Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Bruce Malasky, John Wick	Location	Palo Alto
Subject	**Mesa 6.0 Debugger Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>Debugger60.bravo (and .press)

This memo outlines changes made in the Mesa Debugger since the last release (Mesa 5.0, April 9, 1979); it is intended as a concise guide to conversion, not a detailed specification of the changes. Complete documentation on the Mesa 6.0 Debugger can be found in the *Mesa Debugger Documentation*.

User Interface

The Debugger's user interface incorporates changes made in Tajo (the Tools Environment); the window package **Wisk** has been converted to use **Vista**, the new window package. For more complete documentation on the Tajo design, see the *Tajo User's Guide* and the *Tajo Functional Specification*.

Typein

The assignment of some function keys and mouse buttons has changed. The menu button is now YELLOW (formerly BLUE). FL4 is no longer the stuff key; use FR4 (Spare2), Keyset2, or ^S. 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

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).

Selections

The selection scheme has changed. 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 current selection is now video reversed.

Scrollbars

Scrollbars no longer occupy a dedicated part of the window, but instead come up on top of the left edge. They are twice as wide as before, and 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.

Name Stripe

The name stripe and tiny windows now video reverse when the cursor is in the sections that function as accelerators for the window manager menu commands (`Move`, `Grow`, `Size`, `Top`, `Bottom`, and `Zoom`).

Menus

Except for the change from BLUE to YELLOW, the way menus are invoked has not changed. However, some new menus and commands have been added.

Standard Menus

In addition to `Move`, `Grow`, `Size`, `Top`, `Bottom`, and `Zoom`, the standard window manager menu now also includes the following command:

> `Deactivate`
>
> This command deactivates the selected window; it will no longer appear on the screen and the resources used by it will be freed. The window's name is added to a menu of deactivated windows, which is available outside all windows. The window may be made active again by selecting its menu item.

A new *Text Ops* menu is now supplied with the `Debug.log` and source windows in addition to the Window Manager menu. It contains `Find`, `Position`, `Split`, `Normalize Insertion`, `Normalize Selection`, and `Wrap`; the following commands are new:

> `Split`
>
> The Debugger's wisk window has been replaced by the more general `Split` window command. Feedback is similar to that in Laurel: the split line can be picked up using RED and moved vertically. The subwindow is destroyed by moving the split line off the top or bottom of the (sub)window.
>
> `Normalize Insertion`
>
> For windows containing an insert point (`Debug.log` and editable source windows), this command will position the text in the (sub)window so that the line containing the insert point is at the top.

`Normalize Selection`

This command positions the text in the (sub)window so that the line containing the left most position of the current selection is at the top.

Debugger Menu

A separate *Debugger* menu no longer exists; the `Alter Bitmap` function has been deleted, `Move Boundary` has been superseded by `Split` (see above), and `Stuff It` is now available only on the keyboard.

Source Menus

In addition to the standard menus, the source window has two additional menus, *Source Ops* and *File Ops*. The *Source Ops* menu contains the following commands, which are unchanged: `Create`, `Destroy`, `SetBreak`, `SetTrace`, and `ClearBreak`; the last three commands are available only if a file has been loaded into the window. The *Source Ops* menu contains the following new command:

`Attach`

Causes the Debugger to ignore the creation date of the current source file when setting breakpoints or positioning to a source line. This command is essentially a LOOPHOLE; *because the source-object correspondence may not be correct, it should be used with caution*. If, after using `Attach`, the Debugger sets breakpoints in strange places, chances are that the source file does not match the version of the object in the system you are debugging.

The *File Ops* menu includes the following new commands (plus `Load`, which functions as before):

`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 `Save`d or `Store`d on a file), the amount of information in the window should be kept small.

`Save`

Outputs the contents of the window to its current file; overwriting the file requires confirmation. A backup "$" file is created that is a copy of the unedited version. After the `Save` command completes, access reverts to read only.

`Store`

Outputs the contents of the window to the file named by the current selection; if the file already exists, overwriting it requires confirmation. After the `Store` command completes, access reverts to read only.

`Reset`

Discards all edits that have been made to the window (during this session) and resets access to read only. If the file is not editable, the window is made empty.

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

Editing

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

^RED	Moves the insert point (represented by a blinking caret) 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.

The message `Please terminate editing of` <filename> appears in the `Debug.log` if you try to `Kill` or `Quit` from the Debugger while editing a file.

