not actually reclaimed for reuse until the file is expunged.

:dont-delete When **t**, attempting to delete or overwrite the file signals an error, otherwise **nil** indicating the file can be deleted or written to.

:comment A comment to be displayed in brackets in the directory listing. The comment must be a string.

The following file properties are returned by the **:properties** message, but cannot be modified by **:change-properties**:

:byte-size The number of bits in a byte. The value of this property is always 8.

:length-in-blocks The block length of the file expressed an an integer.

:directory If **t**, the file is a directory, otherwise **nil** if the file is not a directory.

3.6.7 FEP File Locks

A FEP file is *locked* for the interval from when 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**, which is its default, a file that is read-locked can still be opened for reading but signals an error if opened for writing; a file that is write-locked cannot be opened for reading or writing. This permits multiple readers to access a file concurrently, while prohibiting writing to the file being read.

When the **:if-locked** keyword is **:share** in an open call for write, it succeeds in opening the file even if it is already read- or write-locked.

An expunge operation on a file that is either read- or write-locked does not expunge the file. If expunging a directory fails to expunge a file, the file must be closed and the directory expunged again.

3.6.8 FEP File Types

By convention, the following file types are used by the FEP file system for files used by the system.

BOOT
: The file contains FEP commands that can be read be FEP's Boot command. BOOT files are text files, and can be manipulated by the editor.

LOAD
: The file contains a world load image that is used to boot the system. For example, >Release-6.load.NEWEST contains the release 6 world load image.

MIC
: The file contains a microcode image. For example, >TMC5-MIC.MIC.234 contains version 234 of the microcode for version 5 of the TMC.

FSPT
: The file contains a LMFS partition table. For example, >FSPT.FSPT.NEWEST is the default partition table used by LMFS.

FILE
: The file contains a LMFS partition. For example, >LMFS.FILE.NEWEST is the default LMFS file partition.

PAGE
: The file contains disk space that can be used by the virtual memory system. For example, >PAGE.PAGE.NEWEST is the default file used by the virtual memory system as storage for swapping pages in and out of main memory.

FLOD
: The file contains a FEP Load file. FEP Load files contain binary code the FEP can load and execute.

FEP
: The file contains binary information used by the FEP file system. These files should not be written to by user programs. Some examples of these files are:

  >FREE-PAGES.FEP
  : Describes which blocks on the disk are allocated to existing files.

>BAD-BLOCKS.FEP
Owns all the blocks that contain a media defect and should not be used.

>SEQUENCE-NUMBER.FEP
Contains the highest sequence number in use. The FEP file system uses sequence numbers internally to uniquely identify files to assist in rebuilding the file system in case of a catastrophic disk failure.

>DISK-LABEL.FEP
Contains the disk pack's physical disk label. The label is used to identify the pack and describe its characteristics.

DIR
The file contains a FEP directory. For example, FEP0:>ROOT-DIRECTORY.DIR.NEWEST contains the top-level root directory. The directory file for FEP0:>DanG>Examples> would reside in FEP0:>DanG>Examples.DIR.1.

3.7 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. Computations can then continue while the disk is concurrently 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 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.

$$T_n = T_a + T_r/2 + NT_r/S + T_s\lfloor((A \bmod HS)+N-1)/HS\rfloor \qquad (1)$$

Where:

A	The disk block number. The **sys:%%dpn-page-num** field of the disk address.
H	Number of data heads, excluding any servo heads.
N	Number of blocks to transfer.
S	Number of blocks per track.
T_a	Average seek time.
T_n	Minimum time to transfer N blocks.
T_r	Rotation time.
T_s	Average single cylinder seek time.
$\lfloor x\rfloor$	*Floor* of x. The truncated integer value of x.

The terms in Eq. 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 Eq. 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, Eq. 1 becomes:

$$T_n = T_a + NT_r/2 + NT_r/S + T_s\lfloor((A \bmod HS)+N-1)/HS\rfloor \qquad (2)$$

Eq. 2 shows that in addition to the cost of not performing computations in parallel 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, Eq. 2 is only true 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.

3.8 Examples of High Disk Performance

3.8.1 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-structure 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.

3.8.2 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 copy 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 problem 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 making the **:block-in** and **:block-out** messages not wait for completion by supplying a **:hang-p** keyword with a value of **nil**. 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-structure 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 still be 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 eliminating disk seeks by reading multiple source blocks, then seeking to the destination file and writing multiple destination blocks. 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-structure disk-array)
                    (aset disk-array disk-arrays i)))
                (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-structure disk-array))
  (deallocate-resource 'si:disk-array disk-array))))))))
```

3.9 Disk and FEP File System Utilities

3.9.1 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 "Front-end Processor" in *User's Guide to Symbolics Computers*.) 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* *Function*

Permits the disk label of the disk unit *unit* to be edited by exposing a chose 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.

3.9.2 Mounting a Disk Unit

Disk units can be *mounted* either by the FEP or by Lisp. (See the section "Front-end Processor" in *User's Guide to Symbolics Computers*.) 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.

si:mount-disk-unit *unit* *Function*

Make 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 address an online disk unit.

3.9.3 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 (*correct-bittable* **: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.

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.

3.9.4 Writing FEP Files to Tape

You can write files to tape using a local tape drive with the **tape:write-fep-files-to-tape** function. This can be used for large (requiring more than one cartridge tape) FEP files and is very useful with large world loads. To do this, you first get access to the necessary software by making a **fep-tape** system. You can use the **:silent** and **:noconfirm** options, as shown in the following example:

```
(make-system 'fep-tape :silent :noconfirm)
```

The next step is to use the function **tape:write-fep-files-to-tape** to write the FEP files to tape. This can be used to write both microcode and world load files.

To restore these files from tape, use the FEP command Disk Restore. See the section "Software Installation Guide" in *Installation and Site Operations*.

When the end of tape is encountered, the machine will return to the FEP. You then put the second tape into the tape drive, and use another Disk Restore command using the same destination filename. This appends the data from the second tape onto the designated file.

tape:write-fep-files-to-tape &optional *mic-name* *Function*

Writes FEP files to tape. *mic-name* is the name of file-format microcode that precedes the microcode and world load files on distribution tapes.

When an argument is supplied within the form, the function assumes that the argument is the file-format microcode and uses stream format. When an argument is not supplied, you are prompted for a file name, which is assumed to be a microcode or world load file and which is then written out in distribution format. Thus, supplying a file-format microcode name should be used only when writing an initial microcode file to tape.

You will be prompted as to whether the first tape is in place. Put the tape in the local tape drive and then answer "Y". You will then be prompted as to whether you wish to write a file to tape; you should answer "Y". Next, enter the filename of the world load. You will also be prompted for file and restoration comments. As the file is written out, the number of blocks will be printed on the screen. When the end of the tape is reached, the following message is printed:

