Mesa Debugger Documentation

Version 5.0
April 1979

The facilities documented here are the workings of an interactive Mesa debugger. It has been designed to support source level debugging; it provides facilities that allow users to set breakpoints, trace program execution, display the runtime state, and interpret Mesa statements.

XEROX
SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

Preface

April 1979

The facilities documented here are the workings of an interactive Mesa debugger. It has been designed to support source level debugging; it provides facilities that allow users to set breakpoints, trace program execution, display the runtime state, and interpret Mesa statements. Due to the space required to provide all of these capabilities, the Mesa debugger lives a core swap away from the program being debugged.

This documentation is divided into seven parts. **Section 1** is an overview, **Section 2** describes the user interface, **Section 3** explains the debugger's input conventions and contains a summary of the command tree structure, **Section 4** explains the semantics of each command, **Section 5** explains the debugger interpreter, **Section 6** explains the debugger's output conventions, and **Section 7** explains signal and error messages. The *Mesa User's Handbook* contains further details on how to obtain, install, and use the debugger.

The Mesa debugger is intended for use by experienced programmers already familiar with Mesa. All suggestions as to the form, correctness, and understandability of this document should be sent to your support group. All of us involved in the development of Mesa welcome feedback and suggestions on debugger development.

Section 1: Overview

The debugging and runtime facilities differ in their relationship to the user program. When you invoke Mesa, the *Mesa Executive* is the code necessary for your program to communicate with the debugger and resides with the user program. It also serves the function of an executive when the Mesa system is first started (see the *Mesa System Documentation* for further details). The *debugger* resides in a different core image which is loaded when called for (in very much the same way as Swat).

Installing the debugger

Before client programs can use the Mesa debugger, it must be installed (which saves the current core image as the *debugger*). Typing `XDebug` to the Alto Executive automatically installs the debugger. Other programs may be loaded into the debugger by including their name on the command line. See the *Mesa User's Handbook* for further details.

Invoking the debugger

At system start-up, the *Mesa Executive* is given control in a context from which all the various system utilities are visible. At this point there are several ways of invoking the debugger. The straightforward method is to issue the `Debug` command to the *Mesa Executive*; this brings you into the debugger, ready to execute a command. If you wish to enter the debugger at any time (i.e., while your program is running), ↑SWAT interrupts your program. (Note that if you really get in trouble, ↑SHIFT-SWAT brings you into Swat, at which point you may boot the machine and re-enter the debugger. **Section** 7 contains further details on bootloading the debugger.)

In the course of running your program, you may enter the debugger for several other reasons, including an uncaught signal generated by your program, execution of a breakpoint/tracepoint that has been placed in your program, or a fatal system error that forces your program to abort (**Section** 7 contains further details on the messages displayed when entering the debugger in these situations).

Talking to the debugger

The user interface to the debugger is controlled by a command processor which invokes a collection of procedures for managing breakpoints, examining user data symbolically, and setting the context in which user symbols are looked up. The command syntax is tree structured and each character is extended to the maximal unique string which it specifies.

Whenever an invalid character is typed, a ? is displayed and you are returned to command level. Typing a ? at any point during command selection prompts you with the collection of valid characters (in upper case) and their associated maximal strings (in lower case) and returns you to command level. Whenever a valid command is recognized, you are prompted for parameters (**Section** 3 contains further details on the input conventions). Typing DEL at any point during command selection or parameter collection returns you to the command processor; holding down ↑DEL at any point during command execution aborts the command.

When initialized, the debugger creates two windows: the DEBUG.LOG window (which becomes a record of the debugging session) and a source window (which is loaded with the source file when

breakpoints are set or the source location is requested). These windows may be manipulated by the window executive (**Wisk**) which comes with your debugger (see **Section 2** for further details).

Current context

The interpretation of symbols (including displaying variables, setting breakpoints, and calling procedures) is based on the notion of the current context; it consists of the current frame and its corresponding module, configuration, and process. The symbol lookup algorithm used by the debugger is as follows: it searches the runtime stack of procedure frames in LIFO order by examining first the local frame of each procedure (and then its associated global frame) following return links, until the root of the process is encountered.

When you first enter the debugger, the context is set to the frame of whatever process is currently running (i.e., to the *Mesa Executive*, if you enter via the `Debug` command; to your program, if it is interrupted or at a breakpoint). There are commands which make it simple to change between contexts (`SEt Context`), to display the current context (`CUrrent context`), and to examine the current dynamic state (`Display Stack`).

Leaving the debugger

Once you are in the debugger, you may execute any number of commands that allow you to examine (and change) the state of your program. When you are finished, you may decide either to continue execution of your program (`Proceed`), terminate execution of your program (`Quit`), or end the debugging session completely and return to the *Alto Executive* (`Kill session`). **Section 4** contains further details on these commands.

Section 2: User Interface

The Mesa debugger uses the Tools Environment window/menu/selection package (**Wisk**). For more complete documentation on the philosophy behind this interface, see the *Tools Environment: Guide for Tools Users* and the *Tools Environment: Functional Specification for Tools Writers.*

Standard window configuration

The debugger is created with two windows: a debug window (DEBUG.LOG) and an empty source file window. The same selection scheme, scrolling commands, and standard window commands apply to both windows. See below for details on functions specific to each window. Note: these functions are best understood by trying them as you read this document.

Current selection/ typein

There is only one selection visible at any time (*not one per window*); typein goes to the window containing the cursor, regardless of whether that window is on top. Type ahead of mouse clicks and keystrokes are buffered.

Text area

Selections are made by depressing either the RED or YELLOW mouse buttons while in the text area. RED selects between characters and extends selections by characters; YELLOW selects a word and extends selections by words. The current selection is outlined by a gray box.

