Mesa Debugger Documentation

Version 6.0
October 1980

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

OFFICE PRODUCTS DIVISION
Systems Development Department
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

Preface v

1. Overview 1

2. User Interface 3

3. Input Conventions 9

4. Debugger commands 10

5. Debugger Interpreter 20

6. Output Conventions 23

7. Signal and Error Messages 26

Appendices

Debugger Summary 35

Debugger Interpreter Grammar 37

Wisk Summary 39

Preface

October 1980

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 amount of 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 comments on 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 runtime and debugging facilities differ in their relationship to the user program. When you invoke `Mesa.image`, it provides the code necessary for your program to communicate with the debugger; it resides with the user program. An optional *Mesa Executive* serves the function of an executive when the Mesa system is first started (see the *Mesa System Documentation* for further details). The *debugger*, however, resides in a different core image which is loaded when called for; it operates with a complete *world-swap*.

Installing the debugger

Before a client program can use the Mesa debugger, the debugger must be installed. This operation saves the debugger's core image. Typing `XDebug` to the Alto Executive automatically installs the debugger. Other programs may be loaded into the debugger by including their names on the command line. The Debugger now requires a strike font named MesaFont.strike or SysFont.strike. See the *Mesa User's Handbook* for further details.

Invoking the debugger

There are several ways of invoking the debugger. One method is to issue the `Debug` command to the *Mesa Executive*; this brings you into the debugger, ready to execute a command. Invoking a BCD from the Alto Executive with the `/d` switch causes `Mesa.image` to go to the debugger after loading the BCD, but before starting it. See the Mesa User's Handbook for complete information on various debugging switches. If you wish to enter the debugger at any time (i.e., while your program is running), **^SWAT** interrupts your program. (If you really get in trouble **Section 7** contains details on bootloading the debugger.)

In the course of running your program, you may enter the debugger for several other reasons. Your program may generate an uncaught signal, execute a breakpoint/tracepoint that has been placed in your program, or encounter 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 that invokes a collection of procedures for managing breakpoints, examining user data symbolically, and setting the context in which user symbols are looked up.

When receiving commands, the debugger extends each input character to the maximal unique string that 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; typing **^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 manager which comes with your debugger (see **Section 2** for further details).

Current context

Interpreting symbols (including displaying variables, setting breakpoints, and calling procedures) occurs in 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 Root configuration, SEt Module context`), to display the current context (`CUrrent context`), and to examine the current dynamic state (`Display Stack`).

Looking up symbols

Whenever the debugger needs symbols to display some information, it searches for the original compiler-output `BCD` before looking for symbols where they were last copied by the binder. Types used, but not declared, within a module are looked up using the same algorithm as in the compiler. If the module containing the original declaration is unavailable, the debugger uses whatever information has been copied into the symbol table of the module using that type.

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`). **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 *Tajo Functional Specification*. For more information on the window package see the *Vista Functional Specification*.

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 functions specific to each window. Note: these functions are best understood by trying them as you read this document.

Typein

Typein goes to the window containing the cursor, regardless of whether that window is on top. Typein is directed to the Debugger if the cursor is not in any window. Source windows will accept input until a file is loaded; they then direct typein to the Debugger (unless they are editable; see below). Type ahead of mouse clicks and keystrokes is permitted. The following function keys are implemented (see the section on editing for an explanation of the functions):

Function	ADL Keyboard	Microswitch Keyboard	Keyset	Control Key
`Cut`	DEL	DEL	Keyset5	^C
`Paste`	LF	LF	Keyset1	^F
`Next`	FL3	(none)	Keyset3	^N
`Replace`	FL4	(none)	Keyset4	^R
`Swat`	FR1	Spare3	(none)	(none)
`Stuff`	FR4	Spare2	Keyset2	^S
`Back Word`	BW	Spare1	(none)	^W
`Replace/Next`	FR5	(none)	(none)	^K

Selections

There is only one selection at any time (*not one per window*). Clicking **RED** once selects a character, clicking twice selects a word, three times a line, etc. The selection can be extended to the left or right with **BLUE**; a character selection is extended by characters, a word selection by words, and so on. The first and last characters of the selection are always extended by characters, regardless of the current mode of selection. The current selection is video reversed.

Scrollbars

Scrollbars do not occupy a dedicated part of the window, but instead appear on top of the left edge; you can "see through" them. To obtain a scroll bar, move left just past the edge of the window, then move right slightly, back into the window. Scrolling commands are initiated by moving into the scroll bar 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 **YELLOW** 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 **YELLOW** 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 **YELLOW** mouse button. Except where otherwise noted below, clicking the **RED** mouse button over a menu command causes the command to be executed. Whenever a menu command requires the user to click **RED** for confirmation, the user may click **BLUE** instead to abort the command.

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.