```
"starting a new tape"
```

and you will be prompted as to whether a new tape is in place. Put a fresh tape in the drive and type "Y" to continue. This will continue writing the file on the second tape.

4. PC Metering on the 3600 Family

Program counter (PC) metering is a tool to allow the user to determine where time is being spent in a given program.

PC metering essentially produces a histogram. At regular intervals, the front-end processor (FEP) causes the main processor to task switch to special microcode. This microcode looks up the macro PC that contains the virtual address of the macroinstruction that the processor is currently executing. If this virtual address falls outside the *monitored range*, the microcode increments a count of the number of PCs that missed the monitored range. If the address is within the monitored range, the microcode subtracts the bottom of the monitored range from the PC, leaving a word offset. It then divides the word offset by the number of words per *bucket* and uses that as an index into the *monitor array*. Next, it increments that indexed element of the monitor array. This can only measure statistically where the macro PC is pointing; for the results to be valid, a relatively large number of samples per bucket must be available. FEP version 13 samples at about 170 samples per second, so the PC monitoring with that version is probably valid only for sessions that take longer than five to ten seconds.

You specify some range of the program to be monitored. The range is specified by lower and upper bounding addresses, and compiled functions that lie between those addresses are monitored. The range is divided into some number of buckets. The relative amount of time that the program spends executing in each bucket is measured.

The parameters you specify are the range of addresses to be monitored, the number of buckets, and an array with one word for each bucket.

Some of the metering functions deal with *compiled functions*. In this context a compiled function is either a compiled code object or an **art-16b** array, into which escape functions (small, internal operations used by the microcode) compile.

meter:make-pc-array *size* *Function*

Makes a PC array with *size* number of buckets. This storage is wired, so you probably do not want this to be more than about 64. pages, or **(* 64. sys:page-size)** words.

meter:monitor-all-functions *Function*

Changes the microcode parameters so that the monitor array refers to every possible function in the Lisp world at the time of the execution of **meter:monitor-all-functions**. This usually causes many functions to map into a single bucket, and is therefore useful in obtaining a first estimate of which functions are using a significant portion of the execution time.

meter:setup-monitor &optional (*range-start* **0**) (*range-end* **268435456**) *Function*

Monitors the region between *range-start* and *range-end*.

meter:monitor-between-functions *lower-function upper-function* *Function*

Monitors all functions between *lower-function* and *upper-function*. This does not work in some situations, such as:

- You compile a function from a buffer, which puts its definition outside the range

- A previous region is extended, and new functions go there instead of in monotonically increasing virtual addresses.

Example:

