ACCENT LISP MANUAL

PREFACE

This manual contains documentation on Accent Lisp, an implementation of the Common Lisp language for the PERQ Model LN-3500 workstation. Accent Lisp runs under the Accent operating system, which was developed jointly by PERQ Systems Corporation and the Spice Project in the Computer Science Department at Carnegie-Mellon University. "Spice" is an acronym for Scientific Personal Integrated Computing Environment.

Lisp is a programming language widely used for Artificial Intelligence research. It was invented by John McCarthy in 1958. Because of its built-in facilities for symbol-processing and its interactive programming environment, the language is increasingly being used for such applications as compilers, CAD systems, and editors. Common Lisp is a new dialect of Lisp, closely related to Maclisp, Lisp Machine Lisp (Zetalisp), and (somewhat less closely) Franz Lisp. It was developed jointly by several Lisp groups to meet the need for a modern Lisp dialect that is stable, well-documented, and suitable for implementation on a variety of machines.

Hemlock is an editor written in Common Lisp. It is a descendant of EMACS, an editor written by Richard M. Stallman of the Massachusetts Institute of Technology. If in your use of Accent Lisp you implement any additional editing commands, we would appreciate your forwarding them to us.

Before you use this manual you should be familiar with the material in the *Accent User's Manual.* Other manuals for Accent are:

- Accent Programming Manual Accent Microprogramming Manual Accent Languages Manual Accent Qnix Manual (forthcoming in a later release) Accent System Administration Manual

Throughout the Accent manuals the term "PERQ" refers to all models of the PERQ workstation unless stated otherwise. When a distinction is made between the PERQ workstation and the PERQ2 workstation, the term "PERQ2" refers to both Model LN-3000 and LN-3500.

The following symbols have been used throughout the Accent manuals:

< >	Material that is to be replaced by symbols or text as explained in the accompanying text. Do not type the angle brackets. Example: <filename> indicates that you should type the name of your file.
[ ]	Optional feature. Do not type the square brackets.
{ }	0 to n repetitions of an optional item. Do not type

the braces.

CAPITALS Literal, to be reproduced exactly as shown (although it may be reproduced in upper-case or lower-case). Example: <filename.CMD> indicates that the filename must contain the extension .cmd.

| "Or"--choice between the items shown on either side of the symbol.

CTRL Control key

ESC, INS Escape key (labeled as ACC ESC or INS on various models)

DEL Delete key (labeled as REJ DEL on some models)

HELP Help key

LF Linefeed Key

RETURN Carriage return

italics input to be typed by the user

ACCENT LISP USER'S GUIDE

August 1, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Table of Contents Page

1. Introduction

Common Lisp is a new dialect of Lisp that is closely related to Maclisp and Lisp Machine Lisp. Common Lisp was developed in response to the need for a modern, stable, well-documented Lisp dialect that can be implemented efficiently on a variety of machine architectures. Accent Lisp is PERQ Systems' implementation of Common Lisp for microcodable personal machines running PERQ's Accent operating system.

The primary document for users of any Common Lisp implementation is *Common Lisp* by Guy L. Steele Jr. (Digital Press, 1984). All implementations of Common Lisp conform to this standard. However, different implementations are free to make a number of design choices and to add to the basic Common Lisp facilities. This document covers the choices and features that are specific to the Accent Lisp implementation. This document and *Common Lisp*, taken together, provide all information needed by the user.

1.1. Loading Accent Lisp

Accent Lisp will run on a PERQ1a or PERQ2 workstation with 16k control store and the Accent operating system. A partition with at least 5500 free pages is needed to contain the Accent Lisp files. A paging partition with at least 10,000 pages of free space is also recommended.

Accent Lisp is supplied on floppy disks labelled "Accent Lisp." To install Accent Lisp:

1. Insert the floppy disk labelled "Accent Lisp 1" into the floppy drive.

2. Path to the partition that will contain the Accent Lisp files, and type:

   Floppy @AddL1

3. Path to the paging partition, and type:

   Floppy @AddLisp

   @AppSLisp

4. Answer any questions as they appear on the workstation screen. Default responses are provided.

Accent Lisp is loaded to your system and ready to use.

When Accent Lisp is on your workstation, put the directory that it resides in on your search list. Type *Lisp* to the Accent shell to run Accent Lisp.

1.2. Startup Switches

The *Lisp* call recognizes a switch that tells it which "core" file to load. A suspended Lisp system is saved in a core file (see section 2.8) that can be restarted using the -Core switch to Lisp:

lisp -core=<filename>

If this switch is omitted, the file "lisp.core" is used.

When Lisp is starting up, it also looks for the following switches and takes the specified actions in the order that the switches appear.

-Eval=string
: One Lisp object will be read from String and evaluated.

-Edit
: Starts the editor. If there are any input words, the first is taken as a filename, and that file is visited.

-Load=filename
: The named file will be loaded into Lisp.

The syntax of the -Edit switch allows the user to define a Hemlock command with a shell alias:

alias hemlock 'run lisp -edit'

This causes the command

hemlock <filename>

to edit the specified file.

1.3. Accent Lisp Initialization

When Accent Lisp is started up, the following global variables, ports, and streams are initialized. This initialization takes place whether the default Lisp.core or another core file is started.

1.3.1. Command line variables and functions

command-line-words [Variable]

A list of simple strings parsed from the command line by the shell. The first word is the name used to invoke Lisp (usually "Lisp"), and the following words are either switches, switch values, inputs, outputs, or the special word consisting of the tilda ("~") character.

A word with a minus sign as its first character is a switch name. If the last character of a switch name is an equals sign, then the word immediately following the switch name is the switch value.

Words that are not the utility name, switch names, or switch values must be either inputs or outputs. If no tilda is present, then all such words are inputs. Otherwise, the words preceding the tilda are inputs, and the words following the tilda are outputs.

command-line-utility-name [Variable]

This variable is initially set to:

```
(car *command-line-words*)
```

command-line-inputs [Variable]

A list of simple strings consisting of the input words from *Command-line-words*, in the same order.

command-line-outputs [Variable]

A list of simple strings consisting of the output words from *Command-line-words*, in the same order.

command-line-switches [Variable]

A list of command-line-switch objects created from the switch names and switch values in *Command-line-words*.

Command-line-switch objects may be examined with the following functions:

cmd-switch-name
command-line-switch [Function]

Returns the string name of a command-line-switch object. The leading minus sign and ending equals sign are stripped off.

cmd-switch-arg
command-line-switch [Function]

Returns the string argument of a command

line switch, or Nil if the switch has no argument.

cmd-corresponding-arg
command-line-switch [Function]

Returns the input word or output word that appeared most recently before the switch on the command line, or Nil if no inputs or outputs preceded the switch. If the result is not Nil, then it is an element of either *Command-line-inputs* or *Command-line-outputs*.

1.3.2. Port initializations

TimePort [Variable]

Port for time server requests.

SesPort [Variable]

Port for file system requests.

EMPort [Variable]

Port for environment manager requests.

PMPort [Variable]

Port for process manager requests.

NameServerPort [Variable]

Port for name server requests.

UserTypescript [Variable]

Port to use for operations on the process' given typescript. The standard stream *Terminal-io* does its input and output through *UserTypescript*. Changing the value of *UserTypescript* to an object that is not a working typescript usually causes an unrecoverable error.

UserWindow [Variable]

Port to use for operations on the process' given window.

UserWindowShared [Variable]

Boolean. If true, then *UserWindow* is owned by another process that probably will use the same window after Lisp is terminated. If Nil, then *UserWindow* is owned by this process and will be destroyed when Lisp terminates.

TypescriptPort [Variable]

Port to use for creating new typescripts. This port is not a typescript.

SapphPort [Variable]

Port to use for operations on the full screen.

This should only be used for creating new top level windows.

1.3.3. Stream initializations

These variables are all defined in *Common Lisp*.

terminal-io [Variable]

Initially set to a two-way stream that does its input and output through *UserTypescript*. *Terminal-io* never reaches end of file, and it may not be closed.

standard-input [Variable]

Holds the default stream for all input functions. It is usually a synonym stream for *Terminal-io*.

standard-output [Variable]

Holds the default stream for all output functions. It is usually a synonym stream for *Terminal-io*.

error-output [Variable]
trace-output [Variable]

Initially synonym streams to *Standard-output*.

query-io [Variable]

debug-io [Variable]

Initially two-way streams made from *Standard-output* and *Standard-input*.

standard-input-filename [Variable]

If input was redirected at startup, this variable holds the string filename of the input file. Otherwise, it holds Nil.

standard-output-filename [Variable]

If output was redirected at startup, this variable holds the string filename of the output file. Otherwise, it holds Nil.

2. Accent Lisp Specific Information

This chapter describes Accent Lisp. Specifically, it discusses differences and additions to Common Lisp that are implemented in Accent Lisp.

2.1. Numbers

Currently, short-floats and single-floats are the same, and long-floats and double-floats are the same. Short floats use an immediate representation with 8 bits of exponent and a 21-bit mantissa. Long floats are 64-bit allocated objects, with 12 bits of exponent and 53 bits of mantissa. All of these figures include the sign bit and, for the mantissa, the "hidden bit." The long-float representation conforms to the 64-bit IEEE standard, except that all the exceptions, negative 0, and infinities are not supported.

