Inter-Office Memorandum

To	Mesa Implementors	Date	October 13, 1978
From	Barbara Koalkin	Location	Palo Alto
Subject	Specifics for Debugger Interface	Organization	SDD/SD

XEROX

Filed on: [IRIS] < KOALKIN > D5 > DUI2.BRAVO

DRAFT

This memo is the specification for a new user interface for the Mesa 5.0 debugger; it has been drawn together from suggestions from the Mesa task list, Greg Shaw's suggestion list (based on the experience of ETM), the change requests classified as debugger wishes, experiments with the Tools Environment and the SmallTalk Browser, and discussion with experienced Mesa users.

The design is based on the following observations and goals:

* We are aiming for the experienced systems programmer, *not* a clerical person with minimal training who will quit in 6 months.
* We must build on the capabilities (and code) of the current Mesa debugger as well as maintaining the present Mesa style of debugging.
* Speeding up the performance and increasing reliability is as important as improving the interface.
* We should try to cut down on the edit-compile-debug loop as much as possible.
* Keep it simple.

The basic facilities of the debugger can be divided into the following categories: setting breakpoints and tracing program execution, examining (and changing) the runtime state, setting the context in which user symbols are looked up, directing program control, and a collection of less frequently used low-level utility commands. By supplementing the present teletype command processor interface to these facilities with more window/selection/menu based capabilities, we can both speed up and ease the debugging process.

The new debugger interface looks as follows:

* The debugger is initialized with 3 windows: DEBUG.TYPESCRIPT window, a sourcefile window, and a window for manipulating the stack (DEBUG.STACK).
* Each of these windows has a menu containing the standard set of window operations: MOVE (change the position of the window), GROW (change the size of the window), BIG (make the window as large as the entire screen/restore), and SMALL (make the window as into a small window at the side of the screen). The menu also contains the basic selection operations that are common to all windows: STUFF IT (stuffs the selection of the window into the keyboard stream of the DEBUG.TYPESCRIPT window), FIND (finds the next occurence of the selection in the window).
* Each window has its associated filename in the header of the window.

* Each window may have a selection, but only one selection is "the current selection" at any time. This simplifies input to commands that involve using the current selection.
* The mouse buttons remain as before: RED selects and extends the selection by characters, YELLOW selects and extends the selection by words, and BLUE displays the menu.
* The scrolling functions remain as before: RED for scrolling up, YELLOW for thumbing, BLUE for scrolling down, and YELLOW/BLUE for normalizing the selection. Possible enhancements to the present capabilities are to have continuous scrolling and split windows.
* Look into what is involved in adopting the Tools window/menu/selection/editor package.

Debug.Typescript window:

* Retains the present debugger interface capabilities with respect to typing in key letters for invoking commands.
* Is the only debugger-created window that accepts type-in. This means that the STUFF-IT key can continue put characters into the input stream of this window; thus typing a character (when a non-scratchfile window is current) makes this typescript window the current window.
* Used to interpret expressions.
* The place of all debugger to user communcation. This window is used for reporting uncaught signals, error messages from the interpreter, and saving lists of information (such as List Breaks or Display GlobalFrameTable).
* In addition to the standard window operations, the menu for this window has a command to allow you to CREATE a new scratch window.
* The Debug.Typescript file can still be viewed as a log of the debugging session. If a user wishes to save some of the information that is lost by using selections instead of typing in all of the commands, you can record information in this window (in the form of comments) or use the old type-in command processor.

Sourcefile window:

* Used to set breakpoints and for displaying the source position when looking at the stack.
* This window is the only window in which you may set breakpoints. These breakpoint commands enable the user to perform all of the present breakpoint operations simply by selecting the location and choosing the appropriate menu command.
* The semantics of the breakpoint commands are as follows:

Keyword	Selection	Action (old command)
BREAK	PROCEDURE	Break Entry
	RETURN	Break Xit
	source	Break Procedure, Break Module
CBREAK	PROCEDURE	same as above commands but must
	RETURN	specify condition as post-operator
	source	
CLEAR	PROCEDURE	Clear Entry Break
	RETURN	Clear Xit Break
	source	Clear Break, Clear Module Break
BR ALL	PROCEDURE	Break All Entries
	RETURN	Break All Xits
CL ALL	PROCEDURE	Clear All Entries
	RETURN	Clear All Xits
	PROGRAM	Clear All Breaks

* This scheme allows us to set conditional breakpoints by selecting the location, invoking the CBREAK command, and then typing in the condition. Where: either a window pops up (it is nice for this information and its window to go away after being input) or into the typescript window (which is already there but possibly not visible)? See figure 1.