```
(defun start-of-library ()())
            ...code...
            (defun end-of-library ()())
   (meter:monitor-between-functions #'start-of-library #'end-of-library)
```

meter:expand-range *start-bucket* &optional (*end-bucket start-bucket*) *Function*

Changes the microcode parameters so that the entire monitor array refers only to the functions previously contained within the range specified by *start-bucket* and *end-bucket*. *start-bucket* and *end-bucket* are inclusive bounds.

meter:report &optional *function-list* *Function*

Prints a summary of the data collected into the monitor array. You should not have to supply the *function-list* argument.

meter:start-monitor &optional (*clear* **t**) *Function*

Enables collection of PC data. If *clear* is not **nil**, the contents of the monitor array are cleared. If *clear* is **nil**, the array is not modified, so that the new samples are simply added to the old.

meter:stop-monitor *Function*

Disables further collection of PC data.

meter:print-functions-in-bucket *bucket* *Function*

Prints all the compiled functions that map into the specified *bucket*.

meter:list-functions-in-bucket *bucket* *Function*

Returns a list of all the compiled functions that map into the specified *bucket*.

meter:range-of-bucket *bucket* *Function*

Returns the virtual address range that maps into the specified *bucket*.

meter:with-monitoring *clear body...* *Macro*

Enables monitoring around the execution of *body*. If *clear* is not **nil**, clears the monitor array first. See the function **meter:start-monitor**, page 62.

meter:map-over-functions-in-bucket *bucket function* &rest *args* *Function*

Calls *function* for every compiled function in the specified *bucket*. The first argument to *function* should be the compiled function, and any remaining arguments are *args*.

meter:function-range *function* *Function*

Returns two values, the buckets that contain the first and last instructions of *function*.

meter:function-name-with-escapes *object* *Function*

If *object* is a compiled function, returns the function spec of the compiled function. Otherwise, returns **nil**.

PART II.

Initializations

5. Introduction to Initializations

A number of programs and facilities in the Symbolics computer require that "initialization routines" be run either 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.

There is something called an *initialization list*, which 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* **'si:warm-initialization-list** *list-name-supplied-p*) *Function*

Adds an initialization called *name* (a string) with the form *form* to the initialization list specified either by *list-name* or by keyword. If the initialization list already contains an initialization called *name*, its form is changed to *form*.

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 a keyword specifies an initialization list, *list-name* is ignored and should not be specified.

The *keywords* allowed are of two kinds. These specify what initialization list to use:

: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.

: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", page 71.

These specify when to evaluate *form*:

: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** 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 list keyword should come *before* the when keyword in *keywords*; otherwise the list 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", page 71.

delete-initialization *name* &optional *keywords* (*list-name* **'si:warm-initialization-list**) *Function*

Remove 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* **nil**) (*flag* **t**) *Function*

Perform 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-value* 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*

Bashes the flag of all entries in the specified list to **nil**, thereby causing them to get 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 **string-equal** is used to recognize the symbols.

6. 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 **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 only evaluated 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 **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 **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 **login** and **logout** functions, respectively. Note that **disk-save** calls **logout**. Also note that often people do not call **logout**; they just cold boot the machine.

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

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

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.

PART III.

Processes

7. Introduction

The Symbolics computer supports *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", page 3.)

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 inactive.

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 processor scheduler, and does not compete for machine resources.

To get a computation to happen in another process, you must first create a process, and then say what computation you want to happen 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 that 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 function **throw** in *Reference Guide to Symbolics-Lisp.*) 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 how you 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, the Symbolics computer allows 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**, page 17.
See the special form **without-interrupts**, page 78.
See the function **process-lock**, page 83.

A process is a Lisp object, an instance of one of several flavors of process.

8. The Scheduler

At any time there is a set of *active processes*; these are all the processes that are not stopped. Each active process is either currently running, trying to run, or waiting for some condition to become true. The active processes are managed by a special stack group called the *scheduler*, which repeatedly cycles through the active processes, determining for each process whether it is ready to be run or whether it is waiting. The scheduler determines whether a process is ready to run by applying the process's *wait-function* to its *wait-argument-list*. If the wait-function returns a non-**nil** value, then the process is ready to run; otherwise, it is waiting. If the process is ready to run, the scheduler resumes the current stack group of the process.

When a process's wait-function returns non-**nil**, the scheduler resumes its stack group and lets it proceed. The process is now the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable **current-process** to it. It remains the current process and continues to run until either it decides to wait, or a *sequence break* occurs and causes the process to remove itself from scheduling. In either case, the scheduler stack group is resumed and it continues to cycle through the active processes. This way, each process that is ready to run gets its share of time in which to execute.

A process can wait for some condition to become true by calling **process-wait**, which sets up its wait-function and wait-argument-list accordingly, and resumes the scheduler stack group. A process can also wait for just a moment by calling **process-allow-schedule**, which resumes the scheduler stack group but leaves the process runnable; it will run again as soon as all other runnable processes of the same or higher priority have had a chance.

A sequence break is a kind of interrupt that is generated by the Lisp system for any of a variety of reasons; when it occurs, the scheduler is resumed. The function **si:sb-on** can be used to control when sequence breaks occur. The default clock interval used by **si:sb-on** is controlled by the variable **si:*default-sequence-break-interval***. Thus, if a process runs continuously without waiting, it is forced to return control to the scheduler once per this interval so that any other runnable processes get their turn.

The system does not generate a sequence break when a page fault occurs; thus time spent waiting for a page to come in from the disk is "charged" to a process the same as time spent computing, and cannot be used by other processes. It is done this way for the sake of simplicity; this allows the whole implementation of the process system to reside in ordinary virtual memory, and not to have to worry specially about paging. The performance penalty is small since Symbolics computers are personal computers, not multiplexed among a large number of processes. Usually only one process at a time is runnable.

A process's wait-function is free to touch any data structure it likes and to perform any computation it likes. Of course, wait-functions should be kept simple, using only a small amount of time and touching only a small number of pages, or system performance will be affected, since the wait-function consumes resources even when its process is not running. If a wait-function gets an error, the error occurs inside the scheduler. All scheduling comes to a halt and the user is thrown into the Debugger. Wait-functions should be written in such a way that they cannot get errors. Note that **process-wait** calls the wait function once before giving it to the scheduler, so an error due simply to bad arguments will not occur inside the scheduler.

Note well that a process's wait-function is executed inside the scheduler stack group, *not* inside the process. This means that a wait-function cannot access special variables bound in the process. It is allowed to access global variables. It could access variables bound by a process through the closure mechanism, but more commonly any values needed by the wait-function are passed to it as arguments. See the section "Closures" in *Reference Guide to Symbolics-Lisp.*

current-process *Variable*

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.

without-interrupts *body...* *Special Form*

The *body* forms are evaluated with **inhibit-scheduling-flag** bound to **t**. This is the recommended way to lock out multiprocessing over a small critical section of code to prevent timing errors. In other words the body is an *atomic operation*. The value(s) of a **without-interrupts** is/are the value(s) of the last form in the body.

```
Examples:
(without-interrupts
  (push item list))

(without-interrupts
  (cond ((memq item list)
         (setq list (delq item list))
         t)
        (t nil)))
```

inhibit-scheduling-flag *Variable*

The value of **inhibit-scheduling-flag** is normally **nil**. If it is **t**, preempts are deferred until **inhibit-scheduling-flag** becomes **nil** again. This means that no process other than the current process can run.

process-wait *whostate function* &rest *arguments* *Function*

This is the primitive for waiting. The current process waits until the application of *function* to *arguments* returns non-**nil** (at which time **process-wait** returns). Note that *function* is applied in the environment of the scheduler, not the environment of the **process-wait**, so bindings in effect when **process-wait** was called are *not* in effect when *function* is applied. Be careful when using any free references to special variables in *function*. *whostate* is a string containing a brief description of the reason for waiting. If the status line at the bottom of the screen is looking at this process, it shows *whostate*.

```
Examples:
(process-wait "sleep"
        #'(lambda (now)
             (> (time-difference (time) now) 100.))
        (time))

