Inter-Office Memorandum

To	Mesa Users	Date	May 31, 1978
From	Barbara Koalkin	Location	Palo Alto
Subject	Mesa 4.0 Debugger Update	Organization	SDD/SD

XEROX

Filed on: [IRIS]<MESA>DOC>DEBUGGER40.BRAVO

This release of the Mesa debugger introduces many changes of interest and importance to all Mesa programmers. The purpose of this memo is to make you aware of the changes that have taken place. More complete explanations may be found in the *Mesa Debugger Documentation.*

Interpreter

The major addition to the Mesa 4.0 debugger is an interpreter that handles a subset of the Mesa language; it is useful for common operations such as assignments, dereferencing, indexing, field access, addressing, and simple type conversion. It is a powerful extension to the current debugger command language, as it allows you to more closely specify your variables while debugging, thus giving you more complete information with fewer keystrokes. A subset of the Mesa language has been specified as being acceptable to the interpreter (a copy of the grammar is attached to this memo).

Statement Syntax

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

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

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

Loopholes

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

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

Subscripting

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

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

Module Qualification

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

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

Type Expressions

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

Examples

Some of the old commands may now be simplified as follows:

```
Interpret Array [array,index,n] becomes array[index ! n]
Interpret Call [proc] becomes proc[param1, ..., paramN]
Interpret Dereference [ptr] becomes ptr↑
Interpret Expression [exp] becomes exp
Interpret Pointer [address,type] becomes address%@type↑
Interpret SIze [var] becomes SIZE[type]
Interpret STring [string,index,n] becomes string[index ! n]
Interpret @ [var] becomes @var
Display Variable [var] becomes var
Octal Read [address,n] becomes MEMORY[address ! n]
Octal Write [address,rhs] becomes MEMORY[address] ← rhs.
```

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

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

myWindow.menu.array[3].proc

which might give you the following output:

```
CreateWindow (PROCEDURE in WEWindows).
```

If you wanted to look at one of your procedure descriptors, you might type:

```
4601B%@procedure ControlDefs$ControlLink↑
```

which might produce the following output:

```
ControlLink[procedure[gfi: 23B, ep: 0, tag: procedure]].
```

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

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

would produce the following output:

```
3
1.
```

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

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

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

Process Commands

The debugger has added a set of commands for use with the new process capabilities of the language and the system. Display of the new data types is as follows: condition variables and monitor locks are displayed in octal; a process is displayed as PROCESS `[octal number]`. In all of the process commands, the message "`! is an invalid ProcessHandle`" or "`! not a process`" appears if the process is invalid.

`Set Process Context` **process**

sets the current process context to be **process** and sets the corresponding frame context for symbol lookup to be the frame associated with **process**. Upon entering the debugger for the first time, 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. Note also that when you set the octal context or module context, the process context is set to NIL; however, it is restored when you reset the context.

`Display Process` **process**

is a specialized version of `Display Variable` that 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 a condition variable (`waiting ML` or `waiting CV`). Then you are prompted with a "`>`" and enter process subcommand mode. A response of `N` displays the next process in the array of psbs; `R` displays the root frame of the

process; S displays the source text; P displays the priority of the process; and Q or DEL terminates the display and returns you to the command processor. A variable of type PROCESS (returned as the result of a FORK) or an octal **ProcessHandle** is acceptable as input to this command (note that **process** is an interpreted expression).

`Display Queue id`

displays all the processes waiting on the queue associated with **id**. For each process, you enter 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 is prepared to accept either a condition variable, a monitor lock, a monitored record, a monitored program, or an octal pointer (as in a pointer to the **ReadyList**). Note that **id** is an interpreted expression; if **id** is simply an octal number, you are asked whether it is a condition variable in order for the debugger to know where to find the head of the queue (i.e., `Display Queue: 175034B, condition variable? [Y or N]`).

`List Processes [confirm]`

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

Conditional Breakpoints, Multiple Proceeds, and new Breakpoint Syntax

The Mesa 4.0 debugger has extended the former set of breakpoint commands to include the capability to set conditional breakpoints.