Standard Menus

The standard *Window Manager* menu commands are as follows:

`Move`

> repositions the corner of the window closest to the cursor in any direction. Clicking **RED** positions that corner of the window to the cursor location. Note that this command 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 may free up much of the space associated with the window. Sizing the `Debug.log` closes and truncates the file.

`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.

`Deactivate`

deactivates the selected window; it will no longer appear on the screen and in most cases it will free the resources being used by that window. The window's name is added to a menu of deactivated windows; this menu is available outside all windows. The window may be made active again by selecting its menu item.

A *Text Ops* menu is supplied with the `Debug.log` and source windows in addition to the window manager menu. It contains the following commands:

`Find`

finds the next occurence of the current selection in this window. The search begins at the first character visible in the window unless the current selection is in this 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; if it is not visible, it is scrolled to the top of the window; otherwise, the selection remains the same and the display blinks.

`Position`

takes the current selection as a decimal character index and positions the file in the subwindow where the menu was invoked.

`Split`

divides in two the subwindow where the menu was invoked. Feedback is similar to that in Laurel: the split line can be picked up at the small box on the right using **RED** and moved vertically. The subwindow is destroyed by moving the split line off the top or bottom of the subwindow.

`Normalize Insertion`

scrolls a (sub)window containing an insert point (`e.g., Debug.log` and editable source windows), so that the line containing the type-in point is at the top.

`Normalize Selection`

scrolls the (sub)window so that the line containing the leftmost position of the current selection is at the top.

`Wrap`

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

Name Stripe / Tiny Windows

The name stripe and the top half of tiny windows function as accelerators for the window manager menu commands (`Move`, `Grow`, `Size`, `Top`, `Bottom`, and `Zoom`); they both video reverse when the cursor is in the sections that activate the window manager commands. These window operations may be invoked by positioning the cursor in the left, middle, or right region of the name stripe and clicking one of the mouse buttons; the top half of a tiny window works the same. The functions are as described above with these two exceptions:

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.

Grow: does not apply to tiny windows.

The header commands are as follows:

Mouse Button	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.

Source windows

A source window is used to view a text file, edit 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.

In addition to the standard menus, the source window has two additional menus. The *Source Ops* menu contains the following commands:

`Create`

creates a new source window at the place selected by clicking **RED**.

`Destroy`

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

`Attach`

> tells the debugger to ignore the time stamp in the source file when setting breaks. Timestamps are discussed in **Section 4**.

`Set Break`

> uses the current selection to set a breakpoint (breakpoints are discussed in **Section 4**). If you select the word "**PROCEDURE**" or "**PROC**", 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. Note that if the module was compiled with cross jumping, breaks may be set in unpredictable places. 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. If there are multiple instances of a module, the current context must match the source file.

`Set Trace`

> sets a tracepoint at a location specified as in `Set Break` above. Confirmation is given by moving the selection to the place at which the tracepoint is actually set.

`Clear Break`

> clears the breakpoint or tracepoint at the location specified as above.

The breakpoint commands and `Attach` are available in the *Source Ops* menu only if a file has been loaded into the window.

The *File Ops* menu includes the following commands:

`Load`

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

`Store`

> creates a file whose name is the current selection and stores the contents of the window in it. If the file already exists, overwriting it requires confirmation. After the `Store` command completes, the file is no longer editable.

`Save`

> stores the contents of the window in its current file (this always requires confirmation). A Bravo style "$" file is created that is a copy of the unedited version. After the `Save` command completes, the file is no longer editable.

`Edit`

> enables editing of the currently loaded read-only file (see below). Empty windows are always editable, but because they have no backing store (until they are `Store`d or `Save`d on a file), the amount of information in the window should be kept small.

`Reset`

> discards all edits that have been made to the window (during this session). If the file was editable, the file is no longer editable, otherwise the window is made empty.

`Time`

> replaces the current selection with the current date and time.

The `Edit` command is available only if a file has been associated with the window (by a previous `Load`, `Store`, `Save`, or if it is empty); `Reset`, `Store`, and `Save` apply only if the window has been edited.

Editing

The standard source window facilities provide a simple cut and paste style editor. Editing is modeless and is accomplished by simply moving the insert point and typing the desired text. (Note that unlike Bravo, the insert point is independent of the location of the selection.) Backspace and backword functions (BS and BW) are always available. The following functions are also provided:

^RED		Moves the insert point to the cursor position.
DEL (Keyset5)	^C	Cut deletes the current selection and puts the deleted text in the TrashBin (see LF).
LF (Keyset1)	^F	Paste inserts the TrashBin at the insert point (see DEL).
FL4 (Keyset4)	^R	Replace does a cut and moves the insert point to the place where the text was deleted.
FR4 (Spare2)	^S	Stuff inserts the current selection at the insert point.

Caution: The editing facilities are designed not to alter the original file until it is `Saved` or `Stored`, much like Bravo. This is, however, a new facility and should be used with caution; it is intended to support a small number of localized changes, not program creation or massive changes.