(process-wait "Buffer"
        #'(lambda (b) (not (zerop (buffer-n-things b))))
        the-buffer)
```

process-sleep *interval* &optional (*whostate* **"Sleep"**) *Function*

This simply waits for *interval* sixtieths of a second, and then returns. It uses **process-wait**.

process-wait-with-timeout *whostate time function* &rest *args* *Function*

This is a primitive for waiting. It applies *function* to *args* until the function returns something other than **nil** or until the interval times out. *time* is a time in 60ths of a second. When the process times out, **process-wait-with-timeout** returns **nil**. When the function returns something other than **nil** within the interval, **process-wait-with-timeout** returns **t**.

If *time* is **nil**, **process-wait-with-timeout** waits indefinitely for the application of *function* to *arguments* to return something other than **nil**. This behavior is the same as that of **process-wait**.

process-wait-forever &optional (*whostate* **"Wait Forever"**) *Function*

This function causes the current process to wait forever. The process is still active, though, and will begin running again if reset or preset.

process-allow-schedule *Function*

This function simply waits momentarily; all other processes get a chance to run before the current process runs again.

sys:scheduler-stack-group *Variable*

This is the stack group in which the scheduler executes.

sys:clock-function-list *Variable*

This is a list of functions to be called by the scheduler 60 times a second. Each function is passed one argument: the number of 60ths of a second since the last time that the functions on this list were called. These functions implement various system overhead operations, such as blinking the blinking cursor on the screen.

Note that these functions are called inside the scheduler, just as are the functions of simple processes. (See the flavor **si:simple-process**, page 95.) The scheduler calls these functions as often as possible, but never more often than 60 times a second. That is, if there are no processes ready to run, the scheduler calls the functions 60 times a second, assuming that, all together, they take less than 1/60 second to run. If there are processes continually ready to run, then the scheduler calls these functions as often as it can; usually this is ten times a second, since usually the scheduler only gets control that often.

sys:active-processes *Variable*

This is the scheduler's data structure. It is a list of lists, where the car of each element is an active process or **nil** and the cdr is information about that process.

sys:all-processes *Variable*

This is a list of all the processes in existence. It is mainly for debugging.

si:initial-process *Variable*

This is the process in which the system starts up when it is booted.

si:sb-on &optional *when* *Function*

si:sb-on controls what events cause a sequence break, that is, when rescheduling occurs. The following keywords are names of events that can cause a sequence break.

:clock — This event happens periodically based on a clock and is enabled by default. The period is the value of the variable **si:sequence-break-interval**, initially having the value of the variable **si:*default-sequence-break-interval***.

:disk — A sequence break happens whenever the disk hardware/firmware decides to wake up the wired disk system. This might occur with every disk I/O operation or after several have been completed. This event is always enabled; you cannot turn it off. However, these sequence breaks do not cause rescheduling.

:mouse — Happens when the mouse moves. Sixty times per second it tests the variable **tv:mouse-wakeup**, which is set by the FEP. Causes a sequence break if the value is not **nil**. This event is enabled by default.

:keyboard Happens whenever a key is typed.

With no argument, **si:sb-on** returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, or **nil** (which disables sequence breaks entirely, since it is the empty list).

si:*default-sequence-break-interval* *Variable*

This variable controls the interval used by **si:sb-on**. Its default value is 100000 microseconds (0.1 seconds).

9. Locks

A *lock* is a software construct used for synchronization of two processes. 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. A lock protects some resource or data structure so that only one process at a time can use it.

In the Symbolics computer, 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**") *Function*

This is used to seize 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 *lock-value* is **nil** or unsupplied, the value of **current-process** is used.

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

This is used to unlock 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 *error-p* is **t** (the default), an error is signalled if *lock-value* does not have the same value as the contents of the cell. If *lock-value* is **nil** or unsupplied, the value of **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. 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, 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, 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, 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. 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 process 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.

10. Creating a Process

There are two ways of creating a process. One is to create a "permanent" process that you will hold on to and manipulate as desired. The other way is to say simply, "call this function on these arguments in another process, and don't bother waiting for the result." In the latter case you never actually use the process itself as an object.

make-process *name* &rest *init-args* *Function*

Creates and returns a process named *name*. 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", page 95.

: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", page 95.

: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 **(:method si:process :warm-boot-action)**, page 91. See the method **(:method si:process :set-warm-boot-action)**, page 91.

:quantum See the method **(:method si:process :quantum)**, page 90. See the method **(:method si:process :set-quantum)**, page 90.

:priority See the method **(:method si:process :priority)**, page 90. See the method **(:method si:process :set-priority)**, page 90.

:run-reasons Lets you supply an initial run reason. The default is **nil**.

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

In addition, the options of **make-stack-group** are accepted. See the function **make-stack-group**, page 5.

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

The following three functions allow 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 complete. When the function returns, the process is returned to a pool of free processes, making these operations quite efficient. The only difference among these three functions is in what happens if the machine is booted while the process is still active.

Normally the function to be run should not do any I/O to the terminal. For a discussion of the issues: See the section "Input/Output in Stack Groups", page 7.

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 symbol or 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 is "Anonymous".

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

:restart-after-boot
If this is **nil**, warm booting the machine flushes the process. If this is **t**, warm booting the machine restarts 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. 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 method **(:method si:process :warm-boot-action)**, page 91.

:priority The priority of the process. The default is 0.

:quantum The scheduler quantum of the process. The value should be a fixnum in units of 60ths of a second. The default is 60 (one second). For information on the meaning of these numbers: See the section "How to Choose Process Priority Levels", page 87.

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

Creates a process named *name*, presets it so it will apply *function* to *args*, and starts it running. If the machine is warm booted, the process is killed.

process-run-temporary-function is obsolete; use **process-run-function** instead.

process-run-restartable-function *name function* &rest *args* *Function*

Creates a process, presets it so it will apply *function* to *args*, and starts it running. *name* can be a symbol or 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 (Becomes the process's name; default is "Anonymous")
:priority
:quantum
:restart-after-reset
:restart-after-boot
:warm-boot-action

The default values are the same as for **process-run-function**, except that the default values of the **:restart-after-boot** and **:restart-after-reset** options are **t** rather than **nil**.

10.1 How to Choose Process Priority Levels

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

Processes run with a default priority of 0. If the priority number is higher, the process receives higher priority. You should avoid using priority values higher than 20, since some critical system processes use priorities of 25 and 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 5 or 10 for urgent processes are reasonable.

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

Be advised that process priorities are absolute. If a priority 1 process runs forever without calling **process-wait**, no lower-priority process will ever run.

11. Process Messages

These are 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 the user".

11.1 Process Attributes

:name of **si:process** *Method*

Returns the name of the process, which was the first argument to **make-process** or **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.

:stack-group of **si:process** *Method*

Returns the stack group currently executing on behalf of this process. This can be different from the initial-stack-group if the process contains several stack groups that coroutine among themselves.

Note that the stack group of a *simple* process is not a stack group at all, but a function. See the flavor **si:simple-process**, page 95.

:initial-stack-group of **si:process** *Method*

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

:initial-form of **si:process** *Method*

Returns the initial "form" of the 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.

In a simple process, the initial form is a list of one element, the process's function. See the flavor **si:simple-process**, page 95.

To change the initial form, send the **:preset** message.

:wait-function of **si:process** *Method*

Returns the 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").

:wait-argument-list of **si:process** *Method*

Returns the arguments to the 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.

:whostate of **si:process** *Method*

Returns a string that is the state of the process to go in the status line at the bottom of the screen. This is "**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 who-state string is ignored and the status line displays **arrest** if the process is arrested or **stop** if the process has no run reasons.