Fixnums are stored as 28-bit two's complement integers, including the sign bit. The most positive fixnum is 2^27 - 1, and the most negative fixnum is -2^27. An integer outside of this range is a bignum.

2.2. Characters

Accent Lisp characters have eight bits of code, eight font bits, and eight control bits. Of the font and control bits, only four control bits are used: control, meta, super, and hyper.

The control bit functions Control, Meta, Super, and Hyper are defined as in *Common Lisp*. The PERQ

workstation keyboard does not produce these and the Accent operating system does not pass them to Accent Lisp, but programs can use them internally.

2.3. Vector Initialization

If no :Initial-value is specified, vectors of Lisp objects are initialized to Nil, and vectors of integers are initialized to zero.

2.4. Packages

Common Lisp requires four built-in packages: Lisp, User, Keyword, and System. In addition to these, Accent Lisp has separate packages for Hemlock and the compiler, and a Hemlock-internals package.

2.5. The Editor

The Ed function invokes the Hemlock Editor. Hemlock is described in *The Hemlock User's Guide* and *The Hemlock Command Implementor's Reference* in this manual; like Accent Lisp, it contains easily accessible internal documentation.

2.6. Garbage Collection

Accent Lisp automatically does a garbage collection (GC) whenever a specified ratio of memory space in dynamic space to available virtual memory is exceeded. This ratio defaults to 2, meaning that a GC will be done when the unused virtual memory is about half the size of Lisp's dynamic space. To change the ratio, set lisp::gc-flip-ratio to a smaller value like 1 or 3/2. To turn off the GC entirely, Setq lisp::*already-maybe-gcing* to T.

2.7. Timing

There is one timing function in Accent Lisp.

time <form> [Macro]

Evaluates Form and prints the total elapsed time for the evaluation to *Trace-output*. Time returns the result of the evaluation of Form.

2.8. Saving a Core Image

Accent Lisp provides a way to save a core image that has code loaded into it.

save <file> &key

<checksum> [Function]

Saves an image of the current process on file File. If Checksum is T, the corefile will have a checksum entry; if Nil, no checksum entry is generated. Checksum defaults to T. Save returns the number of bytes written to the file. After Save is called, the current Lisp is restarted and continues in the function from which Save was called. It is an error to use this function when there are open files.

When Save is called, Lisp is frozen *in* the call to save. The saved Lisp will start up by returning from the Save function and continuing in the function from

which Save was called. Ports lose their validity when Save is called, but the standard port variables (see 1.3.2) are reinitialized when Lisp is restarted. Any user-defined ports will have to be redefined in order to be used.

The core image can be restored to Lisp by typing:

lisp -core=<corefilename>

This will start Lisp with the suspended process that was stored by the Save command (see 1.2).

3. Debugging Tools

3.1. Function Tracing

The tracer causes selected functions to print their arguments and results whenever they are called. Options allow conditional printing of the trace information and conditional breakpoints on function entry.

trace &rest specs [Macro]

Invokes tracing on the specified functions, and pushes the names onto the global list in *Traced-function-list*. Each spec is either the name of a function, or the form

```
(function-name
   trace-option-name value
   trace-option-name value
   ...)
```

If no specs are given, then trace will return the list of all currently traced functions, *Traced-function-list*.

If a function is traced with no options, then each time it is called, a single line containing the name of the function, the arguments to

the call, and the depth of the call will be printed on the stream *Trace-output*. After the function returns, another line is printed containing the depth of the call and all of the return values. Call depth is highlighted by indenting the lines.

Trace options change the normal output from the trace. Each traced function has its own set of options independent from any other function. Each time a function is specified in a call to Trace, any options have to be specified also; no options are kept from previous specifications. The available options and their values are:

:Condition :Condition's value is a form that is evaluated before each call to the function. Printout is suppressed when the form returns Nil.

:Break Its argument is a form that is evaluated before each call to the function; if the form returns non-Nil, a breakpoint loop is entered immediately before the function call.

:Break-after
Like :Break, but the form is evaluated and the break loop invoked after the function call.

:Break-all Combines :Break and :Break-after.

:Wherein Its value is a function name or list of function names. Printout for the traced

function only occurs when it is called from within a call to one of the functions listed in :Wherein's value.

:Print — Its value is a list of forms to be evaluated and printed whenever the function is called. The values are printed one per line, and indented to match the other trace output. This printout is suppressed whenever the normal trace printout is suppressed.

:Print-after — Like :Print except that the values of the forms are printed whenever the function exits.

:Print-all — The combination of :Print and :Print-after.

untrace &rest function-names [Macro]

Turns off tracing for the specified functions, and removes their names from *Traced-function-list*. If no function-names are given, then all functions named in *Traced-function-list* will be untraced.

traced-function-list [Variable]

A list of function names that is maintained and used by Trace and Untrace. This list should contain the names of all functions which are currently being traced.

trace-print-level [Variable]
trace-print-length [Variable]

Print-level and *Print-length* are bound to *Trace-print-level* and *Trace-print-length* when printing trace output. The forms printed by the :Print options are also affected. *Trace-print-level* and *Trace-print-length* are initially set to Nil.

max-trace-indentation [Variable]

The maximum number of spaces that should be used to indent trace printout. This variable is initially set to some reasonable value.

3.2. The Single Stepper

The single stepper is a mechanism that allows the user to control calls to Eval. When single stepping, each call to Eval first prompts for a command.

step <form> [Function]

Evaluates Form with single stepping enabled or, if Form is T, enables stepping on until explicitly disabled. Stepping can be disabled by quitting to the Lisp top level, or by evaluating the Form (step ()).

While stepping is enabled, every call to Eval will prompt the user for a single character

command. The prompt is the form which is about to be evaluated. It is printed with *Print-level* and *Print-length* bound to *Step-print-level* and *Step-print-length*. All interaction is done through the stream *Query-io*.

The commands are:

n (next)	Evaluates the expression with stepping still enabled.
s (skip)	Evaluates the expression with stepping disabled.
q (quit)	Evaluates the expression, but disable all further stepping inside the current call to step.
p (print)	Prints current form. This does not use *Step-print-level* or *Step-print-length*.
b (break)	Enters break loop, and then prompts for a command again when the break loop returns.
e (eval)	Prompts for and evaluate an arbitrary expression. The expression is evaluated with stepping disabled.
? (help)	Prints a brief list of the commands.
r (return)	Prompts for an arbitrary value to return as result of the current call to evaluate.
g	Throws to top level.

step-print-level [Variable]
step-print-length [Variable]

Print-level and *Print-length* are bound to these values when the current form is printed. *Step-print-level* and *Step-print-length* are initially bound to some small value.

max-step-indentation [Variable]

Step indents the prompts to highlight the nesting of the evaluation. This variable contains the maximum number of spaces to use for indenting. It is initially set to some reasonable number.

3.3. The Debugger

The debugger is an interactive command loop that allows examination of the active call frames on the Lisp function call stack. If invoked from an error breakpoint, it can show the function calls that led up to the error.

Inside the debugger, most commands refer to the current stack frame. The debugger assigns numbers to the frames on the stack, starting with zero as the most recent and increasing deeper into the stack. The debug prompt includes the number of the current frame as its main feature.

Most expressions typed to debug are simply evaluated as they would have been had you not entered debug.

This includes the special debugger functions to be described, which are meaningful only inside the debugger. The biggest exceptions are the debugger commands, which are either one or two letters. These may display information about the current frame or change the current frame, but they generally do not affect the evaluation history. All debugger commands are case insensitive.

3.3.1. Movement commands

These commands move to a new stack frame, and print out the name of the function and the values of its arguments in the style of a Lisp function call. Frames that are not active are marked with a "*", and the reconstructed call consists of what arguments are present on the stack. *Debug-print-level* and *Debug-print-length* affect the style of the printing.

Visible frames are those that have not been hidden by the Debug-hide function described below. The special variable *Debug-ignored-functions* contains a list of function names that are hidden by default.

u Moves up to the next higher visible frame. More recent function calls are considered to be higher on the stack.

d Moves down to the next lower visible frame.

t Moves to the highest visible frame.

b Moves to the lowest visible frame.

f Moves to a given frame, visible or not. Prompts for the number.

3.3.2. Inspection commands

These commands print information about the current frame and the current function.

?	Describes the current function.
a	Lists the arguments to the current function. The values of the arguments are printed along with the argument names.
l	Lists the local variables in the current function. The values of the locals are printed, but their names are no longer available.
p	Redisplays the current function call as it would be displayed by moving to this frame.
pp	Redisplays the current function call using *Print-level* and *Print-length* instead of *Debug-print-level* and *Debug-print-length*.

debug-value symbol

&optional [frame] [Function]

Returns the value of Symbol, considered as a special variable, in the binding context of either the current frame, or the numbered frame, if specified.

debug-local n

&optional [frame] [Function]

Returns the value of the Nth local variable in the current or specified frame.

debug-arg n &optional [frame] [Function]

Returns the Nth argument of the frame.

debug-pc &optional [frame] [Function]

Returns the next instruction to be executed in the specified (active) frame. Can be used with Disassemble.