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**); *identifiers must be typed with their correct capitalization.* 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**. If the parameter is not a numeric constant it will be processed by the Debugger Interpreter (see **Section 5**); any expression which evaluates to a number is legal (the target type must be **(LONG) INTEGER**, **CARDINAL**, or **UNSPECIFIED**). The default radix is octal for addresses (and input to octal commands) and decimal for everything else (unless otherwise specified in **Section 4**). The "**D**" or "**d**" suffix forces decimal interpretation; "**B**" or "**b**" forces 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, 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. 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 input editing characters accepted during input are: **CONTROL-A**, **CONTROL-H**, or **BS** to delete a character, and **CONTROL-W** or **BW** to delete a word.

Command Tree

The command tree structure for the Mesa debugger appears as an appendix at the end of this document. 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**.

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, and directing program control. It also contains 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 and trace commands apply to modules and procedures that are known within the current context. All breakpoints and tracepoints may be conditional; an optional command string can also be attached to each breakpoint/tracepoint which will be executed when the breakpoint/tracepoint is taken. A tracepoint is a breakpoint with a standard default command string. Tracepoints automatically invoke the normal Display Stack command processor (with subcommand p(arameters), v(ariables), or r(esults) as appropriate).

The three valid formats of a **condition** are: **variable relation variable**, **variable relation number**, and **number**. Conditions include relations in the set {**<**, **>**, **=**, **#**, **<=**, **>=**}. A **number** (multiple proceeds) means execute the break **number** times before invoking the debugger. The **variable**s are interpreted expressions that are looked up in the context of the breakpoint. A **variable** may not be an expression that is more than one word long, dereferences a pointer (beware of the implicit derefence in record qualification), or indexes an array. See `Attach Condition`, below, for more information.

You may set breakpoints 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. The debugger can set entry breakpoints on any procedure called from within a module. However, the fact that extra symbols are required to display the parameters or the breakpoint will not be discovered until they are needed. Breakpoints cannot be set on nested procedures (except with Wisk) unless the current context is the enclosing procedure. Note that breakpoints are set in all instances of a module. Breaks on a specific text location can be set only with the breakpoint commands of the *Source Ops* menu. When the source line of the breakpoint is displayed, the indicator <> appears to the left of the source where the breakpoint has actually been set (e.g., **IF foo THEN <> some statement;**). Before the debugger permits any breakpoints to be set using the source window, the creation date in the source file is checked against the corresponding date recorded by the compiler in the BCD. An incorrect version is reported with the message `Can't use` <module> `of` <time> `instead of version created` <time>. Since there is only one exit from a procedure, the debugger shows the beginning of the procedure for exit breaks instead of indicating a potentially incorrect RETURN statement. Local variables may be invisible if this RETURN has a PC that is not in the block with their declarations; use source breaks on the RETURN statements instead of an exit break.

When a break or trace is encountered during execution, a (possibly 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. The debugger types the name of the procedure containing the breakpoint, and the address and **PC** of the currently active frame. If the breakpoint has a condition associated with it, the break is taken only if the condition is satisfied. Note that the multiple proceed counter is reset after being satisfied; e.g. a condition of 5, will actually break on the fifth, tenth, fifteenth, ... times the breakpoint is reached. To continue execution of your Mesa program, execute the `Proceed` command; to stop execution of your program, execute the `Quit` command.

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 (e.g., you may break in the code of an **ELSE** clause when you really want the **THEN** clause if they share some common code). The message Cross jumped! will appear before the source of a cross jumped module is displayed. Entry and exit breakpoints are not affected by cross jumping.

The warning `Eval stack not empty!` will be printed if the debugger is entered via either an interrupt or breakpoint with variables still on the evaluation stack; this indicates that the current value of some variables may not be in main memory, where the interpreter normally looks. Exceptions to this are entry and exit breaks; the debugger has enough information to decode the argument records that are on the stack in this case (if the appropriate symbol tables are available).

`ATtach Condition [`**number**`, `**condition**`]`

changes a normal breakpoint into a conditional one. Arguments are a breakpoint number and a condition, which is evaluated in the context of the breakpoint. The breakpoint number is displayed when the break/tracepoint is set, and may also be obtained using the `LIst Breaks` command.

`ATtach Keystrokes [`**number**`, `**command**`]`

adds an arbitrary command string to breakpoints/tracepoints; the characters from this string are executed by the Debugger when the breakpoint/tracepoint is taken. Arguments are a breakpoint number and a command string terminated with a **CR**. A **CR** can be embedded in the command string by quoting it with **^V**.

`Break All Entries [`**module/frame**`]`

sets a break on the entry point to each procedure in **module** or **frame** (cf. `Break Entry`**).**

`Break All Xits [`**module/frame**`]`

sets a break on the exit point of each procedure in **module** or **frame** (cf. `Break Xit`**).**

`Break Entry [`**proc**`]`

inserts a breakpoint at the first instruction in the procedure **proc**.