* The type of menu to be used is as yet undetermined; either a menu with the breakpoint commands in addition to the window commands (as in the present scheme) or perhaps a fixed menu consisting of the keywords BREAK, CLEAR, CONDITION, and ALL, located either in the header of the window along with the file name or on the bottom of the window, that allows combinations of keywords to be selected, with BREAK and CLEAR actually activating the action. More experimenting and discussion is still neccesary to resolve this.

* All breakpoints are shown by a graphic indication in the source file window. What to use: secondary selection highlighting, or a carat beneath the location? This means the window needs to be refreshed (somehow) each time any breakpoint is set or cleared.

* Selections in the breakpoint window resolve only to places where breakpoints are allowed to be set (according to the compiler-generated fine grain table).

Debug.stack:

* The stack window is used for displaying current context information as well as the procedure call stack. Whichever level of the stack is selected becomes the current context for symbol lookup. It possible to change the current context by moving the selection in either direction along the stack.

* A subwindow of the stack window is reserved for showing context information about the current configuration, psb, module, global frame, and local frame. The rest of this subwindow is used to show the procedure call stack, one level at a time. If you wish to advance along the stack, select the NEXT menu command or hit the next key. How about using the LF key for the next key? You may go back up the stack by selecting the name at the level you wish to look at. How to do jump (as in skip the next n levels)?

* The rest of the stack window is used to show the variables local to the current context. Should this be two separate windows? Should the variable window be cleared for each SHOW or just scrolled? This subwindow has its own scrolling capabilities; stack commands are activated by means of a menu. See figures 2 and 3 for examples. The semantics of the commands are as follows:

Keyword	Selection	Action (old command)
SHOW	*config*	Display Configuration
	psb	Display Process - p, r, w
	module	Display Module - v
	procedure	Display Stack - v
SOURCE	*module*	Display Module - s
	procedure	Display Stack - s
	psb	Display Process - s
NEXT	*procedure*	Display Stack - n
	psb	Display Process - n

* This window may also be used for changing the context. Selecting one of the context keywords (Configuration, Process, Module) means "change this context". When the current context gets modified in some way, all of the context information gets updated. See the section below for details.

* The stack window does not allow user type-in. Updating gets done by the debugger when the context gets modified in some way. Only word selection is allowed in this window, since selections are only used for context setting purposes. Note that all messages like "No symbols for nnnnnB..." and "Sourcefile.mesa not available" continue to be displayed in the DEBUG.TYPESCRIPT window.

Changing the context:

* If you wish to change the context, select one of the context keywords. A (temporary) window (or call it a "view") appears with a list of the choices. Selecting the name changes the context as well as updating all of the corresponding information in the context status subwindow.

* If you select the keyword "Configuration", a window appears, called "List.configs", consisting of a list of all the configurations that have been loaded. Note that this is the same output as the present List Configurations command.

* If you select the keyword "Module", a window appears, called "List.modules", consisting of a list of the names of all of the modules in the current configuration. Note that this is similar to the output of the present Display Configuration command. See figure 4 for an example.

* If you select the keyword "Process", a window appears, called "List.processes", consisting of a list of all processes by ProcessHandle. What other information is neccesary in order for this to be interesting: frame, root, source, priority?

Directing program control:

* Explore the idea of using the header of the DEBUG.TYPESCRIPT window to contain a menu of the PROCEED, QUIT, and KILL commands in order to have a menu way of directing program control.

Scratch window:

* The menu for this type of window has the standard window operations in addition to commands to CREATE a new scratch window, DESTROY a scratch window, and LOAD a file into a scratch window.

* This type of window can also accept keyboard type-in.

Mode changes:

* The debugger keeps a "property sheet" of state information that is not likely to change often. When you wish to examine or change any of these modes, you invoke the Mode Change command. This causes a window to appear (like the specification of a Tool or a property sheet in Desktop) in which you can reverse a mode by selecting ON/OFF. This command is used to set global state information that is currently maintained by the commands CAse on/off, Worry on/off, Keys on/off. What to call this command and how to invoke and display it? We should look into the possibility of extending this sheet to include other options (for example, whether you would like to have records displayed with spaces or carriage returns between the fields).

Consistent rules for invoking commands:

* In general, the commands that require no parameters (Userscreen), print many lines of information (Coremap), take more time to complete (List Processes), change the state (Reset Context), and direct program control (Proceed) are the types of commands that require confirmation (CR) before they are executed.

* Menu commands with just one operand use the current selection as the object of the action, and activate when the menu button is let up.

* Menu commands that require two arguments (such as conditional breakpoints), use the current selection as the first argument and prompt (how?) for the second one when the menu button is let up.

* All octal commands (Octal Read, Write, Set break, Clear break, Set Octal Context) may be invoked only through the command processor. The same applies for other low-level

utility commands.