:quantum of **si:process** *Method*

Returns the number of 60ths of a second this process is allowed to run without waiting before the scheduler runs someone else. The quantum default is governed by the variable **si:default-quantum**.

:set-quantum *60ths* of **si:process** *Method*

Changes the number of 60ths of a second this process is allowed to run without waiting before the scheduler runs someone else. The quantum default is governed by the variable **si:default-quantum**.

si:default-quantum *Variable*

This variable governs the default amount of time a process is allowed to run before rescheduling, in 60ths of a second. The default is 6 (0.1 second).

:quantum-remaining of **si:process** *Method*

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

:priority of **si:process** *Method*

Returns the priority of this process. The larger the number, the more this process gets to run. Within a priority level the scheduler runs all runnable processes in a round-robin fashion. Regardless of priority a process will not run for more than its quantum. The default priority is 0, and no normal process uses other than 0, except for some internal system processes that run at high priority.

:set-priority *priority-number* of **si:process** *Method*

Changes the priority of this process. The larger the number, the more this process gets to run. Within a priority level the scheduler runs all runnable processes in a round-robin fashion. Regardless of priority a process will not run for more than its quantum. The default priority is 0, and no normal process uses other than 0, except for some internal system processes that run at high priority.

:warm-boot-action of **si:process** *Method*

Returns the process's warm-boot-action, which controls what happens if the machine is booted while this process is active. (Contrary to the name, this applies to both cold and warm booting.) This can be **nil**, which means to "flush" the process, or a function to call. The default is **si: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 **si:process-warm-boot-reset**, which throws out of the process's computation and kills the process.

:set-warm-boot-action *action* of **si:process** *Method*

Changes the process's warm-boot-action, which controls what happens if the machine is booted while this process is active. (Contrary to the name, this applies to both cold and warm booting.) This can be **nil**, which means to "flush" the process, or a function to call. The default is **si: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 **si:process-warm-boot-reset**, which throws out of the process's computation and kills the process.

:simple-p of **si:process** *Method*

Returns **nil** for a normal process, **t** for a simple process. See the flavor **si:simple-process**, page 95.

11.2 Run and Arrest Reasons

:run-reasons of **si:process** *Method*

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

:run-reason *object* of **si:process** *Method*

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

:revoke-run-reason *object* of **si:process** *Method*

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

:arrest-reasons of **si:process** *Method*

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

:arrest-reason *object* of **si:process** *Method*

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

:revoke-arrest-reason *object* of **si:process** *Method*

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

:active-p of **si:process** *Method*

This message is the same as **:runnable-p** of **si:process.** **t** is returned if the process is active, that is, it can run if its wait-function allows. **nil** is returned if the process is stopped.

:runnable-p of **si:process** *Method*

This message is the same as **:active-p** of **si:process.** **t** is returned if the process is active, that is, it can run if its wait-function allows. **nil** is returned if the process is stopped.

11.3 Bashing the Process

:preset *function* &rest *args* of **si:process** *Method*

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 **:preset** message 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.

:reset &optional *unwind-option kill* of **si:process** *Method*

Forces the process 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 message 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 **:kill** message only.

A **:reset** message 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.

:flush of **si:process** *Method*

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

:kill of **si:process** *Method*

Gets rid of the process. It is reset, stopped, and removed from **sys:all-processes**.

:interrupt *function* &rest *args* of **si:process** *Method*

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 **:interrupt** message returns immediately, but if the process's stack group is in an unusual internal state it might have to wait for it to get out of that state.

12. Process Flavors

These are the flavors of process provided by the system. It is possible for users to define additional flavors of their own.

si:process *Flavor*

This is the standard default kind of process.

si:simple-process *Flavor*

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 **si:set-process-wait**. So, when a simple process wants to wait for a condition, it should call **si:set-process-wait** to specify the condition, and then return.

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

Set the *wait-function* and *wait-argument-list* of *simple-process*. For more information: See the flavor **si:simple-process**, page 95.

13. Other Process Functions

process-enable *process* *Function*

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

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

Resets *process* then enables it.

process-disable *process* *Function*

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

The remaining functions in this section are obsolete, since they simply duplicate what can be done by sending a message. They are documented here because their names are in the **global** package.

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

Just sends a **:preset** message.

process-reset *process* *Function*

Just sends a **:reset** message.

process-name *process* *Function*

Gets the name of a process, like the **:name** message.

process-stack-group *process* *Function*

Gets the current stack group of a process, like the **:stack-group** message.

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

Gets the initial stack group of a process, like the **:initial-stack-group** message.

process-initial-form *process* *Function*

Gets the initial "form" of a process, like the **:initial-form** message.

process-wait-function *process* *Function*

Gets the current wait-function of a process, like the **:wait-function** message.

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

Gets the arguments to the current wait-function of a process, like the **:wait-argument-list** message.

process-whostate *process* *Function*

Gets the current status line state string of a process, like the **:whostate** message.

PART IV.

Storage Management

14. Overview of Storage Management

The Symbolics-Lisp 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 1 billion bytes.

Symbolics-Lisp 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 ("dynamically") and then discarding or freeing the memory for reuse when it is no longer needed for the same purpose.

14.1 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 Symbolics-Lisp is performed by a suite of programs collectively called the garbage collector. See the section "The Garbage Collector", page 113.

Also provided are areas, which help you improve the locality of reference in programs without giving up the ease of automatic storage management. See the section "Areas", page 103. See the section "Locality of Reference", page 121.

14.2 Manual Storage Management

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

The primary facility for manual storage management is the *resource*. See the section "Resources", page 131.

15. 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 can be greatly increased. Whenever a new object is created the area to be used can optionally be specified. 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 **default-cons-area**; most functions that allocate storage use the value of this variable, by default, to specify the area to use.

There is a default area (**working-storage-area**) that collects objects you have not chosen to control explicitly.

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, a different paging algorithm, and even a different data representation. (The data-representation feature is not currently used by the system, except for the list/structure distinction described here.)

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 when you start.

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 that exist now 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.

15.1 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 **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 **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.

make-area &key *name size region-size representation gc read-only swap-recommendations n-levels capacity capacity-ratio room %%region-space-type %%region-scavenge-enable* *Function*

This 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.

: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 a suitable medium size. Note that if you specify **:size** and not

:region-size, the area will have exactly one region. When making an area that will be very big, it is desirable to make the region size larger than the default region size to avoid creating very many regions and possibly overflowing the system's fixed-size region tables.

: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.

:gc The type of garbage collection to be employed. The choices are **:dynamic** (which is the default), **:temporary**, **:ephemeral**, and **:static**. **: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. **: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. **: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", page 107.

: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.

: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
A fixnum specifying the capacity of a level in words (default 200000 octal); 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

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**.

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

sys:%%region-space-type
Lets you specify the *space type* explicitly, overriding the specification from the other keywords. It is rarely useful in user programs. The default is **nil**.

sys:%%region-scavenge-enable
Lets you override the scavenge-enable bit explicitly. This is an internal flag related to the garbage collector. Do not try to use it! The default is **nil**.

Example:

```
(make-area :name foo-area
           :gc :dynamic
           :representation :list)
```

describe-area *area* *Function*

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

area-list *Variable*

The value of **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.

%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.

%region-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. (This information is generally not very interesting to users; it is important only inside the system.)

area-name *area* *Function*

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

See the function **cons-in-area** in *Reference Guide to Symbolics-Lisp*. See the function **list-in-area** in *Reference Guide to Symbolics-Lisp*. See the function **room**, page 129.

15.2 Interesting Areas

This section lists the names of some of the areas and tells what they are for. Only the ones of the most interest to a user are listed; there are many others.

working-storage-area *Variable*

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

permanent-storage-area *Variable*

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

pname-area *Variable*

Print-names of symbols are stored in this area.

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 **intern** keeps track of symbols.

compiled-function-area *Variable*

Compiled functions are put here by the compiler.

property-list-area *Variable*

This area holds the property lists of symbols.

constants-area *Variable*

This area contains constants used by compiled programs.

15.3 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.

15.4 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.

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.

15.4.1 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 (**si: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.

15.4.2 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 **compiled-function-area**:

```
(defun print-compiled-function-names ()
  (si:map-over-regions-of-area
    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
    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 each 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**, page 109.

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* *Macro*

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 user 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* *Macro*

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**, page 111.)

16. The Garbage Collector

16.1 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 Lisp writing.) It is possible, even considering the large amount of virtual memory on a Symbolics computer, for a program to use up all the virtual memory available, at which point the machine halts and must be rebooted. This event can always be delayed, perhaps almost indefinitely, if the underlying system can reclaim memory originally allocated for objects that are now unused.

Such objects, those with no references from other objects, are termed *garbage*, since they no longer serve any purpose in the current Lisp world and merely take up otherwise useful space. 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, although it persists in the Lisp world, is garbage. The garbage metaphor is extended in several ways in Lisp literature. For example, a *scavenger* process periodically sifts through areas of memory, separating good objects from the garbage. The large-scale operation, which involves scavenging virtual memory, moving good objects to a safe place, and reclaiming the memory occupied by garbage, is called *garbage collection*.

There are several strategies for using garbage collection, some that allow you to continue doing other work and some that do a more complete job but require additional machine resources for some period of time. It is worth noting, too, that garbage collection need not be used at all. Garbage collection should be used when you either 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 preserving the state of an operating Lisp world as long as possible and avoiding a cold boot.

If you would like to preserve the state of your machine as long as possible, with the least effect on performance, you should at least run with the ephemeral-object garbage collector turned on. (See the section "Ephemeral-object Garbage Collection", page 119.) You can turn it on with the **gc-on** or **choose-gc-parameters** functions or with the Start GC command. However, absolutely maximum performance is usually achieved by running with no garbage collection at all, although the machine will probably run out of virtual memory much faster.

There are two 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 in any process 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 areas of memory (*static* areas) that change slowly and so are unlikely to contain garbage.

- *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 **gc-immediately** function, although that variation still ignores static areas; this function allows 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.

16.2 Using the Garbage Collector

If you want to take the easiest advantage of automatic storage management, to preserve the virtual memory of your machine as long as possible, you should run with both the ephemeral- and dynamic-object garbage collectors turned on. Both are turned off by default, but both can be turned on by evaluating **gc-on** with no arguments or entering the command Start GC.

The garbage collector is a program in the Symbolics-Lisp system that automatically finds, tracks, and recovers memory occupied by unused objects (garbage) in the current Lisp world. It is a particular implementation of *automatic storage management*, meaning that programmers (and also nonprogrammer users of the system) can do things that allocate, use, and discard large amounts of virtual memory, without having to pay any attention to the management of the memory. In systems without this feature, most large-scale uses of virtual memory have to be managed "manually" (under control of a user program); manual storage management is difficult and error-prone because it is quite difficult for a program to "prove" that an object really is of no use to any other system component.

Automatic storage management also has the desirable effect of lengthening the "session" you spend with a particular world between cold boots. Without it, most normal uses of a Lisp system will exhaust virtual memory rather quickly. With it, normal use (whether or not for programming) is longer and more convenient.

When the usual, incremental garbage collector is operating, the Scavenger periodically goes through virtual memory, looking for objects that can be proven not to be garbage. These "good" objects are transported to a safe place, and the memory occupied by the garbage is reclaimed automatically. In the meantime, new objects can still be created. (More extensive information on automatic storage management is available elsewhere; See the section "Operation of the Garbage Collector", page 117.)

There are different kinds of garbage collection available in Symbolics-Lisp. All require some additional virtual memory for their own use. Until the scavenging process is complete, running with the garbage collector can require up to twice as much space as running without the 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 approach a certain percentage full. See the section "Storage Requirements for Garbage Collection", page 122.

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 "Allocating Extra Paging Space" in *Reference Guide to Streams, Files, and I/O.*

Garbage collection can be optimized for particular applications by manipulating areas and their attributes. See the section "Areas", page 103.