Scroll bar

The scrolling commands are initiated by moving the cursor into the scroll bar (left margin of a window) and clicking a mouse button; scrolling is activated when the mouse button is released. Moving out of the scroll bar before releasing the button returns you to text selection mode without repositioning the file. The thermometer in the scroll bar shows the current position of the window in the file. The positioning commands are as follows:

scrolling up [RED button]
moves the line next to the cursor to the top of the window.

relative scrolling [YELLOW button]
moves to the position in the file corresponding to the relative position of the cursor in the scroll bar (also called "thumbing").

scrolling down [BLUE button]
causes the line at the top of the window to be moved next to the cursor.

Menu commands

When the BLUE mouse button is pressed in the text area of a window, an array of menus appears and the cursor changes to a left arrow. Select a menu by pointing at its header (causing it to video

reverse) and releasing the mouse button (or alternatively, you may click **RED** over the title of the desired menu while continuing to hold the **BLUE** button down). Similarly, select a menu command by pointing at it (causing it to video reverse) and releasing the mouse button. After seeing the menu, if you do not wish to execute a menu command, move the cursor away from the menu and release the **BLUE** mouse button. Except where otherwise noted below, clicking the **RED** mouse button causes the command to be executed.

When **Wisk** is working on a command, the cursor is changed to an hourglass. When it is done with the current task, the cursor returns to its normal shape. If for some reason it cannot complete the current task, the display is blinked.

The standard window commands are as follows:

`Move`

repositions a corner of the window in any direction. Clicking **RED** positions that corner of the window to the cursor location. Note that the window is clipped at the display boundaries; however, this does not change its actual size.

`Grow`

pulls a corner of the window in any direction, growing or shrinking the window along either dimension (width or height). Clicking **RED** fixes the size of the window (subject to a minimum size restriction).

`Size`

shrinks the window to a small box at the top of the display (or wherever you move it), showing just the window name. This is a toggle command; alternate invocations restore and shrink the window size. It is suggested you do this to windows not currently in use, since this frees up much of the space associated with the window.

`Top`

causes the window to be displayed on top of all other windows.

`Bottom`

causes the window to be displayed underneath all other windows.

`Zoom`

causes the window to grow to take up all of the available bitmap space. Alternate invocations of this command restore and `Zoom` the window.

These window operations may also be invoked by positioning the cursor in the left, middle, or right region of the window header and clicking one of the mouse buttons. The functions are as described above with the exception of Top/Bottom: if the window is not on top, move it to the top; if it is already on top, move it to the bottom. The header commands are as follows:

	Left Region	Middle Region	Right Region
RED	Top/Bottom	Zoom	Top/Bottom
YELLOW	Grow (corner)	Grow (edge)	Grow (corner)
BLUE	Move	Size	Move

Debug window

The debug window is used for user/debugger communication (i.e., invoking commands, reporting uncaught signals). There is a blinking vertical bar at the place that is currently expecting input.

Wisk window

The debug window has the following special property: if some typein or command output occurs when the blinking cursor is not visible, the debug window splits, creating a small subwindow (the wisk window) at the bottom of the debug window. This allows you to view the current user/debugger interaction at the same time as another part of the file. For example, suppose you wish to compare the value of a variable from one breakpoint to the next. When you reach the second breakpoint, simply scroll the top subwindow so that the first value of the variable is displayed, as the new value is being displayed in the wisk window.

The menu commands specific to the debug window are as follows:

`Alter Bitmap`

> takes the current selection as the (decimal) number of pages and expands/contracts the height of the bitmap. This is subject to a minimum of 10 pages and a maximum of 127 pages; the standard size is 46 pages.

`Move Boundary`

> allows you to reposition the upper boundary of the wisk window with the RED button.

`Stuff It`

> takes the current selection and stuffs it into the input stream of the debug window. The lower left function key on Alto II keyboards (FL4) and the blank key to the right of the return key on Alto I keyboards (blank-middle) also stuffs the current selection into the debug window.

Source window

A source window is used to view a text file and set breakpoints. The debugger is initially created with one source window that it uses (i.e., for loading the source of the current module on the `Display Stack` subcommands). However, you may create as many source windows as you like. Note that Bravo formatting is ignored when displaying the file.

The menu commands for a source window are as follows:

`Create`

creates a new source window at the place selected by clicking RED. There is no maximum number of source windows; however, performance decreases considerably with more than about 20 windows on the screen at once.

`Destroy`

destroys this source window after you confirm by clicking (the bullseye cursor) RED. Note that windows belonging to the debugger cannot be destroyed.

`Find`

finds the next occurence of the current selection in this source window. The search begins at the begining of the file unless the current selection is in this source window, in which case the search begins at the end of the current selection. If the search is successful, the text becomes the new selection, and if it is not visible, it is scrolled to the top of the window; otherwise, the selection remains the same and the display blinks.

`Load`

loads a file into this source window, using the current selection as a filename (appending ".mesa" if no extension is specified).

`Set Break`

uses the current selection to set a breakpoint (see **Section 4** for a discussion on breakpoints). If you select the word "PROCEDURE", a breakpoint is set on the entry to the procedure; if you select the word "RETURN", a breakpoint is set on the exit of the procedure; otherwise a breakpoint is set at the closest statement enclosing the selection. Confirmation is given by moving the selection to the place at which the breakpoint is actually set. The window must contain the source file for a module in the current configuration; in the case of multiple instances of a module, the current context must match the source file.

`Clear Break`

clears the breakpoint or tracepoint as specified above.

`Set Trace`

sets a tracepoint as specified above. Confirmation is given by moving the selection to the place at which the tracepoint is actually set.

`Position`

takes the current selection as a (decimal) character index and positions the file in this source window.

`Wrap`

The source window is created with line wrap-around turned off. Executing the `Wrap` command reverses the current state.

Section 3: Input Conventions

The input conventions of the debugger's command processor are summarized below, along with the tree for the command syntax. The command processor prompt character is ">" for the *debugger* and "/" for the *debugger nub* (actually, the character is repeated once for each nesting level of the debugger). Whenever a valid command is recognized, the debugger prompts for the parameters associated with that command (if any are required) according to the conventions described below. Typing DEL terminates the command; **?** gives a list of valid commands. When a command requires a [confirm] (CR), the debugger enters wait-for-DEL mode if an invalid character is typed.