3.3.3. Other commands

h Prints a brief but comprehensive list of commands on the terminal.

q Causes debug to return Nil.

debug-return expression

&optional [frame] [Function]

Forces the current function to return zero or more values. If the function was not called for multiple values, only the first value will be returned.

backtrace [Function]

Prints a history of function calls. The printing is controlled by *Debug-print-level* and *Debug-print-length*. Only those

frames that are considered visible by the frame movement commands will be shown.

debug-hide option

&optional [arg(s)] [Function]

Makes the described stack frames invisible to the frame movement commands. The second argument is evaluated and may be a symbol or a list; the function returns the hidden members of the category. With no arguments, returns the current filter (hidden frames). Option subcommands may be one of these:

Package(s) Calls to hidden packages are visible, but calls within them are not.

Function(s) Calls to the named functions will not be visible.

Type(s) Hides miscellaneous frame and function types, any of:

- *Compiled* - Calls to compiled functions will not be visible.
- *Interpreted* - Calls to interpreted functions will not be visible.
- *Lambdas* - Calls to lambda expressions will not be visible.

open Open frames will not be visible.

active Active frames will not be visible.

catch Catch frames will not be visible.

debug-show options

&optional [arg(s)] [Function]

Cancels the effect of the corresponding Debug-hide. Note that a frame may be hidden in a variety of other ways, though.

debug [Function]

Invokes the debugger. Debug always returns Nil.

debug-print-level [Variable]
debug-print-length [Variable]

Print-level and *Print-length* are bound to these values during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal *Print-level* and *Print-length* are in effect. These variables are initially set to some small number.

debug-ignored-functions [Variable]

A list of functions that are hidden by default. These functions can be made visible with the debug command Show.

3.4. Break Loop

The break loop is a read-eval-print loop similar to the normal Lisp top level. It can be called from any Lisp function to allow interaction with the Lisp system. When giving the command to exit the break loop, choose an arbitrary value for the loop to return.

When a Lisp expression is typed in at the break loop's prompt, it is usually evaluated and printed. However, there are three special expressions that are recognized as break loop commands, and that are not evaluated. These commands are case insensitive.

$G causes a throw to the Lisp top level: The current computation is aborted, and all bindings are unwound.

$P causes the break loop to return Nil.

RETURN form
causes the break loop to evaluate form and return the result(s).

The dollar sign character in the symbols $P and $G is intended to be the <escape> character (ascii 27). However, typing the dollar sign is also understood.

When the break loop is called, it tries to make sure that terminal interaction will be possible. All of the standard input output streams, *Standard-input*, *Standard-output*, *Error-output*, *Query-io*, and *Trace-output* are bound to *Terminal-io* for the duration of the break loop; and the state of the single stepper is bound to "off".

break format-string

&rest args [Function]

The Break function passes Format-string and Args to Format, and then invokes the break loop.

3.5. Cleaning Up

The break loop is called by the system error handlers. Since errors can happen unexpectedly, the break loop provides a mechanism for cleaning up any unusual state that a program may have caused.

error-cleanup-forms [Variable]

A list of Lisp forms will be evaluated for side effects when a break loop is invoked. Whenever a break loop is entered, *Error-cleanup-forms* will be bound to Nil, and then the forms that were its previous value will be evaluated for side effects. There is no way to have the side effects undone when the break loop returns, and if any of the cleanup forms causes an error, the result cannot be guaranteed.

As an example, a program that puts the terminal in an unusual mode might want to do something like this:

```
(let ((*error-cleanup-forms*
       (cons '(progn <code to restore terminal>)
     *error-cleanup-forms*)))
  <code to mess up terminal>]
    ...)
```

4. The Compiler

Functions may be compiled using Compile, Compile-file, or Compile-from-stream. Compile and Compile-file operate as documented in *Common Lisp*.

The Compile-file function takes the following keyword arguments: :Output-file, :Lap-file, and :Error-file. These keywords accept either the name of a file as a string, or the symbol T, which causes an appropriate filename to be created by replacing the type field of the input filename. If the argument to any of these keywords is Nil, no file of that type is created. If no keywords are specified, the output file and error file are created by default.

Compile-from-stream is like Compile-file, but it takes a stream as its only argument and compiles the code read from that stream into the current environment.

4.1. Open and Closed Coding

When a function call is "open coded," inline code whose effect is equivalent to the function call is substituted for that function call. When a function call is "closed coded," it is usually left as is, although it might be turned into a call to a different function with different arguments. As an example, if Nthcdr were to be "open coded," then

```
(nthcdr 4 foobar)
```

might turn into

```
(cdr (cdr (cdr (cdr foobar))))
```

or even

```
(do ((i 0 (1+ i))
     (list foobar (cdr foobar)))
    ((= i 4) list)).
```

If Nth is "closed coded,"

```
(nth x 1)
```

might stay the same, or turn into something like:

```
(car (nthcdr x 1))
```

4.2. Compiler Switches

Several compiler switches are available that are not documented in *Common Lisp*. Each is a global special. These symbols, described below, are all in the compiler package.

peep-enable
: If this switch is non-Nil, the compiler runs the peephole optimizer. The optimizer makes the compiled code faster, but the compilation itself is slower. *Peep-enable* defaults to T.

peep-statistics
: If this switch is non-Nil, the effectiveness of the peephole optimizer (number of bytes before and after optimization) will be reported as each function is compiled. *Peep-statistics* defaults to Nil.

inline-enable

If this switch is non-Nil, then functions which are declared to be inline are expanded inline. It is sometimes useful to turn this switch off when debugging. *Inline-enable* defaults to T.

open-code-sequence-functions

If this switch is non-Nil, the compiler tries to translate calls to sequence functions into do loops, which are more efficient. It defaults to T.

optimize-let-bindings

If this is T, the compiler optimizes some let bindings, such as those generated by lambda expansions and setf based operations. If it is :All, the compiler optimizes all lets. If it is Nil, it does not optimize any. The optimization involves replacing instances of variables that are bound to other variables with the other variables. It defaults to T.

examine-environment-function-information

If this is non-Nil, look in the compiler environment for function argument counts and types (macro, function, or special form) if you don't get the information from declarations. It defaults to T.

nthcdr-open-code-limit

This is the maximum size an Nthcdr can be to be open coded. In other words, if Nthcdr is called with N equal to some constant less than or equal to the *Nthcdr-open-code-limit*, it will be open coded as a series of nested Cdr's. *Nthcdr-open-code-limit* defaults to 10.

complain-about-inefficiency

If this switch is non-Nil, the compiler will print a message when certain things must be done in an

inefficient manner because of lack of declarations or other problems that the user may not be aware of. It defaults to Nil.

eliminate-tail-recursion
If this switch is non-Nil, the compiler attempts to turn tail recursive calls (from a function to itself) into iteration. It defaults to T.

all-rest-args-are-lists
If non-Nil, this has the effect of declaring every &rest arg to be of type list. It defaults to Nil.

verbose If this switch is Nil, only true error messages and warnings go to the error stream. If non-Nil, the compiler prints a message as each function is compiled. It defaults to T.

check-keywords-at-runtime
If non-Nil, compiled code with &key arguments will check at runtime for unknown keywords. It defaults to T.

4.3. Declare Switches

The Optimize declaration controls some of the above switches:

- *Peep-enable* is on unless Cspeed is greater than Speed and Space.
- *Inline-enable* is on unless Space is greater than Speed.
- *Open-code-sequence-functions* is on unless Space is greater than Speed.

- *Eliminate-tail-recursion* is on if Speed is greater than Space.

5. Efficiency

This chapter summarizes ways to improve efficiency of Accent Lisp code.

5.1. Compile the Code

In Accent Lisp, compiled code typically runs at least one hundred times faster than interpreted code. Another benefit of compiling is that it catches many typos and other minor programming errors. Many Lisp programmers find that the best way to debug a program is to compile the program to catch simple errors and then debug the interpreted code. Many programmers use the compiled code only after the program is debugged.

Another benefit of compilation is that compiled (.Sfasl) files load significantly faster, so it is worthwhile to compile files that are loaded many times even if the speed of the functions in the file is unimportant.

Do not be concerned about the performance of a program until it has been compiled -- some techniques that make compiled code run faster make interpreted code run slower.

5.2. Avoid Unnecessary Storage Allocation

The Cons function allocates storage. However, Cons is not the only function that allocates storage -- Make-array and many other functions also do this.

Storage allocation can hinder performance because it reduces program memory access locality, which increases paging activity, and because it takes time.

Also, any space allocated eventually needs to be reclaimed, either by garbage collection or by killing Lisp.

It is necessary to allocate storage sometimes, and the Lisp implementors have tried to make storage allocation and the subsequent garbage collection as efficient as possible. In some cases strategic allocation can improve speed. It would certainly save time to allocate a vector to store intermediate results that are used numerous times.

5.3. Mapping

One of the programming styles encouraged by Lisp is a highly applicative one, involving the use of mapping functions and many lists to store intermediate results. To compute the sum of the square-roots of a list of numbers, one might say:

```
(apply #'+ (mapcar #'sqrt list-of-numbers))
```

This programming style is clear and elegant, but unfortunately results in slow code. There are two reasons why:

1. The creation of lists of intermediate results causes much storage allocation (see 5.2).

2. Each level of application requires another scan down the list. Thus, disregarding other effects, the above code would probably take twice as long as a straightforward iterative version.