Caution: The editing facilities are designed not to alter the original file until it is `Saved` or `Stored`, much like Bravo; the original contents are copied to a file with "$" appended to its name. This is however, a new facility and should be used with caution. It is designed to support a moderate number of localized changes to programs, not to replace your favorite document creation system.

Debugger Commands

Changes in Debugger commands are relatively minor. The Debugger's interpreter is more generally available and more consistent with the language. Tracepoints have been re-implemented as a minor extension of the standard breakpoint facilities.

Old Commands

`Ascii/Octal Read`

The `Ascii` and `Octal Read` commands no longer automatically increment the default value produced by ESC.

`Break/Trace Points`

Break and trace points can no longer be set by typing a source line, and the `Break Module` and `Break Procedure` commands and corresponding `Trace` and `Clear` commands have been deleted; the menu commands must be used.

The distinction between trace and breakpoints has been removed. An optional command string can now be attached to each breakpoint which will be executed when the breakpoint is taken. A tracepoint then becomes a breakpoint with a standard default command string. `LIst Breaks` lists both break and tracepoints (`List Traces` has been deleted). `Clear All Entries/Xits` clears both break and tracepoints.

Tracepoints automatically invoke the normal `Display Stack` command processor (with subcommand `p(arameters), v(ariables),` or `r(esults)` as appropriate). The `q(uit)` subcommand (not `b(reak)`) exits to the Debugger's command level, where the normal `Proceed` command continues execution of the client.

The method of specifying conditional break and tracepoints has changed; see the `ATtach Condition` command in the next section.

When an exit break is set, the Debugger breaks on any return of the procedure by setting the actual breakpoint on a common return instruction. The Debugger has no way of telling which return was taken if there is more than one. When asked to display the source line when at an exit break, the Debugger now shows the declaration line of the procedure instead of the last return statement.

`Case On/Off`

The Debugger no longer ignores case, and the case commands have been deleted; *identifiers must be typed with their correct capitalization.*

`Control DEL`

Typing **^DEL** will now abort the display of long arrays and strings, as well as most searches. This key combination no longer has to be held down to be recognized.

`COremap`

This command now prints more information about some data segments; 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)`; an unknown type is displayed as `data(?)`.

`Display Process [`**process**`]`

The subcommand space (**SP**) can now be used to invoke the interpreter.

`Display Stack`

The new subcommand "`g`" displays the global variables of the module containing the current procedure. A space (**SP**) invokes the interpreter. If the source window is loaded with the `s(ource)` subcommand, the window will remember the appropriate context for setting breakpoints.

`Interpret Call`

The `Interpret Call` command has been deleted; the Debugger's interpreter should be used. There are no longer any restrictions on when the interpreter may be called.

`ReSet Context [confirm]`

This command now requires two keystrokes, to avoid conflict with the `ReMote debuggee` command (not yet implemented on the Alto).

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

This command now requires confirmation.

New Commands

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

Interprets **address** as POINTER TO PACKED ARRAY OF CHARACTER and displays **count** characters (each character separately, not as a string).

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

This command replaces old style conditional breaks; it 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**`]`

Arbitrary command strings can now be attached to break and tracepoints; they are executed by the Debugger when the breakpoint 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**.

`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.

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

This new command is the same as `Trace All Entries/Xits`, except that breakpoints are set.

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

This command clears breakpoints by number. Typing **CR** in place of a number will clear the current breakpoint, i.e., the one that transferred control into the Debugger.

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

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

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

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

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

This command sets the default user name 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.

`ReMote Debugee [`**host**`]`

This command is not implemented on the Alto.

`Trace Stack`

This command is used when 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.

Interpreter

The interpreter provides support for all of the new language features introduced in Mesa 6. All commands requiring numeric input now invoke the interpreter automatically (e.g., `Octal Read: @p, n: SIZE[r]`).

Grammar

A summary of the revised grammar is attached. The constructs ABS, ERROR, LONG, LOOPHOLE, MAX, MIN, NIL, POINTER TO, PROC, PROCEDURE, SIGNAL, WORD, and open and half open intervals have been added to the interpreter's grammar; type REAL has been added for output only. Type expressions following **%** must be enclosed in parentheses. The interpreter syntax **Expression?** has replaced the `Interpret Expression` command; it prints the value of the expression in several formats including octal and decimal.

Target Typing

The interpreter now does a much better job of target typing. As a result, arguments to procedure calls and right hand sides of assignments are type checked. In addition, assignments to enumerated types now work correctly.