String input

Identifiers are sequences of characters beginning with an upper or lower case letter and terminating with a space (SP) or a carriage return (CR). Source text and conditional expressions (used in setting breakpoints) must be terminated by CR, since spaces are significant in these strings. The debugger echoes a delimiting character of its own choice in order to minimize loss of information from the display.

Numeric input

A numeric parameter is a sequence of characters terminated by SP or CR which is processed by a *very simple* expression parser; it accepts constants in either octal or decimal and the operators **+**, **-**, *****, **/**. Evaluation is strictly left-to-right with no precedence or parentheses allowed. All forms of numeric constants allowed by the Mesa syntax are accepted. The default radix is octal for addresses (and input to octal commands) and decimal for everything else (unless otherwise specified in Section 4). Use the "D" suffix to force decimal interpretation and "B" to force octal.

Default values

The debugger saves the last values used as parameters to all of the commands; these values may be recalled by the escape key (ESC). The following parameters have default values which may be used or inspected by typing ESC: octal read address, octal write address, ascii read address, root configuration, configuration, module, procedure, source, condition, expression, process, address, and frame. After the default parameter is displayed by the debugger, the standard input editing characters may be used to modify it. The ESC values for octal read/write addresses are incremented after each use. Typing ESC to the command processor uses the last command as the default command (i.e., you receive the prompt for the parameters, if any, for the previously executed command).

Editing characters

The standard editing characters accepted during input are: CONTROL-A, CONTROL-H, or BS to delete a character, and CONTROL-W to delete a word.

Command Tree

This is the command tree structure for the Mesa debugger. Capitalized letters are typed by the user (in either upper or lower case): the lower case substrings are echoed by the command processor. Each command (and its parameters) is described in Section 4.

```
AScii read
ATtach Image
       Symbols
Break Entry
      Module
      Procedure
      Xit
CAse off
     on
CLear All Breaks
          Entry traces
          Traces
          Xit traces
      Break
      Entry Break
            Trace
      Module Break
             Trace
      Trace
      Xit Break
          Trace
COremap
CUrrent context
Display Configuration
        Eval-stack
        Frame
        GlobalFrameTable
        Module
        Process
        Queue
        ReadyList
        Stack
Find variable
Interpret call
Kill session
List Breaks
     Configurations
     Processes
     Traces
Octal Clear break
      Read
      Set break
      Write
Proceed
Quit
Reset context
SEt Configuration
    Module context
    Octal context
    Process context
    Root configuration
STart
Trace All Entries
          Xits
      Entry
      Module
      Procedure
      Xit
Userscreen
Worry off
      on
-- comment
```

Section 4: Debugger commands

The debugger provides facilities for managing breakpoints, examining user data symbolically, setting the context in which the user symbols are looked up, directing program control, low-level utilities, and a debugger nub used for debugging the debugger itself. The semantics of the commands are summarized below (**Section 3** contains further details regarding input conventions and **Section 6** contains details of output conventions).

Breakpoints

The break/trace commands apply to modules and procedures that are known within the current context. All breakpoints/tracepoints may optionally be conditional. If you type a **SP** after the module (or procedure) name, you receive a prompt for the condition; if you type a **CR** it terminates the command input (in the case of entry/exit breaks) or prompts for the source (in the case of text breaks). All of the breakpoint commands accept a valid **GlobalFrameHandle** as input when prompted for a **module** name.

The three valid formats of a **condition** are: **variable relation variable**, **variable relation number**, and **number** (multiple proceeds). Conditions include relations in the set {<, >, =, #, <=, >=}. The **variables** are interpreted expressions and are therefore looked up in the current context. However, if you are in a module context and wish to specify a local variable of the procedure you are setting the breakpoint in, `proc.var` may be used.

You may set break or tracepoints at the following locations in your program: entry (to a procedure), exit (from a procedure), and at the closest statement boundary preceding a specific text location within a procedure or module body. Note that breakpoints are set in all instances of a module. When a break/trace is encountered during execution, the debugger types the name of the body being broken, the text corresponding to that code location, and the address of the currently active frame; it also positions the source window with the breakpoint source at the top. If the breakpoint/tracepoint has a condition associated with it, the break/trace is taken only if the condition is satisfied. Note that the multiple proceed counter is reset after being satisfied.

If you compile a module with the cross jumping switch turned on, be warned that when setting source breakpoints, the actual breakpoint may not end up where you expect (i.e., you may break in the code of an **ELSE** clause if in fact you are really in the **THEN** clause, if they share some common code). Note that entry and exit breakpoints work just fine with cross jumping.

`Break Entry` **[proc, condition]**

> inserts a breakpoint (optionally conditional) in the procedure **proc** at the first instruction after the code which stores the input parameters in **proc**'s frame (see `Break Procedure` for further details).

`Break Module` **[module, condition, source]**

> sets a breakpoint (optionally conditional) in the program body named **module** at the *beginning* of the statement defined by the line containing the *first* instance of the string **source**. The search for **source** commences at the beginning of the **module** and extends to the end-of-file (see `Break Procedure` for further details).

`Break Procedure [proc, condition, source]`

sets a breakpoint (optionally conditional) in the procedure body named **proc** at the *beginning* of the statement containing the *first* instance of the string **source**. The search for **source** commences at the beginning of the text for **proc** and extends to the end-of-file. When the breakpoint is set, the indicator **<>** appears to the left of the source where the breakpoint has actually been set (i.e., **IF foo THEN <> some statement;**).

When a breakpoint is encountered during execution, a nested instance of the debugger is created and control transfers to the command processor, from which you may access any of the facilities described in this document. To continue execution of your Mesa program, execute the `Proceed` command; to stop execution of your program, execute the `Quit` command (see the **Breakpoints** explanation for further details).

`Break Xit [proc, condition]`

inserts a breakpoint at the *last* instruction of the procedure body for **proc** (see `Break Procedure` for further details). Note: this catches all **RETURN** statements in the procedure.