An example of an iterative version of the same code:

```
(do ((num list-of-numbers (cdr num))
     (sum 0 (+ (sqrt (car num)) sum)))
    ((null num) sum))
```

5.4. Using Lists

Whereas lists were used extensively in early versions of Lisp, there are now other data structures that may be better suited to tasks where lists might have been used before. Think before using a list.

5.4.1. Vectors

Use vectors and use them often. Lists are often used to represent sequences, but for this purpose vectors have the following advantages:

- A vector takes up less space than a list holding the same number of elements. The advantage may vary from a factor of two for a general vector to a factor of sixty-four for a bit-vector. Less space means less storage allocation (see 5.2).

- Vectors allow constant time random-access. It is possible to get any element out of a vector as fast as getting the first out of a list if the right declarations are made.

The only advantage that lists have over vectors for representing sequences is that it is easy to change the length of a list, add to it and remove items from it. Signs of slow Lisp code are Nth and Nthcdr -- if you are using these function you should probably be using a vector.

5.4.2. Structures

Lists have also been used for the representation of record structures. Often the structure of the list is never explicitly stated and accessing macros are not used, resulting in impenetrable code such as:

```
(rplaca (caddr (cadddr x)) (caddr y))
```

The use of defstruct structures can result in much clearer code. One might write instead:

```
(setf (beverage-flavor (astronaut-beverage x)) (beverage-flavor y))
```

Since structures are based on vectors, the Defstruct version would likewise take up less space and be faster to access. Do not try to gain speed by using vectors directly, since the compiler knows how to compile faster accesses to structures. Note that the structure definition should be compiled before any uses of accessors.

5.4.3. Hashtables

In many applications where association lists (alists) have been used in the past, hashtables work much better. An alist may be preferable in cases where the user wishes to rebind the alist and add new values to the front, shadowing older associations. In most other cases, if an alist contains more than a few elements, a hashtable will probably work faster. If the keys in the hashtable are objects that can be compared with Eql or Eq, then hashtable access will be speeded up by specifying the correct function as the test argument to Make-hashtable.

5.4.4. Bit-vectors

Lists are also used for set manipulation. In some applications where there is a known, reasonably small universe of items bit-vectors should be used instead. This is much less convenient than using lists, because instead of symbols, each element in the universe must be assigned a numeric index into the bit vector. If the universe is very small -- twenty-eight items or less -- represent the set as bits in a fixnum and use Logior and so on to get large speed improvements.

5.5. Simple Vs Complex Arrays

Accent Lisp has two different representations for arrays, one that is accessed rapidly in microcode and one which is accessed much more slowly in Lisp code. The class of arrays that can be represented in the fast form corresponds exactly to the one-dimensional simple-arrays, as defined in the Common Lisp manual. Included in this group are the types simple-string, simple-vector and simple-bit-vector.

Declare vector variables -- otherwise the compiler will assume you are using the inefficient form of vector.

Example:

```
(defun iota (n)
  (let ((res (make-vector n)))
    (declare (simple-vector res))
    (dotimes (i n)
      (setf (aref res i) i))
    res))
```

Warning: if you declare things to be simple when they are not, incorrect code will be generated and hard-to-find bugs will result. It is worthwhile to note, however, that system functions that create vectors will always create simple-arrays unless forced to do otherwise.

5.6. Function Calls

The usual Lisp style involves small functions and many function calls; for this reason Lisp implementations strive to make function calling as inexpensive as possible. Accent Lisp is successful in this respect. Function calling is not vastly more expensive than other instructions.

For this reason do not be overly concerned about function-call overhead in programs. However, function calling does take time, so remove it whenever possible. Some techniques that can be used are:

- *Use inline functions.* This often is the best way to remove function call overhead in Common Lisp. A function may be written, and then declared inline if it is found that function call overhead is excessive. Writing functions is easier that writing macros, and it is easier to declare a function inline than to convert it to a macro. Note that the compiler must process first the inline declaration, then the definition, and finally any calls that are to be open coded for the inline expansion to take place.

- *Use macros.* A macro can achieve the effect of a function call without the function-call overhead, but the extreme generality of the macro mechanism makes them tricky to use. If macros are used in this fashion without some care, obscure bugs can result.

- *Write the code in-line.* This is not a very good idea, since it results in obscure code, and spreads the code for a single logical function out everywhere, making changes difficult.

Note that any of the above techniques can result in "bloated" code, since they duplicate the same instructions many places. If code becomes very large, paging may increase, resulting in a significant slowdown. Use inline expansion sparingly. Note that the same function may be called normally in some

places and expanded inline in others.

5.7. Keyword and Rest Arguments

Two Common Lisp argument passing mechanisms, keyword and rest arguments, have a potentially serious performance penalty in Accent Lisp. In Accent Lisp, rest arguments require consing a list to hold the arguments. If a function is called many times or with many arguments, large amounts of data allocation can occur. Keyword arguments are built on top of the rest argument mechanism, causing the same data allocation problem, in addition to requiring a significant amount of time for parsing the list of keywords and values on each function call.

Neither of these problems is significant unless a large number of calls are made to the same function, so the use of keyword and rest arguments in user interface functions is not necessarily discouraged. Use of macros instead of functions can also avoid this situation, because rest and keyword argument overhead occurs at compile time and not necessarily at runtime. If the macro-expanded form contains no keyword or rest arguments, then keywords and rest arguments can be used in macros that are in inner loops.

Note: the compiler open-codes most heavily-used system functions that have keyword or rest arguments, so no runtime overhead is involved in their use.

Optional arguments have no significant overhead.

5.8. Numbers

Accent Lisp provides five types of numbers:

- fixnums bignums ratios short-floats long-floats

Only short-floats and fixnums have an immediate representation; the rest must be allocated and garbage-collected later. In code where speed is important, try to use only fixnums and short-floats. Since most-positive-fixnum is more than one hundred million, bignums will need to be used rarely. The floating point precision that can be obtained with twenty-eight bits is limited, so there will be applications that will require the use of long-floats.

5.9. Timing

The Time function measures total elapsed time (see 2.7). For fairly short things it is often wise to write a compiled driver function which calls the function to be tested several times, and average the times. This helps reduce inaccuracy inherent in the time function due to the size of its timing units and due to paging.

6. Creating and Using Menus

This chapter describes the routines that Accent Lisp provides for creating and using menus.

6.1. The Menu Choose Functions

The basic menu choice macros and functions are:

Menu-prepare &key items position-x
position-y in-window stay-in-place
title title-font default-item
ncolumns items-justified
side-margin top&bottom-margin
spacing font label-font
abort-value [Macro]

menu-choose-from-structure menu
&optional pasteup [Function]

menu-choose &key items position-x
position-y in-window stay-in-place
title title-font default-item
ncolumns items-justified
side-margin top&bottom-margin
spacing font label-font
abort-value paste-up [Macro]

Menu-prepare takes a list of items and other arguments and returns a structure containing the given information in the format of a pop-up menu. This structure is intended to be passed to Menu-choose-from-

structure, which puts the menu onto the screen. Menu-choose combines Menu-prepare and Menu-choose-from-structure into one function call. Arguments for these functions are:

Items: the list of items to go into the menu. The items list is not evaluated, but font information, item names, and return values are evaluated at the time of menu selection. A help string may be provided with each item; this is not evaluated.

Position-x, Position-y, In-window: these specify the location of the pop-up menu. In-window specifies the window the menu is displayed in; it defaults to the window encompassing the whole screen. Position-x and Position-y give the x and y coordinates of the upper left-hand corner of the menu within that window; these default to the pixel x and y coordinates of the mouse pointer.

Stay-in-place: if non-Nil, specifies that the menu must appear exactly at the coordinates specified. If this is not possible, an error is signaled. If Stay-in-place is Nil, the menu position can be moved to fit onto the screen.

Title, Title-font: Title is a string to be displayed at the top of the menu in inverse video in font Title-font. If no Title is given, the inverse video title bar will not appear.

Default-item: the name of the item on which the mouse initially appears. The default Default-item is the item in the middle of the middle column. If Default-item is a non-existent item, it is ignored. If a menu structure has been chosen from before, the default item becomes the last item chosen from it.

Ncolumns: specifies the number of columns in the menu. This defaults to one (see 6.4).

Items-justified, Spacing: Items-justified tells whether to center items in the column or justify them to the left or right (:Center, :Left, :Right); it defaults to :Center. Spacing specifies the amount of white space inserted after each column (except the last one) in pixels; this defaults to twice the width of the letter x in Font.

Side-margin, Top&bottom-margin: the number of pixels of margin to be left at the sides and top and bottom of the menu, respectively; both default to ten.

Font, Label-font: Font is the default font for items that are not labels; it defaults to the standard default font. Label-font is the font used for labels, and defaults to the italic font corresponding to the default font.

Abort-value: the value that is returned if Ctrl-g is typed while choosing from the

menu; it defaults to Nil.