`Break Xit [`**proc**`]`

inserts a breakpoint at the *last* instruction of the procedure body for **proc**. This catches all **RETURN** statements in the procedure.

`CLear All Breaks [confirm]`

removes all breakpoints/tracepoints.

`CLear All Entries [`**module/frame**`]`

removes all entry breakpoints/tracepoints in **module** or **frame**.

`CLear All Xits [`**module/frame**`]`

removes all exit breakpoints/tracepoints in **module** or **frame**.

`CLear All Traces [confirm]`

removes all breakpoints/tracepoints; it is equivalent to `CLear All Breaks`.

`CLear Break [`**number**`]`

removes a breakpoint by number. Typing **CR** in place of a number will clear the current breakpoint, i.e., the one that got you into the Debugger.

`CLear Condition [`**number**`]`

changes a conditional breakpoint into a normal one. Typing **CR** in place of a number behaves as in `CLear Break`.

`CLear Keystrokes [`**number**`]`

clears any command string associated with the breakpoint. Typing **CR** in place of a number behaves as in `CLear Break`.

`CLear Entry Break [`**proc**`]`

converse of `Break Entry`.

`CLear Entry Trace [`**proc**`]`

converse of `Trace Entry`; it is equivalent to `CLear Entry Break`.

`CLear Xit Break [`**proc**`]`

converse of `Break Xit`.

`CLear Xit Trace [`**proc**`]`

converse of `Trace Xit`; it is equivalent to `CLear Xit Break`.

`Display Break [`**number**`]`

displays a breakpoint by number. Its type (entry, exit, source), the procedure and/or module name in which it is found are displayed; for source breakpoints, the source text is also displayed; any attached conditions or keystrokes is also shown. Typing **CR** in place of a number behaves as in `CLear Break`.

`List Breaks [confirm]`

lists all breakpoints, displaying the same information as `Display Break`.

`Trace All Entries [`**module/frame**`]`

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

`Trace All Xits [`**module/frame**`]`

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

`Trace Entry [`**proc**`]`

sets a trace on the entry of the procedure **proc.** When an entry tracepoint is encountered, display stack mode is entered and the parameters are displayed. (cf. `Break Entry`)

`Trace Xit [`**proc**`]`

sets a trace on the exit of the procedure **proc.** When an exit tracepoint is encountered, display stack mode is entered and the return values are displayed. (cf. `Break Xit`)

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. Global frames are searched in the order: declarations, imports, directory. If the variable you wish to examine is not within the current context, use the commands that change contexts.

`AScii Read [`**address**, **n**`]`

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

`AScii Display [`**address**`, `**count**`]`

interprets **address** as **POINTER TO PACKED ARRAY OF CHARACTER** and displays **count** characters.

`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 stack subcommand mode is entered.

`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`). A `*` marks the current process. 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; space enters the interpreter; **--** delimits a comment; 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**. If **id** is simply an octal number, you are asked whether it is a condition variable (i.e., `Condition? [Y or N]`). 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.

`Display ReadyList`

displays all the processes waiting on the queue associated with the **ReadyList**, i.e., 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`

displays 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; **G** displays the global variables of the module containing the current frame; **P** displays the input parameters; **R** displays the return values (`(anon)` appears before those which are not *named* in the parameter lists); **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 a source window (It also sets the context for setting breakpoints in that window.); **L** just displays the source text; space enters the interpreter; **--** delimits a comment; 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 **G, 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 only the **GlobalFrames** of all the modules in the current configuration.

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 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/frame**`]`

> 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. Using a **frame** address has the same effect as `SEt Octal context.`.

`SEt Octal context [`**address**`]`

> changes the current context to the frame whose address is **address**. This is useful when there are several instances of the same module or in setting the current context to a specific local frame.

`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* if it is loaded; otherwise it returns you to the *Alto Executive.* `Quit`ting from the *Mesa Executive* returns you to the *Alto Executive*.

`STart [`**address**`] [Confirm]`

starts execution of the module whose frame is **address**. If the module has already been started, a **RESTART** will be done. Unlike the **START** statement in the Mesa language, 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 Loadstate [`**filename**`]`

> like `ATtach Image`, except that the initial rather than the current loadstate of the image file is used; this command is for wizards only.

`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**". *Only compiler output bcds for program modules can be attached; neither interfaces nor symbols files may be attached. This command overrides version checking.*

`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. The types of data segments are also printed; the (system-assigned) types currently recognized are heap, system, frame, table, bitmap, stream buffer, and Pup buffer. Unrecognized types (assigned by the user) are displayed as `data(t)`; the unknown type is displayed as `data(?)`. 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 most useful at octal breakpoints because the eval-stack is empty between most statements.

`LOgin [`**user**`, `**password**`]`

> sets the default user and password for the debugging session. The new user name and password are not written into the client's core image or onto the disk.

`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**.