`CLear All Breaks [confirm]`

clears all breakpoints.

`CLear All Entry traces [module]`

removes all entry traces in **module**.

`CLear All Traces [confirm]`

clears all tracepoints.

`CLear All Xit traces [module]`

removes all exit traces in **module**.

`CLear Break [proc, source]`

converse of `Break Procedure`.

`CLear Entry Break [proc]`

converse of `Break Entry`.

`CLear Entry Trace [proc]`

converse of `Trace Entry`.

`CLear Module Break [module, source]`

converse of `Break Module`.

`CLear Module Trace [module, source]`

converse of `Trace Module`.

`CLear Trace [proc, source]`

converse of `Trace Procedure`.

`CLear Xit Break [proc]`

converse of `Break Xit`.

`CLear Xit Trace [proc]`

converse of `Trace Xit`.

`List Breaks [confirm]`

lists all breakpoints, their type (entry, exit, source), and the procedure and/or module name in which they are found. For source breakpoints, the source text is also displayed.

`List Traces [confirm]`

lists all tracepoints (cf. `List Breaks`).

`Trace All Entries [module]`

sets a trace on the entry point to each procedure in **module** (cf. `Trace Entry`).

`Trace All Xits [module]`

sets a trace on the exit point of each procedure in **module** (cf. `Trace Xit`).

`Trace Entry [proc, condition]`

sets a trace on the entry point to the procedure **proc.** When an entry tracepoint is encountered, **proc**'s parameters are displayed and you are prompted with ">" (see `Trace Procedure` for further details).

`Trace Module [module, condition, source]`

sets a trace in the program body named **module** at the *beginning* of the statement defined by the line containing the *first* instance of the string **source.** The search for **source** commences at the beginning of the **module** and extends to the end-of-file (see `Trace Procedure` for further details).

`Trace Procedure [proc, condition, source]`

sets a trace (optionally conditional) in the procedure body named **proc** at the *beginning* of the statement defined by the line containing the *first* instance of the string **source.** The search for **source** commences at the beginning of the text for **proc** and extends to the end-of-file.

When the tracepoint is reached, you may respond to the ">" prompt with the standard

replies (cf. `Display Stack`) for listing the parameters, return values, or local variables. In order to continue execution, execute the **Q** (or **q**) subcommand. In addition to the standard replies, you may also type **B** (or **b**) which creates a nested instance of the debugger and sends control to the command processor (as in breakpoints), from which you may access any of the facilities described in this document (see the **Breakpoints** explanation for further details).

`Trace Xit [proc, condition]`

sets a trace on the exit of the procedure **proc.** When an exit tracepoint is encountered, **proc**'s return values are displayed and you are prompted with ">" (see `Trace Procedure` for further details).

Display runtime state

The scope of variable lookup is limited to the current context (unless otherwise specified below to be the current configuration). What this means is the following: if the current context is a local frame, the debugger examines the local frame of each procedure in the call stack (and its associated global frame) following return links until the root of the process is encountered; if the current context is a module (global) context, just the global frame is searched. If the variable you wish to examine is not within the current context, there are commands provided which change between contexts. Upper/lower case distinction is not observed in looking up variables (you may change this default setting with the `CAse` command); however, case shifts are always significant in source strings used in setting breakpoints.

`AScii read [address, n]`

displays **n** (decimal) characters starting at **address** (octal).

`CAse off [confirm]`

ignores the distinction between upper and lower case during symbol lookup. This is the default state when you enter the debugger, except that upper and lower case are always significant in source strings for breakpoints (see **Display runtime state** explanation).

`CAse on [confirm]`

observes the distinction between upper and lower case during symbol-lookup. Once set, this state persists until you execute a `CAse off` command.

`Display Configuration`

displays the name of the current configuration followed by the module name, corresponding global frame address, and instance name (if one exists) of each module in the current configuration.

`Display Frame [address]`

displays the contents of a frame, where **address** is its octal address (useful if you have several instances of the same module.)

`Display GlobalFrameTable`

displays the module name and corresponding global frame address, pc, codebase, and gfi of all entries in the global frame table.

`Display Module [module]`

displays the contents of a global frame, where **module** is the name of a program in the current configuration.

`Display Process [process]`

displays interesting things about a process. This command shows you the **ProcessHandle** and the frame associated with **process**, and whether the **process** is waiting on a monitor or condition variable (`waiting ML` or `waiting CV`). Note that a process can be on one and only one queue (associated with a condition, monitor, ReadyList, etc.). Then you are prompted with a ">" and you enter process subcommand mode. A response of N displays the next process; S displays the source text and loads and positions the sourcefile in the source window; L just displays the source text; R displays the root frame of the process; P displays the priority of the process; and Q or DEL terminates the display and returns you to the command processor. Note that either a variable of type PROCESS (returned as the result of a FORK) or an octal **ProcessHandle** is acceptable as input to this command (**process** is an interpreted expression).

`Display Queue [id]`

displays all the processes waiting on the queue associated with **id**. For each process, you enter process subcommand mode. The semantics of the subcommands remain the same as in `Display Process`, with the exception of `N`, which in this case follows the link in the process. This command accepts either a condition variable, a monitor lock, a monitored record, a monitored program, or an octal pointer. Note that **id** is an interpreted expression; if **id** is simply an octal number, you are asked whether it is a condition variable (i.e., `Display Queue: 175034B, condition variable? [Y or N]`).

`Display ReadyList`

displays all the processes waiting on the queue associated with the **ReadyList**, the list of processes ready to run. For each process, you enter process subcommand mode; the semantics of the subcommands are the same as in `Display Queue`.

`Display Stack`

follows down the procedure call stack. At each frame, the corresponding procedure name and frame address are displayed. You are prompted with a ">". A response of V displays all the frame's variables; P displays the input parameters; R displays the return values (those which are "named" in the RETURNS part of the body declaration); N moves to the next frame; J, **n(10)** jumps down the stack **n** (decimal) levels (if **n** is greater than the number of levels it can advance, the debugger tells you how far it was able to go); S displays the source text and loads and positions the sourcefile in the source window; L just displays the source text; and Q or DEL terminates the display and returns you to the command processor. When the current context is a global frame, the `Display Stack` subcommands J and N are disabled. When the debugger cannot find the symbol table for a frame on the call stack, only the J, N, and Q subcommands are allowed. For a complete description of the output format, see Section 6.