Paste-up: possible values are T, Nil, and :And-choose (anything else is considered T; if not specified, defaults to Nil). T causes the menu to be permanently displayed in the window, but does not allow selections to be made from it. :And-choose allows selections to be made, but the menu is permanently displayed. Nil displays the menu and allows a choice to be made, then deletes the menu. Menus pasted on the screen (by T or :And-choose) are stored in *Pasted-menus* and may be accessed by Pasted-menu-choose (below).

pasted-menu-choose menu [Function]

Takes either a menu structure already pasted on the screen or the title of a menu already on the screen, and allows a selection to be made from it. It returns Nil if the menu or title specified is not displayed or does not exist, and signals an error for any other argument.

unpaste-menu menu [Function]

Takes a menu structure already on the screen or the title of a menu already on the screen, removes it, and returns Nil.

pasted-menu-p menu [Function]

Returns T if its argument is a menu that is currently pasted on the screen, or the title of one; otherwise it returns Nil.

6.2. The Item

Each item that the user may choose consists of a string (or character or symbol) and some other information. The additional information may tell how the string should be printed, what to return if the user selects the item, or possibly a note saying that the item is just for show and may not be chosen.

The item may be just a name X, in which case X is displayed and (string X) is returned if the item is chosen. An item may be the keyword :New-column, which is not really displayed at all but signals that a column break is desired, for easy use of multiple columns. Finally, an item may be a list, whose Car is a name, and whose Cdr is a bunch of keyword arguments telling about the item. In fact each of these keyword arguments is just zero or more forms, giving the item the following format:

```
item ::= name | :new-column | (name {keyword arg* }*)

name ::= symbol | string | character.
```

Each keyword knows how many arguments it wants. If it wants a particular number it simply takes that many; if it wants a variable number then it may scan for the next keyword as a cue that it has seen all of its arguments. If at any time not enough arguments are

left for a keyword, or it is time to start parsing a new keyword argument and a keyword does not follow, an error is signalled.

The following sorts of keyword arguments may be provided:

:Help string The string name :Help is given as help in the Lisp window title line as long as the entry name with help Help is being selected with the mouse.

:Value value
The value is returned if the user selects this item with the mouse.

:Values ({value}*)
The values are returned as multiple values if the user selects this item with the mouse.

:Eval form The form is evaluated and returned when the user selects this item with the mouse.

:Funcall fn ({arg}*)
The result of the fn funcall is returned when the user selects this item with the mouse.

:No-select The item may not be selected. Trying to select this item with the mouse causes the Lisp window to flash.

:Buttons {button-specifier}*
This takes some button specifiers, each of which is a list. The specifier's Car indicates a particular button, :Left, :Middle, or :Right. The rest of the specifier list is a keyword argument of the :Values, :Eval, :Funcall, or :No-select form; if there is anything after the first keyword argument it is ignored. Thus each specifier indicates what to return if that button is pressed, or

that the button is inactive while selecting this item with the mouse. If a button is left unspecified it returns Nil. A button may not be specified more than once.

:Font font This specifies the font in which the item name is to be printed.

:Label This is an abbreviation for :Font label-font :No-select.

Help, font, and inverse are the only options which may be meaningfully combined with other options. If conflicting things are specified, the most recent specification takes precedence, so that ("Hosts" :label :font bold) causes "Hosts" to be displayed as a label, boldface and not selectable, and ("X" :eval (if t 3) :values 'Accent) returns Accent if selected.

6.3. User Selectable Attributes of the Choice Window

Some of the parameters to Menu-choose and Menu-prepare default to global variables whose values are initialized to the values stated in the first section. These parameters are controlled this way because they are relatively independent of the other parameters. If the function caller specifies a value for the parameter, that value is used.

Position-x, Position-y, and In-window are controlled by *Default-menu-choose-position-x*, *Default-menu-choose-position-y*, and *Default-menu-choose-in-window*. Ncolumns is controlled by *Default-menu-choose-ncolumns*. Items-justified is controlled by *Default-menu-choose-items-justified*. Side-margin and Top&bottom-margin are controlled by *Default-

menu-choose-side-margin* and *Default-menu-choose-top&bottom-margin*.

6.4. The Arrangement of Items in the Window

The shape of the window may be specified by putting :New-column markers in the items list. In this case the function simply does as told and its only intervention is when there are too many columns to fit in the allowed width, in which case it signals an error.

If no :New-column markers are present the window shape is controlled by specifying the parameter ncolumns. This parameter must be a positive integer number of columns (less than the number of items) to use, and it defaults to one.

If the menu is too wide to fit on the screen, or some columns are too tall, an error is signalled.

7. Using Matchmaker-Generated Lisp Interfaces

Matchmaker generates Pascal, C, and Lisp source code that implements procedural interfaces to message-based communications facilities in a language-independent manner. This facility is useful for implementing servers and message-based client interfaces. Matchmaker specifications are oriented toward algorithmic languages; IPC messages sent by remote procedure calls contain data structured in a Pascal-like manner. Because of this, Lisp must deal with data passing through messages as "alien" data rather than as Lisp tagged data. Alien data is defined by Pascal-like type definitions that appear in Matchmaker specifications. Lisp aliens provide a mechanism that allows Lisp programs to manipulate this type of data.

To produce a procedural interface to message communication for Lisp, Matchmaker analyzes the language-independent specification and produces Lisp code made up of functions, macros and special forms provided gy the Lisp "alien" facility. Client programs usually require only Matchmaker-generated Lisp forms and do not need to use alien primitives heavily.

Some data structures passed in messages are too complex for Matchmaker specifications (i.e., Matchmaker cannot generate forms for manipulating it). Examples of this are the variant record definitions of command blocks and data used in the IO interfaces

for RS232, Floppy, GBIP, and Speech (see IO.MM and IOAuxDefs.Pas). To write Lisp interfaces to these devices, Lisp forms must be written that work with the alien data and simply put the correct data bits in the correct places.

Detailed information on Matchmaker is in the *Matchmaker: The Accent Remote Call Procedure Language* document in the Accent Languages Manual.

The use of Matchmaker-generated code for Lisp is described in the following sections.

7.1. Interface Parameters and Return Values

Matchmaker generates Lisp remote procedure calls from language-independent specifications. Given a Matchmaker remote procedure specification such as:

```
Remote_Procedure AllocatePort(
                          : Port;
      out NewPort         : Port_All;
          BackLog         : Integer)
   : GR_Value;
```

Matchmaker will generate a Lisp function:

```
(defun AllocatePort (Remote_Port BackLog)
        .
        .
        .)
```

that returns as multiple values the GeneralReturn code of the call (e.g., the Success value) and the new port that was allocated, in that order. In general, Matchmaker generates a function that takes all In and Inout parameters in the order specified and produces multiple values. The multiple values returned are defined to be the returned value of the call, if any, followed by all the Inout and Out parameters in the order specified. For example, given this Matchmaker specification:

```
Remote_procedure CreateWindow(:  Window;
                          fixedPosition: Boolean;
                    inOut leftx: Integer;
                    inOut topy: Integer;
                          fixedSize: boolean;
                    inOut width: Integer;
                    inOut height: Integer;
                          hasTitle: boolean;
                          hasborder: boolean;
                          title: TitStr;
                    inOut progName: progStr;
                          hasIcon: boolean;
                    out   vp: Viewport
        ): Window;
```

a Lisp function of this form will be produced:

```
(defun CreateWindow (Remote_Port
                     fixedPosition
                     leftx
                     topy
```

```
                fixedSize
                width
                height
                hasTitle
                hasborder
                title
                progName
                hasIcon)
        .
        .
        .)
```

that returns as multiple values Window, Leftx, Topy, Width, Height, ProgName, and Vp in that order.

7.2. Matchmaker-Generated Interface Functions

7.2.1. Lisp remote procedures

When a procedural interface is specified by Matchmaker to be of type Remote_Procedure, a Lisp client interface function (with the specified remote procedure name) is generated that takes In parameters, packs them into a message, and sends the message to the specified server. The server will (if the message succeeds) receive the message, take some action, and return a reply message with Out parameters packed into it. The client interface function will receive the reply message, unpack the Out parameters, and return them as multiple values. A client process only needs to obtain an appropriate port to use a remote procedure interface.

7.2.2. Lisp messages

When a procedural interface is specified to be of type Message, a Lisp client interface function (with the specified message name) is generated that takes In parameters, packs them into a message, sends the message to the specified server, and then immediately returns either an Accent general return value or no useful value. The client interface function does not wait for a reply from the server and returns no Out parameters. The server will receive the message and take some action, but will not return a reply message with Out parameters packed in it. A client process only needs to obtain an appropriate port to use a message interface.

7.2.3. Lisp server messages

When a procedural interface is specified to be of type Server Message, a Lisp client function (with the specified server message name) is assumed to already exist that will take as arguments the In parameters and return no particular value. This type of interface is used to allow a server process to make an asynchronous request of a client process. To allow this to happen, the client process must manually call the Lisp Receive primitive and receive a message from the server. The client process then calls a Matchmaker-generated dispatching function that decodes the message and calls the Lisp client function that is assumed to exist as described above. The server process, meanwhile, after sending its message, continues to execute without waiting for any action from the client process. The assumed client function receives the message, takes some action, and returns a

reply message with Out parameters packed in it. The originally-called client Lisp function receives the reply message, unpacks the Out parameters, and returns them as multiple values. To use this type of interface, a client process must obtain appropriate ports and provide a function to be called by the server message. The client process is also responsible for manually receiving the server message and calling the dispatch function provided by Matchmaker. This dispatching function, by convention, is named *InterfaceName*Asynch, where *InterfaceName* is the name of the interface for which this server message is specified.