The interpreter also does a better job of determining signed/unsigned representation. For example, any octal number is assumed to be unsigned.

Symbol Lookup

Even if a module has compressed symbols, the debugger will first look for the file `modulename.bcd` to see if it is the original compiler output for that module (by checking the version stamp). If so, it will use those symbols. Thus, there is no need to `Attach Symbols` if the proper file is on the disk. It makes sense to use compressed symbols for large systems and to also have present the complete symbol files for the specific modules undergoing detailed debugging.

Output Conventions

In display stack mode, variables declared in nested blocks are now shown indented according to their nesting level.

A "`?`" in a variable display now uniformly means that the value is out of range; "`. . .`" indicates that there are additional fields present which cannot be displayed due to lack of symbol table information.

When the debugger refers to a program module, it usually gives the address of its global frame, e.g., "`G: nnnnnB`". If the module has not been started, the debugger now prints a "~" after the `B`. If a module has not been started, the user *should not modify* the global variables of that module, nor should they be displayed, as they are uninitialized.

New Error Messages

The warning `Eval stack not empty!` will be printed if the debugger is entered via either an interrupt or a 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 at 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).

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. The message `Can’t use` <module> `of` <time> `instead of version created` <time> will result if the versions do not match (but see the `Attach` source menu command above).

The message `Resetting symbol table!` is displayed when the interpreter’s scratch symbol table overflows; the command is retried automatically. The Debugger’s performance decreases somewhat until the symbol table is reinitialized.

If a program is compiled with cross-jumping, the debugger will print the warning `Cross jumped!` before displaying the source.

Installation

Fonts

The Debugger now requires a strike font named `MesaFont.strike` or `SysFont.strike`; a version of Gacha10 is available on `<Mesa>MesaFont.strike`. Additional strike fonts are stored on `[Maxc]<AltoFonts>`. (Strike fonts which include kerning are not supported.)

Switches

Installing the debugger with the `/b` switch will video reverse the display (i.e., white characters on a black background).

Memory Bank Management

When running on machines with more than 64K of memory, the client system supplies space to the Debugger for its bitmap (unless all but one bank has been disabled; see below); the client can disable this option by using the `/k` switch or by calling a system procedure before the Debugger is first invoked (see the *Mesa 6.0 System Update*).

It is also possible for the Debugger to be installed with more than one bank of memory available for code swapping; this is done by reducing the amount of memory available to the client using the RunMesa bank switches or the Alto Executive `MesaBanks.~` command (in Executive version 11 or later).

> `MesaBanks.~`
>
> This command establishes the default memory allocation available to client programs. Arguments can be in two forms: a sixteen bit octal mask (followed by an optional `/b` switch) indicating the *available* banks; a one in bit position *n* of the mask (counting from the left) indicates that bank *n* is available. Form two is a series of decimal bank numbers each followed by the `/x` switch; each bank mentioned is *excluded* from use by the client. Note that a request to exclude bank zero will be ignored. If no argument is present, the command will display the current value of the bank mask.
>
> The `MesaBanks.~` command establishes the available memory for each `.image` or `.bcd` program invoked directly by the Alto Executive. The default may be overridden by explicitly using RunMesa to invoke the program and optionally specifying bank switches on its command line, before the `.image` file name. The bank switches have the same format as the arguments to `MesaBanks.~` (except that the `/b` switch is required in the case of a

bitmask). In the absense of any bank switches, RunMesa always assumes that all banks are available to the client.

Using these facilities, it is possible to set up the defaults so that the Debugger has extra banks of memory at the expense of the client program. For example, on a three bank Alto, the following commands might be used to set the default and then install the Debugger:

```
MesaBanks.~ 2/x
RunMesa.run 1/x XDebug.image
```

Under this arrangement, the client would use banks zero and one, and the Debugger would use banks zero and two (because bank zero is swapped onto `Swatee`, it can be used by both). Actually, because the client (by default) is also allocating space for the Debugger's display bitmap, the client actually has only one-and-one-half banks, and the Debugger has two-and-one-half; this can be changed by running the client with the `/k` switch, resulting in two banks available to each.