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 **working-storage-area**, for example.

choose-gc-parameters *Function*

The function **choose-gc-parameters** 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.

gc-on &key *ephemeral dynamic* *Function*

Turns garbage collection on. It 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 **:ephemeral** or **:dynamic** is specified without a value, the default is **nil**; this allows you to turn off one form of garbage collection and leave the other one on.

gc-off *Function*

Turns garbage collection off.

gc-on *Variable*

The value of this variable is non-**nil** when the garbage collector is turned on and **nil** when it is turned off. **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).

gc-immediately &optional *no-query* *Function*

gc-immediately does nonincremental garbage collection, taking less space and less total time than an incremental gc, but running continuously in the process calling it, until the garbage collection is complete. The main advantage of this 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.

You should usually call this rather than **si:full-gc**. The difference is that **gc-immediately** does not lock out other processes, does not run various **full-gc** initializations, and does not affect the static areas.

Suppose garbage collection has already started, 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:full-gc &key *system-release gc-compiled-functions* *Function*

The function **si:full-gc** garbage-collects the entire Lisp environment, including some static areas (those on the list bound to **si:full-gc-static-areas**). However, because static areas change slowly and are not likely to contain much garbage, you should not normally need this function to perform nonincremental garbage collection; use **gc-immediately** instead. Call **si:full-gc** with no arguments if you must use it.

The options **:gc-compiled-functions** and **:system-release** are reserved for use by Symbolics.

si:full-gc does an immediate, complete, nonincremental garbage collection. Two initialization lists, accessed through the **full-gc** and **after-full-gc** keywords to **add-initialization**, are run by **si:full-gc**. It runs the forms on the **full-gc** initialization list and then does garbage collection without multiprocessing (inside a **without-interrupts** form), so 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 passing 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 and, thus, is really garbage and will be collected.

16.3 Operation of the Garbage Collector

There are three agents involved in automatic storage management, or 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 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 tracing through the objects in the root set and the objects they reference. This program is called the *scavenger*. It runs during consing, during idle time, and (in the case of nonincremental garbage collection) in the user or garbage collector process.

- A 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.) This program, the *transporter*, redirects its client to copy space in either case.

The garbage collector treats the machine's virtual memory as if it were divided into two spaces: *dynamic* space and *static* space. 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", page 103.

Static space The parts of memory in which relatively permanent objects are allocated are collectively called static space. Objects allocated in these 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 *new*, *old*, and *copy* spaces. (In addition, ephemeral *levels* are part of dynamic space; **See the section "Ephemeral-object Garbage Collection", page 119.**)

New space New space is the portion of dynamic space in which new objects are allocated. In a pristine system, all objects are allocated here; neither of the other two spaces exists until the first garbage collection operation (scavenging) begins.

Old space Old space is the portion of dynamic space that is created from the previous new and copy spaces and may still contain valid objects. (That is, the scavenger is actually looking for good objects here by perusing references in the current static and copy spaces.) When the scavenger is finished, everything in old space is garbage.

Copy space Copy space is the portion of dynamic space to which the transporter moves good objects found in old space.

When it is time to collect garbage, the spaces are *flipped*:

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.

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 and becomes available for assignment to new space. Another flip can occur any time after old space has been reclaimed.

The incremental garbage collector decides to flip when it estimates that it will require a large portion of the remaining free virtual memory 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", page 122.

16.3.1 Ephemeral-object Garbage Collection

Ephemeral-object garbage collection is a method by which the scavenger agents can pay special attention to short-lived, or ephemeral, objects. It is effective on any area having the **:gc :ephemeral** characteristic as specified by **make-area**. The **working-storage-area** has the ephemeral characteristic by default; since it 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 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 the "dynamic" method, which operates on all of dynamic space rather than just the ephemeral portion of it. They also mean that when the dynamic (nonephemeral) objects are eventually garbage-collected, dynamic space contains less garbage than would otherwise be the case.

The ephemeral-object feature introduces the concept of ephemeral *levels*, subdivisions of a particular area. 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): 2 regions, capacity 196K, 416K allocated, 122K used.
  Second ephemeral level (#1): 3 regions, capacity 98K, 336K allocated, 148K used.
  Last (dynamic) level (#0): 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 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. (Note: Do not be confused by the parenthesized numbers attached to ephemeral levels; they are used internally by the software. "First" means first, even if its so-called level number is 2.)

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 as usual for dynamic areas. It has no capacity in the sense of an ephemeral level because the decision to flip is based on different principles. See the section "Storage Requirements for Garbage Collection", page 122.

The advantage 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.

The output of the function **gc-status** or the command Show GC Status includes one line for each ephemeral level that exists.

By default, **gc-on** or the Start GC command enables the ephemeral collector along with dynamic-object garbage collection. The area **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 by experimenting with the following features:

- Using the function **choose-gc-parameters**, 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, **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 may be possible to extend the length of a session considerably, by adding additional ephemeral levels.

16.3.2 Locality of Reference

Locality of reference is a desirable property of programs that run on virtual memory systems like Symbolics-Lisp. 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 Symbolics-Lisp 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.

The operation of garbage collection in this system improves locality of reference by itself, including in the **working-storage-area**.

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. It 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.

4. By the time the scavenger has finished scanning the last page of copy space, it

has either found no old-space references (in which case all of old space is garbage and can be immediately reclaimed) or it has found them and has evacuated the corresponding object into copy space.

The effect is that object references and the corresponding objects tend to fall on the same page in virtual memory.

16.4 Storage Requirements for Garbage Collection

The output of the Show GC Status command (or **gc-status** function) shows the storage requirement for incremental, dynamic garbage collection, in the form of a "committed guess." (This section is not related to the storage requirements for ephemeral-object garbage collection.) For example, suppose the command reports the following information:

```
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.
```

The "free space" is the total amount of unused space allocated to paging on the local disk(s) and is, in fact, the amount available for new objects if the garbage collector is turned off. 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 while the garbage collection is going on. It is quite accurate for compute-bound programs, on which most of the underlying assumptions are based. For interactive programs, it is somewhat conservative because the garbage collector runs during idle time and so finishes more quickly.

The computation goes as follows, assuming that **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), oldspace 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 $x = 6.17$.