`ReMote debuggee [`**host**`] [confirm]`

not implemented.

`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 `STart, Quit`, and calling procedures with the interpreter.

`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`, `Trace Stack, 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*.

`Trace Stack`

should be used if the Debugger breaks and enters the debugger nub ("`//`" mode); it dumps the Debugger's call stack in octal to the log. Change requests reporting Debugger problems that result in an uncaught signal or other problem should be accompanied by a debug log which includes the output of this command.

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, procedure calls, indexing, field access, addressing, displaying variables (also TYPEs), and simple type conversion. It is a powerful extension to the 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; these are valid only for debugging purposes and are not part of the Mesa language. The interpreter operates much like the compiler; strict target typing is performed on assignments and procedure calls.

Statement Syntax

Typing space (**SP**) to the command processor enables interpreting mode; the limited command processors of `Display Stack` and `Display Process` also permit a space. 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 "**%**" may be used instead of **LOOPHOLE[exp, type]**, with the expression on the left of the **%**, and the **type** on the right. However, **%** is not a valid **LeftSide**; all type expressions must be enclosed in parentheses.

The following expressions are equivalent to the interpreter:

foo % (short red Foo) and **LOOPHOLE[foo, short red Foo]**
(p % (LONG POINTER TO Object))^ and **LOOPHOLE[p, LONG POINTER TO Object]^**

The first pair will loophole the type of the variable **foo** to be a **short red Foo** and display its value. The second pair will loophole **p** to be a **LONG POINTER TO Object** and dereference it.

A number may be loopholed into **PROCEDURE**, **SIGNAL**, or an **ERROR**. If it is valid, the debugger will display the procedure (or signal's) name, module and global frame. If a signal/error is the same as the uncaught signal that trapped to the debugger, the debugger will also display the parameters.

Subscripting

There are two types of interval notation acceptable to the interpreter; the closed, open, and half open interval notation accepted by the compiler and a shorthand version that uses **!**.   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**).

The following expressions all display the octal contents of memory locations **4** through **7**:

```
MEMORY[4 . . 7]
MEMORY[4 . . 8)
MEMORY(3 . . 7]
MEMORY(3 . . 8)
MEMORY[4 ! 4]
```

Note that the interval notation is only valid for display purposes, and therefore is not allowed as a **LeftSide** or inside other expressions.

Explicit Qualification

To improve the performance of the interpreter, the **$** notation has been introduced to distinguish between qualification in the current context and explicit qualification. The character **$** indicates that the name on the left is a module name, 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**" may be used as shorthand to construct a **POINTER TO type**. This notation is used for constructing types in **LOOPHOLE**s (ie., **@foo** will give you the type **POINTER TO foo**). There is no shorthand to construct **LONG POINTER TO type**.

Radix conversion

The notation "**expression?**" will print the value of the expression in several formats, including octal and decimal. 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.

Arithmetic

Target typing is applied to arithmetic expressions. In complex expressions atoms which change the target type must occur first. For example:

```
(POINTER + offset)^      -- correct      (offset + POINTER)^      -- error message
LONG[400B] * 400B        -- 200000B      400B * LONG[400B]        -- overflow
```

Sample Expressions

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

If you were interested in seeing which procedure is 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.
```

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

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

The following command would display **rec** in octal:

```
Octal Read: @rec, n: SIZE[Rec].
```

Grammar

A copy of the Debug Interpreter grammar is in an appendix at the end of this document.

Section 6: Output Conventions

A "?" in any variable display uniformly means that the value is out of range. Elipses (". . .") indicate that there are additional fields present in a record which cannot be displayed due to lack of symbol table information. This can happen either in OVERLAID records or because a defs file is not present on the disk. In display stack mode, variables declared in nested blocks are now shown indented according to their nesting level.

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

displays all the elements of an array; e.g., `a = (3)[[x: 0, y:0], [x: 1, y: 1], [x: 3, y:3]]`. The parenthesized value to the right of the "=" is the length of the array.Typing **^DEL** will abort the display of long arrays.

ARRAY DESCRIPTOR

displays the descriptor followed by the contents of the array; e.g., `a = DESCRIPTOR[146013B^,3](3)[[x: 0, y:0], [x: 1, y: 1], [x: 3, y:3]]`. For a RELATIVE ARRAY DESCRIPTOR, the word RELATIVE is displayed first. Typing **^DEL** will abort the display of long array descriptors.

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.

CONDITION

displays a record containing an **UNSPECIFIED** and a `timeout`; a **CARDINAL**.

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`.

EXPORTED TYPES

displays the name of the type followed by an octal display of the contents if the length of the type is known. For example, an instance of the type **Handle: TYPE [2]** is displayed as `Handle (2)  1  1234B.`

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.

MDSZone

displays a **POINTER**.

MONITORLOCK

displays a record containing an **UNSPECIFIED**.

POINTER

displays an octal number, terminated with an "`^`", i.e., `p=107362B^`. **RELATIVE POINTER**s are decimal and are terminated with "^R", i.e., `r=123^R.`

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)`.

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.

REAL

displays a floating point number, e.g., `-1.45`.

RECORD

displays 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=[x: 9, y: -1]`.

SEQUENCE

displays as an array. For example, an instance **s** of the record **Sequence: RECORD [length: UnsignedInt, text: PACKED SEQUENCE maxLength: UnsignedInt OF CHARACTER]** is displayed as `s=[length: 3, text: (3)['a, 'b, 'c]]`.

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 **NIL**, `s=NIL` is displayed. Typing **^DEL** will abort the display of long strings.

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