`Find variable [id]`

displays the contents and module location of the variable named **id**, searching through all the modules in the current configuration.

`Interpret call [proc]`

calls the procedure **proc**, after prompting for parameters one word at a time. The parameters must be constants (the default radix is decimal) or simple identifiers. Note that unlike the interpreter, this command does no type checking. However, this command is useful since the interpreter cannot take breakpoints during a procedure call.

Current context

The current context is used to determine the domain for symbol lookup. There are commands provided which make it simple to display the current context, to display all the configurations and processes, to restore the starting context, and to change between contexts.

`CUrrent context`

displays the name and corresponding global frame address (and instance name if one exists) of the current module, the name of the current configuration, and the **ProcessHandle** for the current process.

`List Configurations [confirm]`

lists the name and instance name (if one exists) of each configuration that is loaded, beginning with the last configuration loaded. If you wish to see more information about a particular configuration, use the `Display Configuration` command.

`List Processes [confirm]`

lists all processes by **ProcessHandle** and frame. If you wish to see more information about a particular process, use the `Display Process` command.

`Reset context [confirm]`

restores the context which this instance of the debugger had upon entering the session.

`SEt Configuration [config]`

sets the current configuration to be **config**, where **config** is nested within the root configuration that is current. This command is useful for "jumping" further into the nested block structure of a configuration.

`SEt Module context [module]`

changes the context to the program module whose name is **module** (within the current configuration). If there is more than one instance of **module**, the debugger lists the frame address of each instance and does *not* change the context. You may use the `SEt Octal context` command to set the context to a frame address.

`SEt Octal context [address]`

changes the context to the frame whose address is **address** (cf. `SEt Module context`). This is useful when there are several instances of the same module.

`SEt Process context [process]`

sets the current process context to be **process** and sets the corresponding frame context to be the frame associated with **process**. Upon entering the debugger, the process context is set to the currently running process. Note that either a variable of type **PROCESS** (returned as the result of a **FORK**) or an octal **ProcessHandle** is acceptable as input to this command.

`SEt Root configuration [config]`

sets the current configuration to be **config**, where **config** is at the outermost level (of its configuration). This command is sufficient for simple configurations of only one level. It is also useful in getting you to the outermost level of nested configurations, from which you may move "in" using `SEt Configuration`.

Program control

There are commands provided which allow you to determine the flow of control between the debugger and your program.

`Kill session [confirm]`

ends your debugging session, cleans up the state as much as possible, and returns to the *Alto Executive*. Use this command instead of **SHIFT-SWAT** or the boot button to leave the debugger.

`Proceed [confirm]`

continues execution of the program (i.e., proceeds from a breakpoint, resumes from an uncaught signal).

`Quit [confirm]`

returns control to the dynamically enclosing instance of the debugger (if there is one). Executing a `Quit` has the effect of cutting the runtime stack back to the nearest enclosing instance of the debugger. `Quit`ting from the outermost level of the debugger returns you to the *Mesa Executive*; `Quit`ting from the *Mesa Executive* returns you to the *Alto Executive*.

`STart [address]`

starts execution of the module whose frame is **address**. Unlike the language **START** statement, no parameters may be passed.

`Userscreen [confirm]`

swaps to the user world for a look at the screen. Control is returned to the debugger with the SWAT key.

Low-level facilities

There are additional commands provided which allow the user to examine (and modify) what is going on in the underlying system.

`ATtach Image [filename]`

specifies the **filename** to use as an image file when the debugger has been bootloaded. It is useful when the user core image has been clobbered. The default extension for **filename** is **".image"**.

`ATtach Symbols [globalframe, filename]`

attaches the **globalframe** to **filename**. This is useful for allowing you to bring in additional symbols for debugging purposes not initially anticipated. The default extension for **filename** is **".bcd"**.

`COremap [confirm]`

prints the following information (in octal) about the segments currently in memory: memory page number, memory address, file page number (if it is a file), number of pages, state {**busy**, **free**, **data**, **file**}, serial number (if it is a file), class {**code**, **other**}, access {**Read**, **Write**, **Append**}, lock. If the class is **code**, the module name is also given. Holding down ↑DEL terminates the printout.

`Display Eval-stack`

displays the contents of the Mesa evaluation stack (in octal), useful for low-level debugging or for displaying the (un-named) return values of a procedure which has been broken at its exit point. This command is only useful when reaching octal breakpoints because the eval-stack is empty between statements.

`Octal Clear break [globalframe, bytepc]`

converse of `Octal Set break`. (Note: these *octal* commands are low-level debugging aids for system maintainers who must diagnose the higher-level debugging aids and system.)

`Octal Read [address, n]`

displays the **n** (decimal) locations starting at **address**.

`Octal Set break [globalframe, bytepc]`

sets a breakpoint at the byte offset **bytepc** in the code segment of the frame **globalframe**.

`Octal Write [address, rhs]`

stores **rhs** (octal) into the location **address**; the default for **rhs** is the current contents of **address**.

`Worry on [confirm]`

taking a breakpoint in worry mode brings you into the debugger with the user core image undisturbed (i.e., no cleanup procedures are invoked, no frames are allocated, and memory is left unchanged). All of the debugger commands are allowed, with the exception of `Interpret call`, `STart` and `Quit`.

`Worry off [confirm]`

turns off worry mode (this is the default state upon entering the debugger).

`-- [comment]`

inserts a comment into the debugger's typescript file. Input is ignored after the dashes until a carriage return (CR) is typed.

`↑Debug [confirm]`

invokes the *debugger nub* which prompts with a "`//`". See **Debugger nub** for further details about the capabilities of the nub.