7.2.4. Lisp alternate replies

When a procedural interface is specified to be of type Alternate Reply, a Lisp client function (with the specified alternate reply name) is assumed to already exist that takes as arguments the In parameters as specified and returns whatever value the client process wants. This type of interface allows a server process to report exceptional conditions to a client process by returning as the reply to a remote procedure call a different sort of message than was expected. To allow this to happen, the client process must provide a function with the specified name that will take the specified arguments. The client process then makes remote procedure calls as usual. When the Matchmaker-provided client interface function for any remote procedure interface receives an unexpected reply message (that is, one with a message ID other than that it was expecting), an alternate reply dispatching function is called that calls the user-

supplied alternate reply function appropriate to the alternate reply message received. The message is unpacked and the In parameters supplied to the alternate reply function. Whatever the alternate reply function returns is returned as the value of the original remote procedure call.

While alternate replies can be thought of as being a sort of alternative "second half" of a remote procedure call, it is not the case that an alternate reply is necessarily associated with a particular remote procedure. Instead, it is associated with a particular *interface*, and any remote procedure call on that interface may give rise to any of the specified alternate replies for that interface.

7.3. Accessing Lisp Alien Data Structures

Matchmaker can generate translations of data going into and out of messages when the data is fairly simple. More complex, "structured" data must be manipulated in such a way that the underlying alien mechanisms become more apparent to the Lisp user. Matchmaker attempts to hide Lisp alien mechanisms as much as possible, with varying degrees of success depending upon the complexity of the data involved in the interface.

When passing simple parameters into a Matchmaker-generated remote procedure call, one can simply use the obvious Lisp counterparts to the specified types. These are:

signed, unsigned integers : fixnums

ports : fixnums

enumerations : fixnums, usually given keyword names

booleans : T or NIL

strings : Lisp strings

pointer : system-area-pointer

subranges : fixnums

floating point : either short or long floats

Similarly, when receiving values out of a remote procedure call, one can expect the above specified type of Lisp object.

7.3.1. Alien enumerations

An alien enumeration type causes Matchmaker to generate Lisp keywords corresponding to the enumeration values. These enumeration values are then passed into and out of Matchmaker-generated interfaces using the defined keywords. For example, given an enumeration type like:

```
type era = (stone-age medieval now space-age)
```

Lisp alien data of this type would be referred to using the keywords :Stone-age, :Medieval, :Now, and :Space-age.

7.3.2. Alien records

When remote procedure data is an array or record type, Matchmaker generates accessing macros for obtaining subparts of the data object. Given data and a remote procedure call specified as:

```
type

  ProcState  = (Supervisor,     ! 00 - supervisor with privileges
                Privileged,     ! 01 - user with privileges
                BadSupervisor,  ! 10 - supervisor without privileges
                User);          ! 11 - user without privileges

  ProcID     = integer;

  PriorID    = 0..NUMPRIORITIES-1;

  QID        = 0..NUMQUEUES;

  PStatus    = record
                   State        : ProcState;
                   Priority     : PriorID;
                   MsgPending   : boolean;
                   EMsgPending  : boolean;
                   MsgEnable    : boolean;
                   EMsgEnable   : boolean;
                   LimitSet     : boolean;
                   SVStkInCore  : boolean;
                   QueueID      : QID;
                   SleepID      : long;
                   RunTime      : long;
                   LimitTime    : long
               end record;
```

```
Remote Procedure Status(
                        : Port;
    out NStats          : PStatus)
  : GR_Value;
```

the Lisp function Status generated by Matchmaker would take one parameter, the kernel port of the process for which status is being requested. It would return two values: the GeneralReturn code of the call followed by the PStatus record NStats. The PStatus record would be returned as an alien value; that is, it would be in such a form that Matchmaker-generated accessing macros would need to be used to access it. The alien value would be of the form:

```
#<Alien value, Address = #x800184C, Size = 240, Type = PStatus>
```

Macros to access such an alien value would be generated with names:

Access-PStatus-State

Access-PStatus-Priority

Access-PStatus-MsgPending

Access-PStatus-EMsgPending

Access-PStatus-MsgEnable

Access-PStatus-LimitSet

Access-PStatus-SVStkInCore

Access-PStatus-QueueID

Access-PStatus-SleepID

Access-PStatus-RunTime

Access-PStatus-LimitTime

Thus, if variable PS was given the alien value returned by Status, the Priority record field could be accessed using the form (Access-PStatus-Priority PS), which would return a fixnum, since Matchmaker subranges are translated into Lisp fixnums.

Due to the way alien mechanisms are implemented in Lisp, only accessing macros that return Lisp data (as opposed to other alien values) are named in the form "Access-X" as in the above example. Macros that access alien values and return other alien values are named in the form "X-Op". For example, given these record definitions:

```
type
  InnerRec  = record
                field1 : integer;
                field2 : long
              end record;

  OuterRec  = record
                fieldA : InnerRec;
                fieldB : long;
                fieldC : short
              end record;
```

the macro for accessing FieldB of OuterRec would be named "Access-OuterRec-FieldB", while the macro for accessing FieldA would be named "OuterRec-FieldA-

Op". To access the Lisp data in an OuterRec alien value bound to some variable Foo, the following forms would be used:

```
(Access-InnerRec-Field1 (OuterRec-FieldA-Op Foo))

(Access-InnerRec-Field2 (OuterRec-FieldA-Op Foo))

(Access-OuterRec-FieldB Foo)

(Access-OuterRec-FieldC Foo)
```

7.3.3. Alien arrays

To access alien array values, macros named exactly as above are used, but an additional argument is given them along with the form that evaluates to the alien value. This argument is the array index. Arrays that have alien-valued elements are accessed using macros named as "X-Op" while alien arrays that have elements automatically translated into Lisp data are accessed using macros named as "Access-X" Given type specifications of the form:

```
type
  AlienValuedArray = array [10] of PStatus;

  LispDataArray = array [37] of integer;
```

Array X of type AlienValuedArray would be accessed using a form like (AlienValuedArray-Op X n) and Array Y of type LispDataArray would be accessed using a form like (Access-LispDataArray Y n).

Alien accessing macros may be composed as needed; but only the outermost macro (at most) should have

names of the form "Access-X" while all inner macros of the composition should have names of the form "X-Op."

7.3.4. Alien pointers

Data passed as pointers in a Matchmaker-generated interface is returned from and passed into these interfaces as Lisp system-area-pointers. When a Matchmaker interface specification uses a pointer type, Lisp currently must use system-area-pointers that give the internal address of the data returned or passed. Alien data structures may be created from these pointers using the primitives defined later in this document (see section 7.6).

Pointers to pre-existing alien data structures may be passed into an interface by using the Alien-Address primitive to obtain a system-area-pointer. Matchmaker generates the same sort of accessing macros as those described above for records and arrays for data structures specified in an interface specification even if those records and arrays are passed only through pointers (or not used as interface parameters at all). Thus one may take system-area-pointers, create an alien, and access it using the Matchmaker-generated macros for such data.

7.4. Alien Values

Objects in messages are manipulated via typed pointers to the data involved. These typed pointers are called Alien values. An alien value is a Lisp object consisting of three components:

Address — The address of the object pointed to. This is a word address, which may be a ratio, since objects need not be word aligned.

Size — The size in bits of the object pointed to. This information is used to make sure that accesses to the object fall within it.

Type — The alien type of the object pointed to. Since alien values have a type, functions that use them can check that their arguments are of the correct type.

7.5. Alien Types

Alien types are tags attached to alien values that may be checked to assure that they are not used inappropriately. When types are compared the comparison is done with the Lisp Equal function. Types are typically represented by symbols or lists of symbols such as the following:

```
string

(directory-entry type-file)

(signed-byte 7)

string-char
```

A convention that is encouraged, but not enforced, is

that an ordinary type is represented by a symbol, and a type with some subtype information, such as a discriminated union, is represented as a list of the main type and the subtype information.

7.6. Alien Primitives

This section describes the primitives defined by alien. Some of these primitives are intended to be used only in code generated by Matchmaker, while others are available to users.

make-alien type size
&optional address [Function]

Make an alien object of type Type that is Size bits long. Address may be either a number, :Static or :Dynamic. If Address is a number, then that becomes the returned alien's address. If Address is :Static or :Dynamic then storage is allocated to hold the data. Aliens that are allocated statically are packed as many as will fit on a page, resulting in increased storage efficiency, but disallowing the deallocation of the storage. Since static aliens are allocated contiguously, the Save function can arrange to save their contents. Dynamic Aliens are allocated on page boundaries, and may be deallocated using Dispose-Alien.

alien-type alien [Function]

alien-size alien [Function]

alien-address alien [Function]

These functions return the type, size and address of Alien, respectively.

copy-alien alien [Function]

Copies the storage pointed to by Alien and returns a new alien value that describes it.

dispose-alien alien [Function]

Releases any storage associated with Alien. Any reference to Alien afterward may fail.

alien-access alien lisp-type [Function]

Returns the object described by Alien as a Lisp object of type Lisp-type. An error is signaled if the type of Alien cannot be converted to the given Lisp-type. For most Lisp-types the corresponding alien type is identical.