Note that the `MesaBanks.~` command affects *all* Mesa programs invoked by the Alto Executive, including the Compiler and Binder. So the above example would run the Compiler in only two banks, not three; this can be changed by saying `RunMesa Compiler` on the command line, which, because there are no bank switches specified, defaults to all banks available (not really necessary in this case, since the Compiler runs almost as well in two banks as in three). On the next new session, the Debugger is smart enough to notice that the Compiler (or whoever) has smashed what it thought was in bank two. (It is also smart enough not to use any memory that the client owns, so that the `1/x` switch on the command line above is actually unnecessary.)

Since there are a lot of options here, some "standard" examples of client and Debugger configurations might be helpful:

Two Banks: Normally, do nothing; client and Debugger will each have one-and-one-half banks. For small clients and better Debugger performance, use `RunMesa 1/x Mesa.image Client.bcd`, which will give the client one bank and the Debugger two. (If you were to use `MesaBanks.~ 1/x` in this case, the Compiler would also be restricted to one bank).

Three Banks: As in the three bank example above.

Four Banks: Use `MesaBanks.~ 3/x` to give the client and the Debugger two-and-one-half banks each and the Compiler three; use `MesaBanks.~ 2/x 3/x` and `Client/k` to increase the Debugger's allocation to three banks and restrict the client to two. Obviously, this can be adjusted based on the size of the client and the desired performance of the Debugger.

Extended Features

Nearly all of **Alto/Tajo** is now included in the Debugger (Librarian support and communications are not). Accordingly, there is little (if any) distinction between UserProcs and Tools, and **Fetch** (the **FileTool** plus communications) which runs in the Debugger is the same as the **FileTool** provided by **Alto/Tajo**. A copy of section 10 of the *Tajo User's Guide* describing the **FileTool** is attached to this memo.

Distribution:
Mesa Users
Mesa Group
SDSupport

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	

R	Y	B	R	Y	B	R	Y	B
T/B	┌G	┌M	Zoom	T	Size	T/B	┌G	┌M

X

10.0 File Tool

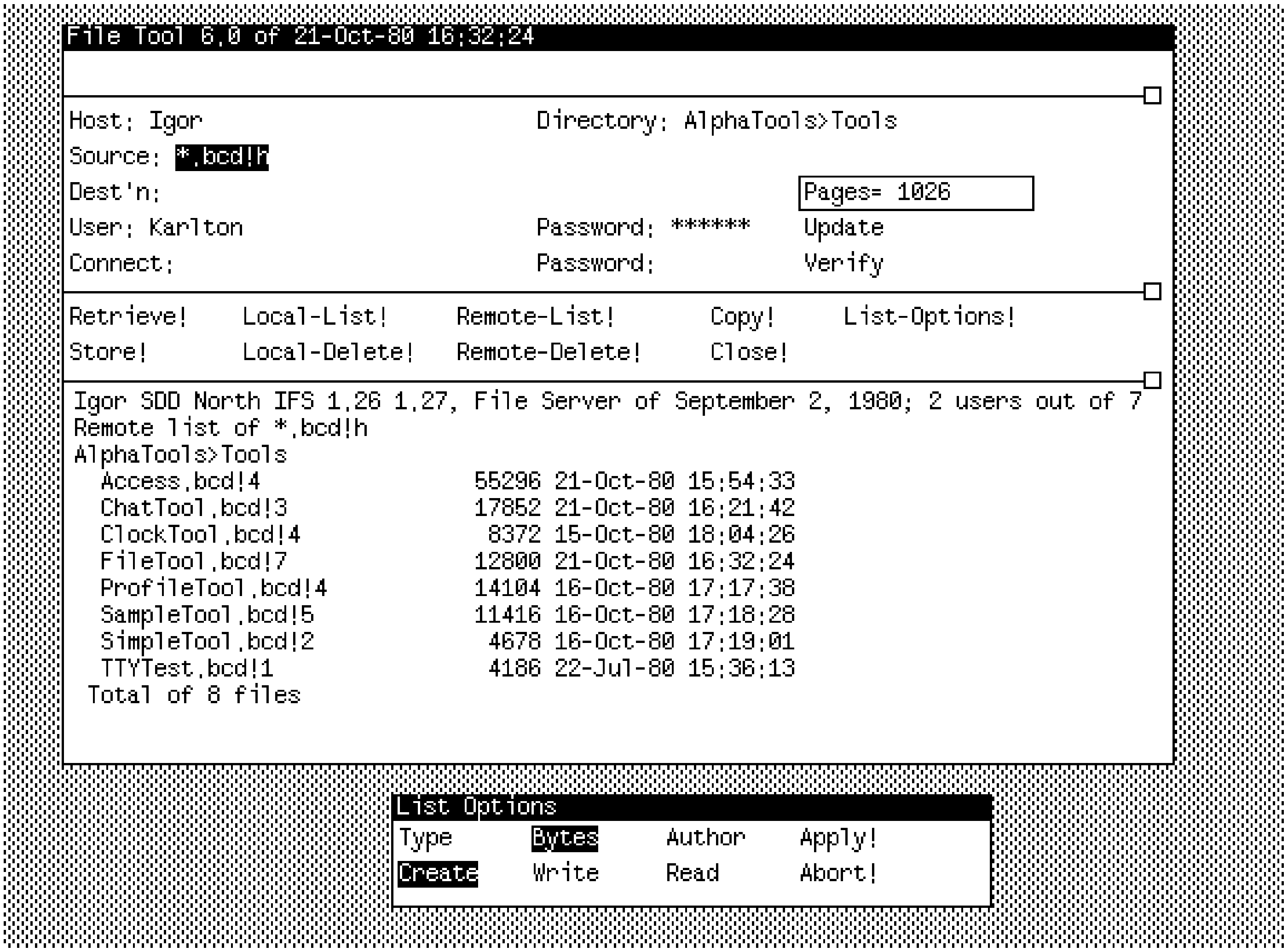
The File Tool provides a means of dealing with local as well as remote file systems from within the Development Environment.