Debugger nub

The *nub* is a part of the debugger that contains primitive facilities for debugging the debugger as well as providing a minimal signal catcher and interrupt handler.

Typing `↑D` (to the command processor of the *debugger*) brings you into the *nub* with a "`//`" prompt. The following limited set of commands are available in the *nub*: `New`, `Read`, `Write`, `Proceed`, `Quit`, and `Start`. The semantics of the `New` command are explained below; the other commands have already been explained above (`Read` and `Write` are the same as `Octal Read` and `Octal Write`).

`New [filename]`

is just like the `New` command in the *Mesa Executive*.

Section 5: Debugger Interpreter

The Mesa debugger contains an interpreter that handles a subset of the Mesa language; it is useful for common operations such as assignments, dereferencing, indexing, field access, addressing, and simple type conversion. It is a powerful extension to the current debugger command language, as it allows you to more closely specify variables while debugging, thus giving you more complete information with fewer keystrokes. A specific subset of the Mesa language is acceptable to the interpreter (see below for details on the grammar). Several specialized notations (abbreviations) have been introduced in the interpreter grammar; note that these are not part of the Mesa language (valid only for debugging purposes).

Statement Syntax

Typing space (SP) to the command processor enables interpreting mode. At this point the debugger is ready to interpret any expression that is valid in the (debugger) grammar.

Multiple statements are separated by semicolons; the last statement on a line should be followed by a carriage return (CR). If the statement is a simple expression (not an assignment), the result is displayed after evaluation.

For example, to perform an assignment and print the result in one command, you would type **foo ← exp; foo**.

Loopholes

A more concise LOOPHOLE notation has been introduced to make it easy to display arbitrary data in any format. The character "%" is used to denote LOOPHOLE**[exp, type]**, with the expression on the left of the %, and the **type** on the right.

For example, the expression **foo % short red Foo** means LOOPHOLE the type of the variable **foo** to be a **short red Foo** and display its value.

Note that since **LONG** in not in the interpreter's grammar, in order to loophole something into a **LONG type** you must have a variable of type **LONG type** in your program.

Subscripting

There are two types of interval notation acceptable to the interpreter. The notation **[a . . b]** means start at index **a** and end at index **b**. The notation **[a ! b]** means start at index **a** and end at index **(a+b−1)**.

For example, the expressions MEMORY**[4 . . 7]** and MEMORY**[4 ! 4]** both display the octal contents of memory locations **4** through **7**. Note that the interval notation is only valid for display purposes, and therefore is not allowed as a **LeftSide** or inside other expressions.

Module Qualification

To improve the performance of the interpreter, the **$** notation has been introduced to distinguish between module and record qualification. The character **$** indicates that the name on the left is a module, in which to look up the identifier or TYPE on the right. If a module cannot be found, it uses the name as a file (usually a definitions file). A valid octal frame address is also acceptable as the left argument of **$**.

For example, **FSP$TheHeap** means look in the module **FSP** to find the value of the variable **TheHeap**. In dealing with variant records, be sure to specify the variant part of the record before the record name itself (ie., **foo % short red FooDefs$Foo**, *not* **foo % FooDefs$short red Foo**).

Type Expressions

The notation "**@type**" is used to construct a POINTER TO **type**. This notation is used for constructing types in LOOPHOLEs (ie., **@foo** will give you the type POINTER TO **foo**).

Sample Expressions

Here are some sample expressions which combine several of the rules into useful combinations:

If you were interested in seeing which procedure was associated with the third keyword of the menu belonging to a particular window called **myWindow**, you would type:

```
myWindow.menu.array[3].proc
```

which might produce the following output:

```
CreateWindow (PROCEDURE in WEWindows, G: 120134B).
```

The basic arithmetic operations are provided by the interpreter (with the same precedence rules as followed by the Mesa compiler).

```
3 + 4 MOD 2 ; (3 + 4) MOD 2
```

would produce the following output:

```
3
1.
```

Radix conversion between octal and decimal can be forced using the loophole construct; for example, **exp%CARDINAL** will force octal output and **exp%INTEGER** will force decimal.

A typical sequence of expressions one might use to initialize a record containing an array of **Foo**s and display some of them would be:

```
rec.array ← DESCRIPTOR[FSP$AllocateHeapNode[n*SIZE[FooDefs$Foo]], n];
InitArray[rec.array]; rec.array[first..last].
```

Grammar

```
StmtList          ::= Stmt | StmtList; Stmt

AddingOp          ::= + | -

BuiltinCall       ::= LENGTH [ LeftSide ] | BASE [ LeftSide ] |
                      DESCRIPTOR [ Expression ] |
                      DESCRIPTOR [ Expression , Expression ] |
                      SIZE [ TypeSpecification ]

Expression        ::= Sum

ExpressionList    ::= Expression | ExpressionList, Expression |

Factor            ::= - Primary | Primary

Interval          ::= Expression .. Expression | Expression ! Expression

LeftSide          ::= identifier | Literal | MEMORY [ Expression ] |
                      LeftSide Qualifier | ( Expression ) Qualifier |
                      identifier $ identifier | numericLiteral $ identifier

Literal           ::= numericLiteral |
                      stringLiteral | -- all defined outside the grammar
                      characterLiteral

MultiplyingOp     ::= * | / | MOD

Primary           ::= LeftSide | ( Expression ) | @ LeftSide | BuiltinCall

Product           ::= Factor | Product MultiplyingOp Factor

Qualifier         ::= . identifier | ↑ | % | % TypeSpecification | [ ExpressionList ]

Stmt              ::= Expression | LeftSide ← Expression | MEMORY [ Interval ] |
                      LeftSide [ Interval ] | ( Expression ) [ Interval ]

Sum               ::= Product | Sum AddingOp Product

TypeConstructor   ::= @ TypeSpecification

TypeIdentifier    ::= INTEGER | BOOLEAN | CARDINAL |
                      CHARACTER | STRING | UNSPECIFIED |
                      identifier | identifier $ identifier |
                      identifier TypeIdentifier

TypeSpecification ::= TypeIdentifier | TypeConstructor
```

Section 6: Output Conventions

The debugger uses information about the types of variables to decide on an appropriate output format. Listed below are the built-in types which the debugger distinguishes and the convention used to display instances of each type.