Lisp-type, which is not evaluated, must be one of the following:

(Unsigned-byte n)
: An unsigned integer N bits wide, as in Common Lisp.

(Signed-byte n)
: A signed integer N bits wide.

Boolean
: A one-bit value, represented in Lisp as T

or Nil.

(Enumeration name)
Accesses a value of the enumeration Name. Enumerations are defined by the macro Defenumeration.

String-char An eight-bit ASCII character.

Simple-string
A perq-string, which is a PERQ Pascal string.

Sort An Accent IPC port.

Short-float, Long-float
There is only one alien type accepted by these, ieee-single, which is a floating point number in pseudo-IEEE single format, as used by PERQ Pascal. Both Lisp-types are allowed so that one may choose whether to Cons long-floats or lose precision.

System-area-pointer
Returns as a system-area-pointer the long-word described by Alien. It is an error for the address not to be in the system area.

alien-store alien lisp-type
new-value [Function]

Alien-Store is the inverse of Alien-Access. It stores the Lisp object New-value in the place specified by Alien. Lisp-type may have the

same values as for Alien-Access, with one additional value:

(Pointer type)
: Type may be any unboxed Lisp type such as Simple-string, Simple-bit-vector and (Simple-array (Unsigned-byte 8)). When an object of such a type is stored the address of the first data word is stored in the corresponding location.

alien-assign dest dest-offset
source source-offset
size [Function]

Copies part or all of an existing alien into another alien. Alien-Assign takes five arguments: the destination alien (Dest), a bit offset into the destination alien (Dest-offset), the source alien (Source), a bit offset into the source alien (Source-offset), and a bit size to be copied into the destination alien (Size).

7.7. Alien Variable Primitives

An alien variable is a symbol that has had an alien value associated with it. An alien variable is not a Lisp variable -- in order to obtain the value of an alien variable, the special form Alien-Value must be used. The reason for using alien variables as opposed to Lisp variables is that various additional information can be associated with the alien variable that may permit code referring to it to be compiled more efficiently.

alien-value name [Function]

Returns the value of the alien variable Name.

alien-bind
((name value type)*)
{form}*[Function]

Defines a local alien variable Name having the specified alien Value. Bindings are done serially, as by Let*.

defalien name type
size [address] [Macro]

Defines Name as an alien variable, creating a value from Type, Size and Address as for Make-Alien. Name and Type are not evaluated.