10.1 The User Illusion

The File Tool employs the standard features of the Development Environment. See section 3 for further details.

10.2 Tool Appearance

Below is an illustration of a File Tool with the List Options window (explained below) visible.



```
File Tool 6.0 of 21-Oct-80 16:32:24

Host: Igor                          Directory: AlphaTools>Tools
Source: *.bcd!h
Dest'n:                                                Pages= 1026
User: Karlton                       Password: ******   Update
Connect:                            Password:          Verify

Retrieve!   Local-List!    Remote-List!     Copy!    List-Options!
Store!      Local-Delete!  Remote-Delete!   Close!

Igor SDD North IFS 1.26 1.27, File Server of September 2, 1980; 2 users out of 7
Remote list of *.bcd!h
AlphaTools>Tools
  Access.bcd!4             55296 21-Oct-80 15:54:33
  ChatTool.bcd!3           17852 21-Oct-80 16:21:42
  ClockTool.bcd!4           8372 15-Oct-80 18:04:26
  FileTool.bcd!7           12800 21-Oct-80 16:32:24
  ProfileTool.bcd!4        14104 16-Oct-80 17:17:38
  SampleTool.bcd!5         11416 16-Oct-80 17:18:28
  SimpleTool.bcd!2          4678 16-Oct-80 17:19:01
  TTYTest.bcd!1             4186 22-Jul-80 15:36:13
 Total of 8 files

List Options
Type     Bytes    Author   Apply!
Create   Write    Read     Abort!
```

10.3 Parameter Subwindow

The upper form subwindow contains parameters that can be set by the user; they will be used by the next File Tool command.

Host: the name of the host to be used for remote file operations. If a connection is already open, any editing of this field causes it to be closed; if a transfer is in progress, the connection will not be closed until it is complete.

Directory: the default remote directory. If empty, the value in the User: field is used.

Source: a list of files (separated by spaces or returns) to be operated on. If the first character of a file name is "@", then the file is taken to be an indirect file and its contents are used as a list of files. Indirect files may nest.

Dest'n: file name for the destination of a transfer. If this field is left blank, the file name is the same as the source.

Pages= number of free pages left on the disk. This item is read only.

User:, Password: the primary directory and the associated password. This field is initialized from the value of the user's last Alto Operating System login. Editing of this field is local to the File Tool and does *not* affect the user's login in the Alto Operating System.

Update only store or retrieve the file if the source is newer than the destination (comparing creation dates). The default is false.

Connect:, Password: the secondary directory and the associated password.

Verify request confirmation for each file operation. The default is false.

10.4 Command Subwindow

File Tool commands are available in the second form subwindow. Some of the commands are accomplished by a background process. Those commands clear the Command subwindow so that a second operation cannot be invoked while one is under way. The Copy! command operates only on the local disk. It does not take a list of files to operate upon. Close! closes a remote connection (if there is one).