```
ProcedureName, L: nnnnnB, PC: nnnB (in ModuleName, G: nnnnnB) --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: nnnnnB --global frame
```

A global context is displayed as the module name and its global frame. If the global frame is followed by `*`, i.e., `nnnnnB*`, it is a copy created by the **NEW** construct. If the global frame has not yet started, it will be followed by a ~.

In response to an expression followed by a **?**, the interpeter will show:

```
Octal = Hexadecimal = Unsigned Decimal = Signed Decimal =
Byte,,Byte = Octal Byte,,Octal Byte = CHAR,,CHAR =
Nibble:Nibble,,Nibble:Nibble
```

If any of the values are 0 or out of range, they will not be shown. For **LONG** values the interpreter will show:

```
Octal = Hexadecimal = Decimal = OctalWord OctalWord =
Byte,,Byte Byte,,Byte
```

For example, in response to `61141B?` the debugger displays

```
61141B = 6261X = 25185 = 98,,97 = 142B,,141B = 'b,,'a = 6:2,,6:1
```

and for `1234567B?` it shows

```
1234567B = 53977X = 342391 = 34567B 5 = 57,,119 0,,5
```

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

All of these errors will cause Wisk to flash the screen. Trying to set an entry or exit break near the same place may provide more information.

```
Can’t dereference or access array to test condition!
```

You have specified a condition that requires dereferencing or an array indexing to test; the runtime is unable to evaluate conditions that complex.

```
too many conditional breaks!
```

You have tried to set more conditional breaks than the system allows.

```
invalid relation!
```

You have specified an illegal relation expression for a condition.

```
user break block not found!
```

You have tried to free a conditional breakpoint when the conditional breakpoint information cannot be found (probably a core clobber).

```
variable is larger than a word!
```

You have tried to set a condition that uses a multiword value.

```
rhs on stack not allowed!
```

You have tried to set a condition where the right hand side of the relational expression is on the stack. Only the left hand side can be on the eval stack. This can only happen on entry and exit breakpoints

```
can’t break on port!
```

An attempt was made to set a breakpoint on an opcode on which it is not allowed; specifically in the middle of a port transfer.

```
no exchangable code found!
```

The debugger has tried several consecutive instuctions, and has not found an opcode on which a breakpoint is allowed. The code has probably been clobbered.

`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).

`no breaks have been set!`

You did a `LIst Breaks` when there weren't any.

`symboltable missing!`

The debugger is trying to manipulate a breakpoint for which there is no symboltable and it is not prepared to handle the situation.

`not allowed in INLINE!`

You have tried to set a breakpoint in an **INLINE** procedure.

`already set!`

You have already set a breakpoint on the location.

`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). This occurs most often in procedures that end with **ERROR** or a loop which does not terminate; a code clobber is also possible.

`conditions not checked in Worry mode!`

You have attached a condition while in worry mode. This is a warning only.

`??`

Unknown error.

Command execution

`... aborted`

Execution of the current command has been aborted (**^DEL** has been typed).

`!Resetting symbol table`

This warning is displayed before the debugger's scratch symbol table overflows; the command is retried automatically. The Debugger's performance decreases somewhat until the symbol table is refilled.

`!Number`

An invalid number has been typed.

`xxx not implemented!`

Feature `xxx` is not implemented.