ARRAY, ARRAY DESCRIPTOR

displays the first, second and last values of the array, unless the number of elements is "small", e.g., `a=(10)[Vector[x: 0, y:0], Vector[x: 1, y: 1], ... , Vector[x: 9, y:9]]`. The parenthesized value to the right of the "=" is the length of the array.

BOOLEAN

displays **TRUE** or **FALSE**. Since **BOOLEAN** is an enumerated type = **{FALSE, TRUE}**, values outside this range are indicated by a ? (probably an uninitialized variable).

CHARACTER

displays a printing character (c) as `'c`. A control character (X) other than **BLANK, RUBOUT, NUL, TAB, LF, FF, CR,** or **ESC** is displayed as ↑X. Values greater than **177B** are displayed in octal.

ENUMERATED

displays the identifier constant used in the enumerated type declaration. For example, an instance **c** of the type **ChannelState: TYPE = {disconnected, busy, available}** is displayed as `c=busy`. If the value is out of range (probably an uninitialized variable), a ? is displayed.

INTEGER

always displays a decimal number. Uniformly, numeric output is decimal unless terminated by "**B**" (octal).

LONG

numbers are displayed following the same conventions as short numbers, i.e., **LONG CARDINAL** and **LONG UNSPECIFIED** are displayed in octal, **LONG INTEGER** in decimal.

POINTER

displays an octal number, terminated with an "↑", i.e., `p=107362B↑`.

PORT

displays two octal numbers, i.e., `p = PORT [0, 172520B]`.

PROCEDURE, SIGNAL, ERROR

displays the name of the procedure (with its local frame) and the name of the program module in which it resides (with its global frame), e.g., `GetMyChar, L: 165064B (in CollectParams, G: 166514B)`. Procedure variables which do not contain valid descriptors generate a "?".

PROCESS

displays a **ProcessHandle** (pointer to a **ProcessStateBlock**), i.e., `p = PROCESS [2002B]`. See the process section of the *Mesa System Documentation* for further details.

RECORD

the record's type identifier is followed by a bracketed list of each field name and its value. For example, an instance **v** of the record **Vector: RECORD [x,y: INTEGER]** is displayed as `v=Vector[x: 9, y: -1]`.

STRING

displays the name of the string, followed by its current length, its maximum length, and the string body, e.g., `s=(3,10)"foo"`. If the string is longer than 60 characters, the first 40 and the last 10 are displayed. If the string is NIL, `s=NIL` is displayed.

SUBRANGE

displays an octal number if the upper limit exceeds **77777B**, decimal otherwise.

Listed below is the convention used to display context information throughout the debugger.

`ProcedureName, L: nnnnnnB (in ModuleName, G: nnnnnnB) --local frame`

A local context is displayed as the procedure name with its local frame, followed by the module name and its global frame.

`ModuleName, G: nnnnnnB --global frame`

A global context is displayed as the module name and its global frame. If the global frame is followed by *, i.e., `nnnnnnB*`, it is a copy created by the NEW construct.

Section 7: Signal and Error Messages

The following messages are generated by the debugger. Wherever possible, there is also an explanation of what might have caused the problem and what you can do about it.

Breakpoints

`-- not allowed here!`

An attempt was made to set a breakpoint on an opcode on which it is not allowed.

`-- does not return!`

An attempt was made to set an exit breakpoint on a procedure in which the return statement is not in the correct location (check the code for your program).

`Breakpoint not found!`

You have swapped to the debugger when the breakpoint information (frame, pc, etc.) cannot be found (check the code for your program).

Command execution

`... aborted`

Execution of the current command has been aborted (↑DEL has been held down).

`!Command not allowed`

Execution of the current command is not allowed since the state of the user core image appears to be invalid.

`Core image not healthy, can't swap!`

You may only `Quit` or terminate the session (`Kill session`) after the debugger has been bootloaded.

Displaying the stack

`No previous frame!`

The end of the call stack has been reached.

`No symbol table for nnnnnnB`

The symbol table file corresponding to the frame `nnnnnnB` is missing; any attempt to symbolically reference variables in this module will fail. (In general, this message is a warning.)

Entering the debugger

`*** Debugger Bootloaded! ***`

Appears at the top of the DEBUG.LOG window after you have booted from the MESADEBUGGER file (by typing `Bootfrom MesaDebugger` to the *Alto Executive*). This gets you into the debugger and allows you to look at what was going on. However, you may not proceed after the debugger has been bootloaded.

`*** Fatal System Error (Punt) ***`

Appears when the system can no longer continue, often a result of running out of memory or frame space. (`Display Stack` for several levels and look at the variables to try to figure out what was going on. A `Coremap` may also help to explain the memory space problem.)

`*** Interrupt ***`

Appears at the top of the DEBUG.LOG window after you have entered the debugger via interrupt mode (↑SWAT has been held down).

`ResumeError!`

You have attempted to continue execution from an ERROR. This may occur both in the situation described below or as the result of a programming error. (The debugger does not support resuming SIGNALs which return values.)

`*** uncaught SIGNAL SoS (in MayDay)`

The user program has raised a SIGNAL (ERROR) which no one dynamically nested above the SIGNAL invocation was prepared to catch. The debugger prints the name of the SIGNAL, lists its parameters (if any), creates a new instance of the debugger, and gives control to the command processor. At this point you may, for example, display the stack to see who raised the uncaught SIGNAL.

If the semantics of the situation permit, you may proceed execution at the point of the SIGNAL's invocation by issuing a `Proceed` command. Alternatively, you retire to the dynamically enclosing instance of the debugger by issuing a `Quit` command. If the SIGNAL actually was an ERROR and you elect to `Proceed`, you get a `ResumeError`.