It is important to remember that the commands are postfix; e.g., fill in the Host: and Source: fields before invoking the Retrieve! command. The following commands are available:

Retrieve! transfers the file specified in Source: from the remote file system to the local disk. The file name must conform to the file-naming conventions on the remote host. You may designate multiple files by the use of * expansion only to the extent that the remote server supports it. If the local file is already in use, the transfer will not be made and the message "<filename>: can't be modified" will be displayed in both the message window and the log window. See warning in Section 10.6

Local-List! lists all files on the local disk corresponding to the name in Source:. This command will expand *s and #s.

Remote-List! lists all files on the remote file system corresponding to the name in `Source:`. This must conform to the file naming conventions of the remote host. You may designate multiple files by the use of * expansion only to the extent that the remote server supports it (currently Maxc and IFS do, but differently).

Copy! makes a copy of a local file on the local disk. Only a single file may be copied and *s and #s are not allowed.

List-Options! creates a List Options window if one does not already exist.

Store! transfers the file specified in Source: from the local disk to the remote `Host`. Alto file name conventions apply to the local file.

Local-Delete! deletes the files specified in Source: from the local disk. If the local file is already in use, the delete will be skipped and the message "<filename>: can’t be modified" will be displayed in both the message window and the log window. See warning in Section 10.6

Remote-Delete! deletes the file specified in Source: from the remote file system. You may designate multiple files by the use of * expansion only to the extent that the remote server supports it.

Close! closes the currently open FTP connection.

If Verify is TRUE, then for each file that might be acted upon, the following commands are displayed

Confirm! do the operation.

Deny! abort the operation.

Stop! abort the operation and terminate the command. This will close the connection with the server if a retrieve is being aborted.

10.5 List Options window

The List Options window is created by the List-Options! command. The properties that will be displayed, in addition to the file name, by a Local-List! or Remote-List! are governed by the booleans in this window. After changing the options, use Apply! to effect those changes. The Abort! command will restore the options which existed before the List-Options! command was selected. Choosing either of the commands in the List Options window will cause that window to be removed.

If the Type attribute is requested for a Local-List! and the type is unknown, it will be listed as such to prevent the time it would take to read the file and determine the type.

10.6 Exceptions

The actual transfer takes place in a background process, so the user is free to issue other commands or even change the values in the parameter subwindow without affecting the command currently executing. Changing a parameter while the File Tool is waiting for Confirm! will **not** affect the name of the destination file; you should skip the transfer (by using Deny!) and reissue the command with the desired parameter correctly set.

Warning: Do not attempt to use a file while it is being retrieved. This includes issuing commands to the Debugger that cause it to try to reference the file. For example, Display Stack may cause the Debugger to reference symbols contained in the file being retrieved.

Warning: If you are using the File Tool in the Debugger, be careful not to change any files out from under the program you are debugging; the file tool makes no provisions for checking if the file is in use *in the client world* when you modify a local file.

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Bruce Malasky	Location	Palo Alto
Subject	Debugger: Extended Features	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>XDF.bravo (and .press)

DRAFT

This memo discusses *Debugger User Procedures* (UserProcs) and contains a sample *Printer*, a special type of UserProc.

The Debugger is now the functional equivalent of the Alto/Tajo environment (with the exception of Librarian support and communications). As a result, there are no longer any differences between the **FileTool** and **ChatTool** that run in Alto/Tajo and the versions that run in the Alto/Mesa Debugger.

Loading User Procedures

To install the Debugger from the command line with some UserProcs, type:

```
XDebug YourProc1[/l] YourProc2[/l] ...
```

to the Alto Executive. To load files in an installed debugger, simply enter the Debugger nub; then do a `>New` **filename**, followed by `>Start` **globalframe**. More information on the mechanism for loading programs into the Debugger can be found in the *Mesa User's Handbook* and the *Mesa Debugger Documentation*.

Hints for Writing User Procedures

The Debugger gives you added help in gaining access to the information it already knows about your program. The Debugger's configuration exports all of the Debugger's and Tajo's interfaces; see `XDebug.config` for details. A user program can access any of the Debugger's public procedures simply by importing the definitions modules of the procedures that you want to use. When writing your own debugging routines, look carefully at some of the utility routines that the Debugger already provides (e.g., **Name, Frame**, **ShortREAD**, etc.). In particular, **DebugUsefulDefs** contains most of the interesting procedures you might want. The interface **DOutput** contains utility procedures for displaying information in the `Debug.log` (a la **IODefs**). You should also look at the `<MesaLib>` and `<AlphaHacks>` directories for UserProcs that other Mesa users have already written and debugged.