The syntax for all of the Mesa 3.0 breakpoint commands remains basically the same with the following extension: if you type a SP after the procedure or module name you receive a prompt for the condition; if you type a CR it terminates the command input (in the case of entry/exit breaks) or just prompts for the source (in the case of text breaks), ie.,

```
Break Procedure: proc(SP), condition: x < 2 (CR), source: IF a = b (CR)
Break Procedure: proc(CR), source: IF a = b (CR).
```

The three valid formats for a conditional expression are:

variable relation variable - *eg., stop when x < y*

variable relation number - *eg., stop when x >= 10*

number - *eg., stop the 5th time you reach this breakpoint*

These commands accept relations belonging to the set: {<, >, =, #, <=, >=}, corresponding to: less than, greater than, equal, not equal, less than or equal, greater than or equal.

This gives us the ability to do multiple proceeds with the same syntax as simple conditional breakpoints.

Since the variables are interpreted expressions, they are looked up in the current context. However, if you are in a module context and wish to specify a local variable of the procedure you are setting the breakpoint in, you may do this by saying:

`proc.var` - *ie., use the local variable var defined in proc*

You may change the condition on a particular breakpoint or change a breakpoint from a conditional to a non-conditional one or vice-versa simply by setting the breakpoint again using the new condition.

There has also been a simplification to the breakpoint syntax as follows:

All commands to set breakpoints begin with the key letter `B`,
eg., `Break Procedure` instead of `SEt Procedure Break`

All commands to set tracepoints begin with the key letter `T`,
eg., `Trace Procedure` instead of `SEt Procedure Trace`

The keyword `Program` has been replaced by the word `Module`,
eg., `CLear Module Break` instead of `CLear Program Break`

All of the breakpoint commands now accept a valid **GlobalFrameHandle** as input when prompted for a module name.

New Commands / Changes to Existing Commands

`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.

`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.

`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**".

`Display Stack` subcommands

The `Display Stack` command now makes a distinction between displaying module (global) and procedure (local) contexts. The valid subcommands for local contexts remain as in Mesa 3.0: `n,p,v,r,s,q`; with the addition of the subcommand `>j, n(10)` which means jump down the stack `n` levels. Note that if `n` is greater than the number of levels it can advance, the debugger tells you how far it was able to go. Most of these subcommands apply to `Display Stack` on a global context with the exception of `j` and `n`. If the debugger cannot find a symboltable for some frame on the call stack, you get the message "`No symbols for nnnnnnB`" and enter restricted `Display Stack` subcommand mode in which only the subcommands `j`, `n`, and `q` are allowed. Note that a local context is displayed as the procedure name with its local frame, followed by the module name and its global frame; a global context is displayed as the module name and its global frame. For example,

```
>Display Stack -- on a global frame
StreamsA, G: 172674B  >?--Options are: p,q,r,s,v.
>Display Stack -- on a local frame
TArrays, L: 165064B (in TArrays, G:166514B)
  >? --Options are: p, v, r, s, q, j, n.
```

Notice that the convention, `proc, L:nnnnnnB (in module, G: nnnnnnB)`, applies throughout debugger output, wherever procedures and modules are displayed.

`CUrrent context`

The notion of the current context has been extended to include the current **ProcessHandle** as well as the name and corresponding global frame address of the current module and the current configuration.

Old commands that are gone

`Join Ports` and `Interpret SIze` have been taken out of the debugger's command language since their functions have been taken over by the debugger interpreter; `Display Binding path` has been removed since the concept of a binding path has gone away.

Additional Capabilities

More breakpoints - local procedures

Due to a change in the lookup algorithm for procedures, it is now possible to set breakpoints/tracepoints on a local (nested) procedure without being in the context of its enclosing procedure. However, in order to display a local procedure you must still be in its enclosing context.

Validity checking

The debugger makes a considerable effort to check if the user core image has been smashed in any way. When it determines that something is wrong, rather than printing out incorrect information it sets the context to NIL and disables all of the commands that rely on getting information either from symbol tables or from the loadstate. The user gets a message that says "`Current context invalid.`" or "`Command not allowed.`" whenever this situation occurs. At this time you might want to attach an image file or some symbols (see the `ATtach` comands) to find out what is wrong.