The primary reason that a nonincremental garbage collection (as by **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" computation 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.

Now consider the additional 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.

16.5 Controlling Garbage Collection

gc-status *Function*

gc-status 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.

```
(gc-status)

Status of the ephemeral garbage collector:                On
First level of WORKING-STORAGE-AREA: capacity 196K, 416K allocated, 10K
used.
Second level of WORKING-STORAGE-AREA: capacity 98K, 256K allocated, 137K
used.

Status of the dynamic garbage collector:                  On
Dynamic (new+copy) space 1,746,767.  Old space 0.  Static space 6,856,801.
Free space 6,957,056.  Committed guess 6,559,133, leaving 135,779 to use
  before flipping.
There are 2,343,001 words available before (GC-IMMEDIATELY) might run out
of space.
Doing (GC-IMMEDIATELY) now would take roughly 14 minutes.
There are 6,957,056 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 2 (1 full GC, 0 dynamic GC's, and 1 ephemeral
GC).
Evaluate (CHOOSE-GC-PARAMETERS) to examine or modify the GC parameters.
```

In the **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 **gc-off** function) and still have almost 7 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:inhibit-gc-flips *body ...* *Macro*

si:inhibit-gc-flips prevents the garbage collector from flipping within the body of the macro.

The following variables' values control various aspects of the garbage collector's operation; all are accessible via the **choose-gc-parameters** function.

si:gc-report-stream *Variable*

si:gc-report-stream 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*

si:gc-area-reclaim-report 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*

si:gc-warning-threshold controls the warnings to turn on the garbage collector. When the storage manager notices that the amount of free space remaining before it would be too late to garbage collect has reached the threshold, it notifies you that you need to turn on the garbage collector. The default value is 1000000.

si:gc-warning-ratio *Variable*

si:gc-warning-ratio 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.

si:gc-warning-interval *Variable*

This 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 default value is 18000.

si:gc-flip-ratio *Variable*

si:gc-flip-ratio specifies when a flip takes place. When this number times the amount of committed free space (the "committed guess" reported by **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**, page 126.

si:gc-flip-minimum-ratio *Variable*

si:gc-flip-minimum-ratio 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 immediate garbage collection is performed as soon as the capacity of the first ephemeral level is exceeded.

si:gc-reclaim-immediately-if-necessary *Variable*

si:gc-reclaim-immediately-if-necessary 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 real (nongarbage) objects. (This is the value of **si:gc-flip-minimum-ratio**.)

si:gc-process-immediate-reclaim-priority *Variable*

This 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 **choose-gc-parameters**. Note: This variable is not related to the **gc-immediately** function nor to the :Immediate option of the Start GC command.

si:gc-process-foreground-priority *Variable*

This variable provides the process priority for the garbage collector while it is waiting to flip. 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*

si:gc-flip-inhibit-time-until-warning 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 10 seconds. Flipping cannot occur when some program (such as **maphash**) is running in an **si:inhibit-gc-flips** special form.

16.6 Strategy for Unattended Operation with the Garbage Collector

It is chancy 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 it makes about the amount of space needed or available. See the section "Controlling Garbage Collection", page 124.

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 "Operation of the Garbage Collector", page 117. See the section "Principles of Garbage Collection", page 113.

Some people find it necessary to have garbage collection working in order to load large systems. The following strategies are recommended:

- Before loading the system, turn on ephemeral-object garbage collection with the form `(gc-on :ephemeral t)` or with the command `Start GC :Ephemeral`.

- After loading the system, do an immediate garbage collection with the function **gc-immediately** or with the command `Start 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.

17. 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*

Tells you the amount of physical memory on 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.

(room t) tells you about all the areas.

(room nil) 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.

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)
```

18. Resources

Storage allocation is handled differently by different computer systems. In many languages, you must spend a lot of time thinking about when variables and storage units are allocated and deallocated. In Lisp, 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.

However, automatic freeing of storage incurs an expense: more computer resources must be devoted to the garbage collector. If a program is designed to allocate temporary storage, which is then left as garbage, more of the computer must be devoted to the collection of garbage; 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 order to prevent a great deal of overhead from the garbage collector.

It is usually not worth worrying about 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 within the Symbolics computer system are in this position. The Chaosnet software allocates and frees "packets", which are moderately large, at a very high rate. The window system allocates and frees certain kinds of windows, which are very large, moderately often. Both of these programs manage their objects manually, keeping track of when they are no longer used.

When we say that a program "manually 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 in this section perform the above function. The set of objects forming each such list is called a "resource"; for example, there might be a Chaosnet packet 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", page 26. See the section "The Data Stack", page 28.

defresource *name parameters* &rest *options* *Special Form*

The **defresource** special form is used to define 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 "values" of options are numbers, names, or (often) forms that follow the option keyword in a call. 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.

: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 **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. 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. In addition to the parameters, a form here can access the variable **object** (in the current package). 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 contains 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 size of the array that the resource uses to remember the objects it allocates and deallocates.

If these options are used with forms (rather than functions), the forms get compiled into functions as part of the expansion of **defresource**. These functions are given names like
(:property *resource-name* **si:resource-constructor)**; these names may change in the future.

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. Furthermore, 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:

```
(defresource sloppy-two-dimensional-array (rows columns)
    :constructor (make-array (list rows columns))
    :matcher (and (≥ (array-dimension-n 1 object) rows)
                  (≥ (array-dimension-n 2 object) 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 (si:fill-array object nil nil)
  :deinitializer (si:fill-array object nil 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.

Note that the **using-resource** special form is usually what you want to use, rather than **allocate-resource** itself.

deallocate-resource *resource-name object* *Function*

Frees the object *resource-name*, returning it to the free-object list of the resource specified by *object.*

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...* *Special Form*

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**.

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.

Here is an example of the use of resources:

```
(defresource huge-16b-array (&optional (size 1000))
  :constructor (make-array size :type 'art-16b))

(defun do-complex-computation (x y)
  (using-resource (temp-array huge-16b-array)
    ...                               ;Within the body, the array can be used
    (aset 5 temp-array i)
    ...))                             ;The array is returned at the end
```

Index

D D D

G

H H H

I

M M M

N

O

P

S S S