Warning: *The Mesa Group makes no guarantees about the stability of these interfaces between releases. Use at your own risk!*

Printers

The Debugger is capable of calling a user supplied procedure to print variables of specific types. To do this, a program must first register any type it will display by calling

```
AddPrinter: PROC [type: STRING, proc: PROC [DebugOps.Foo]]
```

from the interface **Dump**. The Debugger's interpreter evaluates **type** at the beginning of each session and remembers the target type of the result. Unfortunately, **type** is not a simple type expression, but rather a statement evaluated by the interpreter; the type is extracted from the result. Any additional information such as the address of a variable used when evaluating the statement is ignored.

Later, whenever the Debugger encounters a variable of that type, it will call **proc** to display it. If, for a given printer, calling **proc** or evaluating **type** ever causes an UNWIND, the printer is never called again. The parameter to **proc** is defined as follows:

```
Foo: TYPE = POINTER TO Fob;

Fob: TYPE = RECORD [
  there: BOOLEAN,
  addr: BitAddress,
  words: CARDINAL,
  bits: [0..WordLength),
  . . .]];

BitAddress: TYPE = RECORD [
  base: LONG POINTER,
  offset: [0..WordLength],
  . . .];
```

If **there** is TRUE, the **BitAddress** is a location in the user core image. For large structures, **LongREAD** and **LongCopyREAD** from **DebugUsefulDefs** should be used to access the data; for small structures the procedure **GetValue** in the interface **DI** (it takes a **Foo** as its argument) copies the information into the Debugger's core image and updates the **addr**. The Debugger owns the storage for **Foo**s and the values copied into them from the user's core image; they are freed by the Debugger between commands.

A good technique for debugging the string used in the call to **AddPrinter** is to actually try it out using the interpreter. All REALs could be intercepted by supplying the following STRING to **AddPrinter**:

```
0%(REAL)
```

The following STRING is used by the sample printer attached at the end of this memo.

```
LOOPHOLE[1400B,   StackFormat$Stack]^
```

The constant 1400B is simply a location that is always mapped; **AddPrinter**'s evaluation of the STRING does not actually use that location.

Once **StackPrinter** is instantiated in the Debugger, **PrintStack** is called whenever the Debugger wants to display a **StackObject**. Since **PrintStack** understands the format of **StackObject**s, it can show the complete contents of a **stack**, something the Debugger is unable to do because of the zero length array.

```
-- StackFormat.mesa
-- Last Edited:  Keith, October 21, 1980  10:30 PM

StackFormat: DEFINITIONS =
  BEGIN

  Stack: TYPE = POINTER TO StackObject;

  StackObject: TYPE = RECORD [
    top: CARDINAL _ 0,
    max: CARDINAL _ 0,
    overflowed: BOOLEAN _ FALSE,
    stack: ARRAY [0..0) OF CARDINAL];

  END.


-- StackPrinter.mesa
-- Last Edited:  Keith, October 21, 1980  10:38 PM

DIRECTORY
  DebugOps USING [Foo, LongREAD],
  DI USING [GetValue],
  DOutput USING [Char, Line, Octal, Text],
  Dump USING [AddPrinter],
  StackFormat USING [StackEntry, StackObject];

StackPrinter: PROGRAM IMPORTS DebugOps, DI, DOutput, Dump =
  BEGIN

  PrintRecord: PROC [lp, lps: LONG POINTER TO StackFormat.StackObject] =
{
    lpStack: LONG POINTER TO CARDINAL _ LOOPHOLE[@lps.stack];
    IF lp.top = 0 THEN DOutput.Text["empty "L]
    ELSE
      FOR i: CARDINAL DECREASING IN [0..lp.top) DO
        DOutput.Octal[DebugOps.LongREAD[lpStack + i]]; DOutput.Char[' ];
        ENDLOOP;
    IF lp.overflowed THEN DOutput.Text["(overflow!) "L];
    IF lp.max = lp.top THEN DOutput.Text["(full!)"L];
    DOutput.Line[" "L]};

  PrintStack: PROC [f: DebugOps.Foo] = {
    g: LONG POINTER _ f.addr.base;
    DI.GetValue[f]; PrintRecord[f.addr.base, g]};

  Dump.AddPrinter[
    type: "LOOPHOLE[1400B, StackFormat$Stack]^", proc: PrintStack];

  END.
```