Installing

To install the debugger with a command line to the Alto Executive, use the "I" switch; use the "L" switch to load programs with code links (to save space). For example, typing `XDebug WindEx/i` installs the debugger with the window manager (WINDEX.BCD); typing `XDebug WindEx/il` installs WINDEX with code links.

Missing definition files

Instead of simply refusing to give you any information about your variables when you are missing a definitions file or have the wrong version of a file, the debugger prints a "?" to give you an indication that something is missing.

Invalid values.

The debugger will print "?[value]" when displaying enumerated types that appear to be wrong (ie., out of range).

Comments in the typescript file

A comment command has been added to the debugger command language. Use "--" to ignore type-in until a carriage return (CR). This is useful for saving your own notes along with your typescript file as well as type-in to be used for window selections.

Current Date and Time

The current date and time is inserted at the beginning of your typescript file along with the date and time that your version of the Mesa 4.0 debugger was created.

Default command

Typing the escape character (ESC) to the command processor of the debugger, uses the last command as the next valid command (i.e., you receive the prompts for the parameters (if any) for the previously executed command).

Confirming commands

When a command requires a [confirm] (CR), the debugger goes into wait-for-DEL mode if an invalid character is typed.

Extended Features

See the *Debugger - Extended Features* memo for further details on the following.

FTP in the debugger

The ↑FTP command (control-F) is used to provide file transfer capabilities from within the debugger using the standard FTP package. Any comments and/or problems regarding FTP itself should be addressed to the Communications Group. If FTP has not been loaded, trying to use any of the FTP commands will give you the message "-- FTP not installed".

UserProcs

The ↑UserProc command (control-U) allows you to load your own debugging package into the debugger. If you have only one user proc loaded when you type control-U, it will be invoked automatically. If you have several user procs loaded, typing "?" will give you a list of the command names for the user procs that you have loaded. If it can't find any procedures that have been loaded, you will just get the message, " !No user procs are currently loaded".

Window manager

The new window manager **WindEx** has several commands which allow you to set breakpoints and tracepoints by selecting text locations. Confirmation is given by moving the selection to the place at which the breakpoint is actually set.

Internal Changes

Debugger Nub

The Mesa 4.0 debugger has been able to realize a significant space reduction by removing its own internal debugging facilities and replacing them with a nub. Typing ↑D to the command processor brings you into the debugger nub with a "//" prompt. The following limited set of commands are available in the nub: `Install`, `Bitmap`, `New`, `Start`, `Proceed`, and `Quit`. `Bitmap[n(10)]` reallocates the bitmap to n pages (the default size is about 50 pages). The nub also provides a minimal signal catcher and interrupt handler as well as primitive debugging facilities. It is possible to install a different version of the debugger to use for debugging the debugger itself (see a member of the Mesa Group if you are interested in knowing more about how this works).

Savings in space

The global frame size of the Mesa 4.0 debugger has been reduced by over 50% from the previous release. This has created space for the interpreter and for a larger bitmap. Some of the savings is due to the split of the internal and external debugger. Another reason is due to the way in which the debugger handles strings. By putting the command strings, command prompts, signal and error messages, and debugger FTP commands into a separate file (and running this file through the string compactor), the string segment can be swapped in only when needed. Additional space was saved by making many of the remaining strings into local strings so that they do not take up space in the global frame.

Documentation

More complete documentation on the Mesa 4.0 Debugger may be found in the *Mesa Debugger Documentation.* Bug fixes may be found in the closed change requests maintained by **<SDSupport>.**

Debugger Summary

Version 4.0

AScii read [address, n]
ATtach Image [filename]
Symbols [globalframe, filename]
Break Entry [proc, condition]
Module [module, condition, source]
Procedure [proc, condition, source]
Xit [proc, condition]
CAse off **[confirm]**
on **[confirm]**
CLear All Breaks **[confirm]**
Entry traces [module]
Traces **[confirm]**
Xit traces [module]
Break [proc, source]
Entry Break [proc]
Trace [proc]
Module Break [module, source]
Trace [module, source]
Trace [proc, source]
Xit Break [proc]
Trace [proc]
COremap **[confirm]**
CUrrent context
Display Configuration
Eval-stack
Frame [address]
GlobalFrameTable
Module [module]
Process [process] - **n,p,q,r,s**
Queue [id]
Stack - **j,n,p,v,r,s,q**
Variable [id]
Find variable [id]
Interpret Array [array, index, n]
Call [proc]
De-reference [ptr]
Expression [exp]
Pointer [address, type]
String [string, index, n]
@ [var]