```
!Invalid Address [nnnnB]
!Write protected [nnnnB]
!Non-existant memory page [nnnnB]
```

An illegal memory location has been referenced.

`! unknown file problem!  Your directory probably needs scavenging.`

Something is wrong with your directory.

`!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.

`!MDS exhausted [n]`

The debugger has run out of memory.

`Please terminate editing xxx.`

The file `xxx` is still being edited and you tried to leave the debugger.

`Disk full! Typescript reset to beginning.`

The debugger resets the typescript to the beginning of `Debug.log` if you run out of disk space. This is a warning message.

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.)

`Can’t use` <module> `of` <time> `instead of version created` <time>

This message is printed if the creation date in the source file on your disk is different than the corresponding date recorded by the compiler in the BCD.

`Cross jumped!`

The BCD was compiled with the cross jumping switch turned on. The source line displayed may not be what you expect.

`Pc not in any procedure!`

The debugger was unable to find a procedure or mainline code that matched the current pc. This is probably due to a clobber.

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. Extra banks available to the client must be unchanged. The debugger will run in only one bank after bootloading. You may not proceed after bootloading the debugger.

`*** 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 **SIGNAL**s 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*).

`Eval stack not empty!`

The warning is printed if the debugger is entered via either an interrupt or breakpoint with variables still on the evaluation stack; this indicates that the current value of some variables may not be in main memory, where the interpreter normally looks. Exceptions to this are entry and exit breaks; the debugger has enough information to decode the argument records that are on the stack in this case (if the appropriate symbol tables are available).

Interpreter

`! x is an invalid character`

The character `x` typed to the interpreter is illegal.

`! Syntax error at [n]`

There was a syntax error at location `n` in the expression given the interpreter.

`! Parse error at [n]`

There was a error at location `n` parsing the expression given the interpreter.

The following errors may have the offending identifier preceding the message:

`can’t call an INLINE!`

You tried to call an **INLINE PROCEDURE**.

`can’t lengthen!`

The interpreter needed to lengthen something while evaluating an expression that it couldn’t in order to make two types conformable.

`can’t make a constructor!`

Use field by field assignments. You gave the interpreter an expression using `[]` that looks like a constructor.

`double word array index!`

The index for an array must be a single word.

`has an invalid address!`

The expression to the right of the `@` is not word aligned.

`is an invalid number!`

This is probably a type mismatch.

`is an invalid pointer!`

This is probably a type mismatch.

`invalid subrange!`

This is probably a type mismatch.

`pointer fault!`

You tried to dereference **NIL**.

`is not a valid control link!`

The procedure or signal in your expression has an illegal value.

`is not a relative pointer!`

In the expression `base[rel]`, `rel` wasn't a **RELATIVE POINTER**.

`is not a type!`

The identifier used in a type expression was not a type.

`is not a unique field selector!`

The field selector occurs more than once in the computed or overlaid variant.

`is not a valid field selector!`

The identifier given for a field selector is not in the record. This could be because you lack the symbols for the record declaration on your disk.

`overflow!`

Overflow occured while doing arithmetic. Perhaps you need a **LONG** in the expression.

`relations not implemented!`

`a = b` is not allowed.

`size mismatch!`

You tried to assign or loophole two things of different sizes. Loopholing pointers is a useful trick for records of different sizes.

`too many arguments for stack!`

You can only call procedures that take 5 or fewer words of arguments.

`has incorrect type!`

Type mismatch.

`unknown variant!`

The interpreter found a garbage tag field.

`Won’t dump that much memory!`

You tried to print more than 64K with the **MEMORY** construct.

`not permitted in worry mode!`

You can’t call procedures in worry mode.

`is the wrong base!`

In the expression `base[rel]`, the type of `base` is not what `rel` expects.

`has the wrong number of arguments!`

The arguments to a procedure call are wrong.

`used incorrectly with []!`

You probably tried to use `[]` as a type constuctor.

`xxx$ is ambiguous; use frame $!`

There is either more than one instance of `xxx` instantiated, or the code for `xxx` is packed with another module.

Symbol Lookup

`xyz not found!`

The variable or file named `xyz` cannot be found.

`!File: xyz`

The file named `xyz` cannot be found.

`nnnnnB not started!`

The global frame `nnnnnB` has not yet been started. Any variables looked up will be uninitialized.

`xyz not bound!`

The imported variable `xyz` is not exported by anyone.

```
!File: --compressed symbols--
```

The symbol file is compressed.

```
--- has incorrect version!
```

The symbol file has an incorrect version stamp.

```
!Tree for xx not in symbol table
```

A multiword constant in your code wasn't copied into the symbol table. Look in the source file to find the value.

```
Use Interface.importedVariable, not Interface$importedVariable
```

The debugger cannot find imported variables from an interface file (the "$" notation). The "." notation will tell it to use the interface record (if found) available in the current context.

Validity checking

```
--- is not a valid frame!
--- is a clobbered frame!
--- not a 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 6.0

