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!*

Copyright c Xerox Corporation 1980

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.
```