Kill session **[confirm]**
List Breaks **[confirm]**
Configurations **[confirm]**
Processes **[confirm]**
Traces **[confirm]**
Octal Clear break [globalframe, bytepc]
Read [address, n]
Set break [globalframe, bytepc]
Write [address, rhs]
Proceed **[confirm]**
Quit **[confirm]**
Reset context **[confirm]**
SEt Configuration [config]
Module context [module]
Octal context [address]
Process context [process]
Root configuration [config]
STart [address]
Trace All Entries [module]
Xits [module]
Entry [proc,condition]
Module [module, condition, source]
Procedure [proc, condition, source]
Xit [proc, condition]
Userscreen **[confirm]**
Worry off **[confirm]**
on **[confirm]**
-- [comment]

Debugger Interpreter Grammar

Version 4.0

StmtList	::=	Stmt | StmtList; Stmt
AddingOp	::=	+ | -
BuiltinCall	::=	LENGTH [LeftSide] | BASE [LeftSide] | DESCRIPTOR [Expression] | DESCRIPTOR [Expression , Expression] | SIZE [TypeSpecification]
Expression	::=	Sum
ExpressionList	::=	Expression | ExpressionList, Expression |
Factor	::=	- Primary | Primary
Interval	::=	Expression .. Expression | Expression ! Expression
LeftSide	::=	identifier | Literal | MEMORY [Expression] | LeftSide Qualifier | (Expression) Qualifier | identifier $ identifier | numericLiteral $ identifier
Literal	::=	numericLiteral | stringLiteral | -- all defined outside the grammar characterLiteral
MultiplyingOp	::=	* | / | MOD
Primary	::=	LeftSide | (Expression) | @ LeftSide | BuiltinCall
Product	::=	Factor | Product MultiplyingOp Factor
Qualifier	::=	. identifier | ↑ | % | % TypeSpecification | [ExpressionList]
Stmt	::=	Expression | LeftSide ← Expression | MEMORY [Interval] | LeftSide [Interval] | (Expression) [Interval]
Sum	::=	Product | Sum AddingOp Product
TypeConstructor	::=	@ TypeSpecification
TypeIdentifier	::=	INTEGER | BOOLEAN | CARDINAL | CHARACTER | STRING | UNSPECIFIED | identifier | identifier $ identifier | identifier TypeIdentifier
TypeSpecification	::=	TypeIdentifier | TypeConstructor

Windex Summary

Version 4.0

WHAT WINDEX MOUSE BUTTONS DO:

	Scroll Bar	Text Area
RED	ScrollUp	Select/Extend characters
YELLOW	Thumb	Select/Extend words
BLUE	ScrollDown	Menu Commands
YELLOW/BLUE	NormalizeSelection	

MENU COMMANDS:

Create [window]
Destroy [window]
Move [window]
Grow [window]
Load [selection, window]
Stuff It [selection, window]
Find [selection, window]
Set Brk [selection]
Clr Brk [selection]
Set Trc [selection]
Set Pos [index, window]
Keys On/Off

WHAT MENU MOUSE BUTTONS DO:

RED	"Do it" - in this window/ at this spot
BLUE	Reset to previous state

WHAT KEYSET BUTTONS DO:

BS	DEL	ESC	CR	STUFF IT

DURING TYPE IN:

BS	Backspace character
CONTROL-W	Backspace word
FL4	Stuff current selection into default window

Fetch Command Summary

Close connection [confirm]
DElete filename [filename]
DUmp from remote file [dumpfile]
Free pages
LIst remote file designator [filelist]
LOad from remote file [dumpfile]
Open connection [host, directory]
Quit [confirm]
Retrieve filename [filename]
Store filename [filename]