Appendix A. Example of Matchmaker-generated Alien Data

An example of the use of Matchmaker-generated alien data is provided by the piece of Matchmaker-generated code for the Message/NameServer client interface below. The alien variables denoted To-X and From-X are the actual message records used by interface routine X for message communication. Also shown is the use of Send and Receive using alien data structures.

```
;;;;   -*-Lisp-*-
;;;;
;;;;  Matchmaker generated user interface file for MsgM
;;;;
;;;;
;;;;            THIS FILE SHOULD NOT BE HAND EDITED
;;;;
;;;;  To change this interface, edit the interface Matchmaker
;;;;  file and run it through Matchmaker to generate this file.
;;;;


(in-package "MSGMUSER")
(use-package "MSGMDEFS")
(use-package "BUILTINDEFS")
(use-package "MMINTERNALDEFS")


(use-package "ACCINTUSER")
(defvar *receiveport* NIL)
(defconstant interface-backlog 0)
```

```
(use-package "ACCINTDEFS")


;;; MsgN-Send -- internal
;;;
;;; Send macro for MsgN
;;;
(defmacro MsgN-Send (msg argblock wait-time send-option)
  "User send macro for interface MsgN"
  `
  (cond ((fixnump *receiveport*)
         (alien-store (msg-localport-op , msg) port *receiveport*)
         (send , argblock , wait-time , send-option))
        (t
         notaport)))


;;;
;;; User side interface initialization function.
;;;
(defun MsgN-Init (user-port)
  "The user side intialization function"
  (cond ((eql user-port nullport)
         (multiple-value-bind (gr port)
           (allocateport kernelport interface-backlog)
           (if (eql gr success) (setq *receiveport* port)
               (setq *receiveport* dataport))
           gr))
        (t
         (setq *receiveport* user-port)
         success)))
```

```
;;; CheckIn -- public
;;;
;;; User-side remote procedure call interface.
;;;
(defun CheckIn (Remote_Port PortsName Signature PortsID)
  "The user side of remote procedure CheckIn"
  (prog ((send-waittime 0)
         (send-option 0)
         (receive-waittime 0)
         gr)
    (alien-bind ((to to-CheckIn to-CheckIn))
                (alien-store
                 (msg-remoteport-op (Msgize-to-CheckIn-op
                         (alien-value to))) port Remote_Port)
                (alien-store
                 (to-CheckIn-PortsName-op (alien-value to))
                             simple-string PortsName)
                (alien-store
                 (to-CheckIn-Signature-op (alien-value to)) port
                             Signature)
                (alien-store
                 (to-CheckIn-PortsID-op (alien-value to)) port
                             PortsID)
                (setq gr
                      (MsgN-send
                       (Msgize-to-CheckIn-op (alien-value to))
                       to-CheckIn send-waittime send-option)))
    (alien-bind ((from from-CheckIn from-CheckIn))
                (cond ((eql gr success)
                       (alien-store
                        (msg-msgsize-op
                         (Msgize-from-CheckIn-op
                                 (alien-value from)))
```

```
 (signed-byte 32) 28)
(alien-store
 (msg-localport-op
  (Msgsize-from-CheckIn-op
         (alien-value from)))
 port *ReceivePort*)
(setq gr
   (receive from-CheckIn receive-waittime
    localpt receivewit))
(cond ((not (eql gr success))
       NIL
       (return gr))
      ((=
        (access-msg-id
         (Msgsize-from-CheckIn-op
           (alien-value from)))
        1100)
       (cond ((=
               (access-typetypei-typename
                (msg-retcode-typetypei-op
                 (Msgsize-from-CheckIn-op
                  (alien-value from))))
               1)
              (setq gr
                    (access-msg-retcode
                     (Msgsize-from-CheckIn-op
                      (alien-value from)))))
             (t
              NIL
              (return badreply)))
       NIL
       (return (values gr)))
      (t
       NIL
```

```
                        (return
                         (user-alternate-return
                          (Msgize-from-CheckIn-op
                            (alien-value from)))))))
                    (t
                     NIL
                     (return gr))))))


;;; Lookup -- public
;;;
;;; User-side remote procedure call interface.
;;;
(defun Lookup (Remote_Port PortsName)
  "The user side of remote procedure Lookup"
  (prog (PortsID
         (send-waittime 0)
         (send-option 0)
         (receive-waittime 0)
         gr)
    (alien-bind ((to to-Lookup to-Lookup))
                (alien-store
                 (msg-remoteport-op (Msgize-to-Lookup-op
                                     (alien-value to)))
                            port Remote_Port)
                (alien-store (to-Lookup-PortsName-op
                              (alien-value to))
                             simple-string PortsName)
                (setq gr
                      (MsgN-send (Msgize-to-Lookup-op
                                  (alien-value to)) to-Lookup
                       send-waittime send-option)))
    (alien-bind ((from from-Lookup from-Lookup))
```

```
(cond ((eql gr success)
       (alien-store
        (msg-msgsize-op
         (Msgize-from-Lookup-op (alien-value from)))
        (signed-byte 32) 36)
       (alien-store
        (msg-localport-op
         (Msgize-from-Lookup-op (alien-value from)))
        port *ReceivePort*)
       (setq gr
             (receive from-Lookup receive-waittime
                      localpt receiveit))
       (cond ((not (eql gr success))
              NIL
              (return gr))
             ((=
               (access-msg-id
                (Msgize-from-Lookup-op
                 (alien-value from)))
               1101)
              (cond ((=
                      (access-typetype1-typename
                       (msg-retcode-typetype1-op
                        (Msgize-from-Lookup-op
                         (alien-value from))))
                      1)
                     (setq gr
                           (access-msg-retcode
                            (Msgize-from-Lookup-op
                             (alien-value from)))))
                    (t
                     NIL
                     (return badreply)))
              (cond ((eql
```

```
                                (access-typetypei-typename
                                 (from-Lookup-PortsID-
                                                typetypeI-op
                                  (alien-value from)))
                                6)
                               (setq PortsID
                                 (access-from-Lookup-PortsID
                                  (alien-value from))))
                              (t
                               NIL
                               (return BADREPLY)))
                        NIL
                        (return (values gr PortsID)))
                       (t
                        NIL
                        (return
                         (user-alternate-return
                          (Msgize-from-Lookup-op
                            (alien-value from)))))))
                (t
                 NIL
                 (return gr))))))


;;; CheckOut -- public
;;;
;;; User-side remote procedure call interface.
;;;
(defun CheckOut (Remote _Port PortsName Signature)
  "The user side of remote procedure CheckOut"
  (prog ((send-waittime 0)
         (send-option 0)
         (receive-waittime 0)
```

```
    gr)
(alien-bind ((to to-CheckOut to-CheckOut))
            (alien-store
             (msg-remoteport-op
                 (Msgize-to-CheckOut-op (alien-value to)))
             port Remote_Port)
            (alien-store (to-CheckOut-PortsName-op
                             (alien-value to))
                         simple-string PortsName)
            (alien-store (to-CheckOut-Signature-op
                             (alien-value to))
                         port
                         Signature)
            (setq gr
                  (MsgN-send (Msgize-to-CheckOut-op
                                 (alien-value to))
                   to-CheckOut send-waittime send-option)))
(alien-bind ((from from-CheckOut from-CheckOut))
            (cond ((eql gr success)
                   (alien-store
                    (msg-msgsize-op
                     (Msgize-from-CheckOut-op
                        (alien-value from)))
                    (signed-byte 32) 28)
                   (alien-store
                    (msg-localport-op
                     (Msgize-from-CheckOut-op
                         (alien-value from)))
                    port *ReceivePort*)
                   (setq gr
                         (receive from-CheckOut receive-waittime
                                  localpt receiveit))
                   (cond ((not (eql gr success))
                          NIL
```

```
                (return gr))
               ((=
                 (access-msg-id
                  (Msgize-from-CheckOut-op
                      (alien-value from)))
                 1102)
                (cond ((=
                        (access-typetypei-typename
                         (msg-retcode-typetypei-op
                          (Msgize-from-CheckOut-op
                           (alien-value from))))
                        1)
                       (setq gr
                             (access-msg-retcode
                              (Msgize-from-CheckOut-op
                               (alien-value from)))))
                      (t
                       NIL
                       (return badreply)))
                NIL
                (return (values gr)))
               (t
                NIL
                (return
                 (user-alternate-return
                  (Msgize-from-CheckOut-op
                      (alien-value from)))))))
          (t
           NIL
           (return gr))))))


;;; MsgPortStatus -- public
```

```
;;;
;;; User-side remote procedure call interface.
;;;
(defun MsgPortStatus (Remote_Port PortsID)
  "The user side of remote procedure MsgPortStatus"
  (prog (GlobalPort
         Owner
         Receiver
         SrcID
         SeqNum
         NetWaiting
         NumQueued
         Blocked
         Locked
         RecvQueue
         DataOffset
         InSrcID
         InSeqNum
         (send-waittime 0)
         (send-option 0)
         (receive-waittime 0)
         gr)
    (alien-bind ((to to-MsgPortStatus to-MsgPortStatus))
                (alien-store
                 (msg-remoteport-op
                   (Msgize-to-MsgPortStatus-op (alien-value to)))
                 port Remote_Port)
                (alien-store
                 (to-MsgPortStatus-PortsID-op
                   (alien-value to)) port
                            PortsID)
                (setq gr
                      (MsgN-send
                       (Msgize-to-MsgPortStatus-op (alien-value to))
```

```
                    to-MsgPortStatus send-waittime send-option)))
    (alien-bind ((from from-MsgPortStatus from-MsgPortStatus))
                (cond ((eql gr success)
                       (alien-store
                        (msg-msgsize-op
                         (Msgize-from-MsgPortStatus-op
                             (alien-value from)))
                        (signed-byte 32) 122)
                       (alien-store
                        (msg-localport-op
                         (Msgize-from-MsgPortStatus-op
                             (alien-value from)))
                        port *ReceivePort*)
                       (setq gr
                             (receive from-MsgPortStatus
                                 receive-waittime localpt
                                 receiveit))
                       (cond ((not (eql gr success))
                              NIL
                              (return gr))
                             ((=
                               (access-msg-id
                                (Msgize-from-MsgPortStatus-op
                                    (alien-value from)))
                               1102)
                              (cond ((=
                                      (access-typetypei-typename
                                       (msg-retcode-typetypei-op
                                        (Msgize-from-
                                                  MsgPortStatus-op
                                         (alien-value from))))
                                      1)
                                     (setq gr
                                       (access-msg-retcode
```

```
                (Msgsize-from-
                        MsgPortStatus-op
                 (alien-value from)))))
            (t
             NIL
             (return badreply)))
        (cond ((eql
                (access-typetypei-typename
                 (from-MsgPortStatus-
                       GlobalPort-typetypeI-op
                  (alien-value from)))
                2)
               (setq GlobalPort
                (access-from-
                   MsgPortStatus-GlobalPort
                 (alien-value from))))
            (t
             NIL
             (return BADREPLY)))
        (cond ((eql
                (access-typetypei-typename
                 (from-MsgPortStatus-
                    Owner-typetypeI-op
                  (alien-value from)))
                2)
               (setq Owner
                (access-from-
                    MsgPortStatus-Owner
                 (alien-value from))))
            (t
             NIL
             (return BADREPLY)))
        (cond ((eql
                (access-typetypei-typename
```

```
            (from-MsgPortStatus-
               Receiver-typetypeI-op
             (alien-value from)))
           2)
          (setq Receiver
           (access-from-
              MsgPortStatus-Receiver
            (alien-value from))))
         (t
          NIL
          (return BADREPLY)))
  (cond ((eql
          (access-typetypei-typename
           (from-MsgPortStatus-
                     SrcID-typetypeI-op
            (alien-value from)))
           2)
          (setq SrcID
           (access-from-
              MsgPortStatus-SrcID
            (alien-value from))))
         (t
          NIL
          (return BADREPLY)))
  (cond ((eql
          (access-typetypei-typename
           (from-MsgPortStatus-
                    SeqNum-typetypeI-op
            (alien-value from)))
           2)
          (setq SeqNum
           (access-from-
              MsgPortStatus-SeqNum
            (alien-value from))))
```

```
        (t
         NIL
         (return BADREPLY)))
    (cond ((eql
            (access-typetypei-typename
             (from-MsgPortStatus-
                NetWaiting-typetypeI-op
              (alien-value from)))
            0)
           (setq NetWaiting
            (access-from-
                MsgPortStatus-NetWaiting
             (alien-value from))))
          (t
           NIL
           (return BADREPLY)))
    (cond ((eql
            (access-typetypei-typename
             (from-MsgPortStatus-
                NumQueued-typetypeI-op
              (alien-value from)))
            1)
           (setq NumQueued
            (access-from-
                MsgPortStatus-NumQueued
             (alien-value from))))
          (t
           NIL
           (return BADREPLY)))
    (cond ((eql
            (access-typetypei-typename
             (from-MsgPortStatus-
                Blocked-typetypeI-op
              (alien-value from)))
```

```
        0)
       (setq Blocked
        (access-from-
            MsgPortStatus-Blocked
         (alien-value from))))
      (t
       NIL
       (return BADREPLY)))
(cond ((eql
        (access-typetypei-typename
         (from-MsgPortStatus-
                 Locked-typetypeI-op
          (alien-value from)))
        0)
       (setq Locked
        (access-from-
            MsgPortStatus-Locked
         (alien-value from))))
      (t
       NIL
       (return BADREPLY)))
(cond ((eql
        (access-typetypei-typename
         (from-MsgPortStatus-
             RecvQueue-typetypeI-op
          (alien-value from)))
        1)
       (setq RecvQueue
        (access-from-
            MsgPortStatus-RecvQueue
         (alien-value from))))
      (t
       NIL
       (return BADREPLY)))
```

```
(cond ((eql
        (access-typetypei-typename
         (from-MsgPortStatus-
             DataOffset-typetypeI-op
          (alien-value from)))
        2)
       (setq DataOffset
        (access-from-
            MsgPortStatus-DataOffset
         (alien-value from))))
      (t
       NIL
       (return BADREPLY)))
(cond ((eql
        (access-typetypei-typename
         (from-MsgPortStatus-
                 InSrcID-typetypeI-op
          (alien-value from)))
        2)
       (setq InSrcID
        (access-from-
            MsgPortStatus-InSrcID
         (alien-value from))))
      (t
       NIL
       (return BADREPLY)))
(cond ((eql
        (access-typetypei-typename
         (from-MsgPortStatus-
             InSeqNum-typetypeI-op
          (alien-value from)))
        2)
       (setq InSeqNum
        (access-from-
```

```
                                    MsgPortStatus-InSeqNum
                                  (alien-value from))))
                                (t
                                 NIL
                                 (return BADREPLY)))
                            NIL
                            (return
                             (values gr GlobalPort Owner Receiver
                                     SrcID SeqNum NetWaiting
                                     NumQueued Blocked Locked
                                     RecvQueue DataOffset InSrcID
                                     InSeqNum)))
                           (t
                            NIL
                            (return
                             (user-alternate-return
                              (Msgize-from-MsgPortStatus-op
                                      (alien-value from)))))))
                     (t
                      NIL
                      (return gr))))))

;;; MsgNAsynch -- internal
;;;
;;; User-side asynchronous message dispatching function.
;;;
(defun MsgNAsynch (msg)
  "User-side asynchronous message dispatching function."
  (alien-bind ((alien-msg msg msg))
              (case (access-msg-id (alien-value alien-msg))
                (t badmsgid))))
```

```
;;; User-Alternate-Return -- internal
;;;
;;; User-side alternate return dispatching function.
;;;
(defun user-alternate-return (msg)
  "User-side alternate return dispatching function."
  (alien-bind ((alien-msg msg msg))
              (case (access-msg-id (alien-value alien-msg))
                (t badreply))))


(export '(MsgN-Init MsgNAsynch CheckIn
          Lookup CheckOut MsgPortStatus))
```

INDEX