- **AS**cii
  - **R**ead [**address**, **count**]
  - **D**isplay [**address**, **count**]
- **AT**tach
  - **I**mage [**filename**]
  - **C**ondition [**number**, **condition**]
  - **K**eystrokes [**number**, **command**]
  - **L**oadstate [**filename**]
  - **S**ymbols [**globalframe**, **filename**]
- **B**reak
  - **A**ll
    - **E**ntries [**module/frame**]
    - **X**its [**module/frame**]
  - **E**ntry [**procedure**]
  - **X**it [**procedure**]
- **CL**ear
  - **A**ll
    - **B**reaks [confirm]
    - **E**ntries [**module/frame**]
    - **T**races [confirm]
    - **X**its [**module/frame**]
  - **B**reak [**number**]
  - **C**ondition [**number**]
  - **E**ntry
    - **B**reak [**procedure**]
    - **T**race [**procedure**]
  - **K**eystrokes [**number**]
  - **X**it
  - **B**reak [**procedure**]
  - **T**race [**procedure**]
- **CO**remap [confirm]
- **CU**rrent context
- **D**isplay
  - **B**reak [**number**]
  - **C**onfiguration
  - **E**val-stack
  - **F**rame [**address**] (**g,j,l,n,p,q,r,s,v**)
  - **G**lobalFrameTable
  - **M**odule [**module**]
- **D**isplay
  - **P**rocess [**process**] (**l,n,p,q,r,s**)
  - **Q**ueue [**identifier**] (**l,n,p,q,r,s**)
  - **R**eadyList (**l,n,p,q,r,s**)
  - **S**tack (**g,j,l,n,p,q,r,s,v**)
- **F**ind variable [**identifier**]
- **K**ill session [confirm]
- **LI**st
  - **B**reaks [confirm]
  - **C**onfigurations [confirm]
  - **P**rocesses [confirm]
- **LO**gon [**user**, **password**]
- **O**ctal
  - **C**lear break [**globalframe**, **bytepc**]
  - **R**ead [**address**, **number**]
  - **S**et break [**globalframe**, **bytepc**]
  - **W**rite [**address**, **value**]
- **P**roceed [confirm]
- **Q**uit [confirm]
- **R**e**S**et context [confirm]
- **R**e**M**ote debuggee [**host**] [confirm]
- **SE**t
  - **C**onfiguration [**config**]
  - **M**odule context [**module/frame**]
  - **O**ctal context [**address**]
  - **P**rocess context [**process**]
  - **R**oot configuration [**config**]
- **ST**art [**address**] [confirm]
- **T**race
  - **A**ll
    - **E**ntries [**module/frame**]
    - **X**its [**module/frame**]
  - **E**ntry [**procedure**]
  - **S**tack
  - **X**it [**procedure**]
- **U**serscreen [confirm]
- **W**orry
  - off [confirm]
  - on [confirm]
- **^D**ebug [confirm]

Debugger Interpreter Grammar

Version 6.0

```
StatementList     ::=  Statement | StatementList;  | StatementList; Statement
Statement         ::=  LeftSide Interval | LeftSide _ Expression |
                        MEMORY Interval | Expression | Expression ?
LeftSide          ::=  identifier | ( Expression ) |  LeftSide Qualifier |
                        identifier $ identifier | number $ identifier |
                        MEMORY [ Expression ] | LOOPHOLE [ Expression ] |
                        LOOPHOLE [ Expression , TypeExpression ]
Qualifier         ::=  ^ | . identifier | [ ExpressionList ]
Interval          ::=  [ Bounds ]  | [ Bounds )  | ( Bounds ]  | ( Bounds )  |
                        [ Expression ! Expression ]
Bounds            ::=  Expression .. Expression
Expression        ::=  Sum
Sum               ::=  Product | Sum AddOp Product
AddOp             ::=  + |
Product           ::=  Factor | Product MultOp Factor
MultOp            ::=  * | / | MOD
Factor            ::=    Primary | Primary
Primary           ::=  Literal | LeftSide | @ LeftSide | BuiltinCall |
                        Primary % | Primary % ( TypeExpression )
Literal           ::=  number | character | string
BuiltinCall       ::=  NIL | NIL [ TypeExpression ] | PrefixOp [ ExpressionList ] |
                        TypeOp [ TypeExpression ]
PrefixOp          ::=  ABS | BASE | LENGTH | LONG | MAX | MIN
ExpressionList    ::=  empty | Expression | ExpressionList,  Expression
TypeOp            ::=  SIZE
TypeExpression    ::=  identifier | TypeIdentifier | TypeConstructor
TypeIdentifier    ::=  BOOLEAN | INTEGER | CARDINAL | WORD | REAL | CHARACTER |
                        STRING | UNSPECIFIED | PROC | PROCEDURE | SIGNAL | ERROR |
                        identifier identifier | identifier TypeIdentifier |
                        identifier . identifier | identifier $ identifier
TypeConstructor   ::=  LONG TypeExpression | @ TypeExpression |
                        POINTER TO TypeExpression
```

Wisk Summary

Version 6.0

WHAT WISK MOUSE BUTTONS DO:

	Scroll Bar	Text Area
RED	Scroll Up	Select
YELLOW	Thumb	Menu
BLUE	Scroll Down	Extend

NAME STRIPE/SMALL WINDOW 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	Deactivate

STANDARD TEXT OPS MENU COMMANDS:

Find [selection]	Normalize Insertion	Split
Position [selection]	Normalize Selection	Wrap

SOURCE WINDOW SOURCE OPS MENU COMMANDS:

Create	Set Break [selection]	Clear Break [selection]
Destroy	Set Trace [selection]	Attach

SOURCE WINDOW FILE OPS MENU COMMANDS:

Load [selection]	Store [selection]	Reset
Edit	Save	