Note: if the debugger does not have access to the required symbol tables, the information will be printed in octal. For standard Mesa software, listings which decode these numbers are available (see the *Mesa Users Handbook*).

Interpreter

```
! Invalid Type.
! Invalid Expression.
! Invalid Character.
! Invalid Number.
! Not Implemented.
```

The interpreter has been given an invalid expression.

Parameters

`-- is an invalid identifier!`

The first character of your identifier is not an upper or lower case letter.

`!Number`

An invalid number has been typed.

`!String too long`

The string you have just typed is too long. String parameters are subject to the following restrictions: identifiers and string constants are limited to 40 characters, source-text parameters are limited to 60 characters, and conditions and expressions are limited to 100 characters.

Symbol Lookup

`!xyz`

The variable named `xyz` cannot be found.

`!File: xyz`

The file named `xyz` cannot be found.

`!File: --compressed symbols--`

The symbol file is compressed.

`--- has incorrect version!`

The symbol file has an incorrect version stamp.

`!String: xyz`

The search for the string `xyz` has failed.

`!Code links.`

The module in question has links in the code; variable lookup with code links is not implemented in the debugger.

`!Insufficient VM [n]`

The debugger has run out of memory for swapping; `[n]` is the number of pages needed. At this point you should check to see you have the correct context and the correct parameters for your command and try executing the command again.

Validity checking

```
--- is not a frame!
--- is not a global frame!
--- is a clobbered frame!
--- has a NULL returnlink!
--- has a clobbered accesslink!
--- is an invalid ProcessHandle!
--- is an invalid image file!
```

The structure in question appears to be clobbered (invalid in some way).

Debugger Summary

Version 5.0

AScii read [address, n]
ATtach Image [filename]
Symbols [globalframe, filename]
Break Entry [proc, condition]
Module [module, condition, source]
Procedure [proc, condition, source]
Xit [proc, condition]
CAse off **[confirm]**
on **[confirm]**
CLear All Breaks **[confirm]**
Entry traces [module]
Traces **[confirm]**
Xit traces [module]
Break [proc, source]
Entry Break [proc]
Trace [proc]
Module Break [module, source]
Trace [module, source]
Trace [proc, source]
Xit Break [proc]
Trace [proc]
COremap **[confirm]**
CUrrent context
Display Configuration
Eval-stack
Frame [address]
GlobalFrameTable
Module [module]
Process [process] - **l,n,p,q,r,s**
Queue [id]
ReadyList
Stack - **j,l,n,p,v,r,s,q**
Find variable [id]
Interpret call [proc]
Kill session **[confirm]**
List Breaks **[confirm]**
Configurations **[confirm]**
Processes **[confirm]**
Traces **[confirm]**
Octal Clear break [globalframe, bytepc]
Read [address, n]
Set break [globalframe, bytepc]
Write [address, rhs]
Proceed **[confirm]**
Quit **[confirm]**
Reset context **[confirm]**
SEt Configuration [config]
Module context [module]
Octal context [address]
Process context [process]
Root configuration [config]
STart [address]
Trace All Entries [module]
Xits [module]
Entry [proc,condition]
Module [module, condition, source]
Procedure [proc, condition, source]
Xit [proc, condition]
Userscreen **[confirm]**
Worry off **[confirm]**
on **[confirm]**
-- [comment]

Debugger Interpreter Grammar

Version 5.0

```
StmtList          ::= Stmt | StmtList; Stmt

AddingOp          ::= + | -

BuiltinCall       ::= LENGTH [ LeftSide ] | BASE [ LeftSide ] |
                      DESCRIPTOR [ Expression ] |
                      DESCRIPTOR [ Expression , Expression ] |
                      SIZE [ TypeSpecification ]

Expression        ::= Sum

ExpressionList    ::= Expression | ExpressionList, Expression |

Factor            ::= - Primary | Primary

Interval          ::= Expression .. Expression | Expression ! Expression

LeftSide          ::= identifier | Literal | MEMORY [ Expression ] |
                      LeftSide Qualifier | ( Expression ) Qualifier |
                      identifier $ identifier | numericLiteral $ identifier

Literal           ::= numericLiteral |
                      stringLiteral | -- all defined outside the grammar
                      characterLiteral

MultiplyingOp     ::= * | / | MOD

Primary           ::= LeftSide | ( Expression ) | @ LeftSide | BuiltinCall

Product           ::= Factor | Product MultiplyingOp Factor

Qualifier         ::= . identifier | ↑ | % | % TypeSpecification | [ ExpressionList ]

Stmt              ::= Expression | LeftSide ← Expression | MEMORY [ Interval ] |
                      LeftSide [ Interval ] | ( Expression ) [ Interval ]

Sum               ::= Product | Sum AddingOp Product

TypeConstructor   ::= @ TypeSpecification

TypeIdentifier    ::= INTEGER | BOOLEAN | CARDINAL |
                      CHARACTER | STRING | UNSPECIFIED |
                      identifier | identifier $ identifier |
                      identifier TypeIdentifier

TypeSpecification ::= TypeIdentifier | TypeConstructor
```

Wisk Summary

Version 5.0

WHAT WISK MOUSE BUTTONS DO:

	Left Scroll Bar	Text Area
RED	Scroll Up	Select/Extend characters
YELLOW	Thumb	Select/Extend words
BLUE	Scroll Down	Menu Commands

WINDOW HEADER COMMANDS:

	Left	Middle	Right
RED	Top/Bottom	Zoom	Top/Bottom
YELLOW	Grow (corner)	Grow (edge)	Grow (corner)
BLUE	Move	Size	Move

STANDARD WINDOW MENU COMMANDS:

Move	Size	Bottom
Grow	Top	Zoom

DEBUG WINDOW MENU COMMANDS:

Alter Bitmap [selection]	Move Boundary	Stuff It [selection]

SOURCE WINDOW MENU COMMANDS:

Create	Load [selection]	Set Trace [selection]
Destroy	Set Break [selection]	Position [selection]
Find [selection]	Clear Break [selection]	Wrap