Changes to the way things are now:

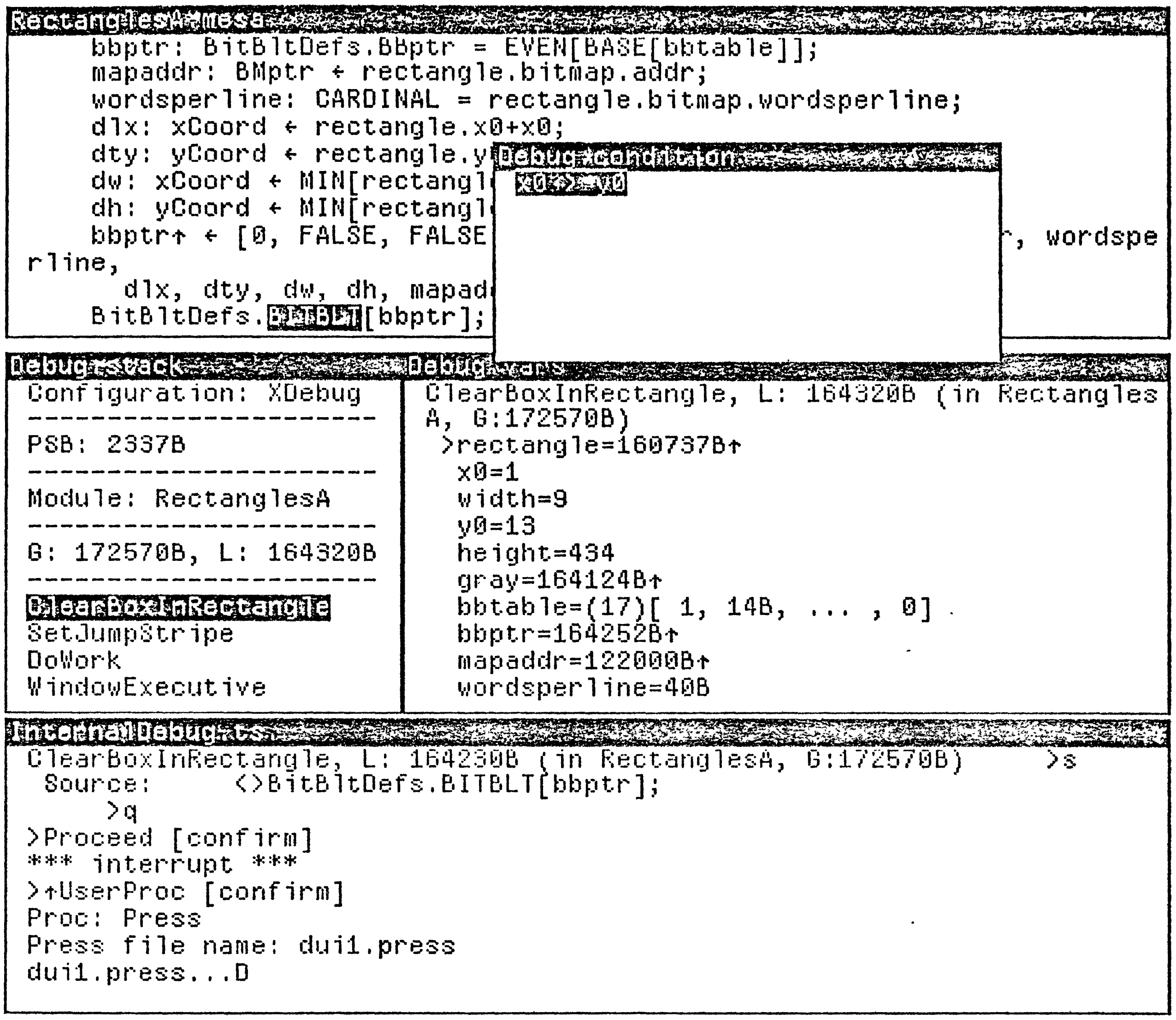
* As a result of implementing more selection based schemes for invoking debugger commands, WindEx has to become a standard part of the Mesa debugger.

* The need for the MESA.TYPESCRIPT window has gone away, so this feature is no longer supported.

* Different types of windows have their own menu commands in addition to the basic set of window operations.

* The distinction between the old form of tracepoints and breakpoints has almost disappeared with the new display stack mode. Therefore only breakpoints are now being supported, with tracepoints being reserved for future design in combination with a macro facility. You would like to be able to specify a set of actions to be performed (including the ability to proceed) when reaching a tracepoint, without the user having to be there to type in the commands.

* The functions of the Display Variable and Interpret commands in Mesa 4.0 debugger have been superceded by the interpreter and therefore are no longer supported.



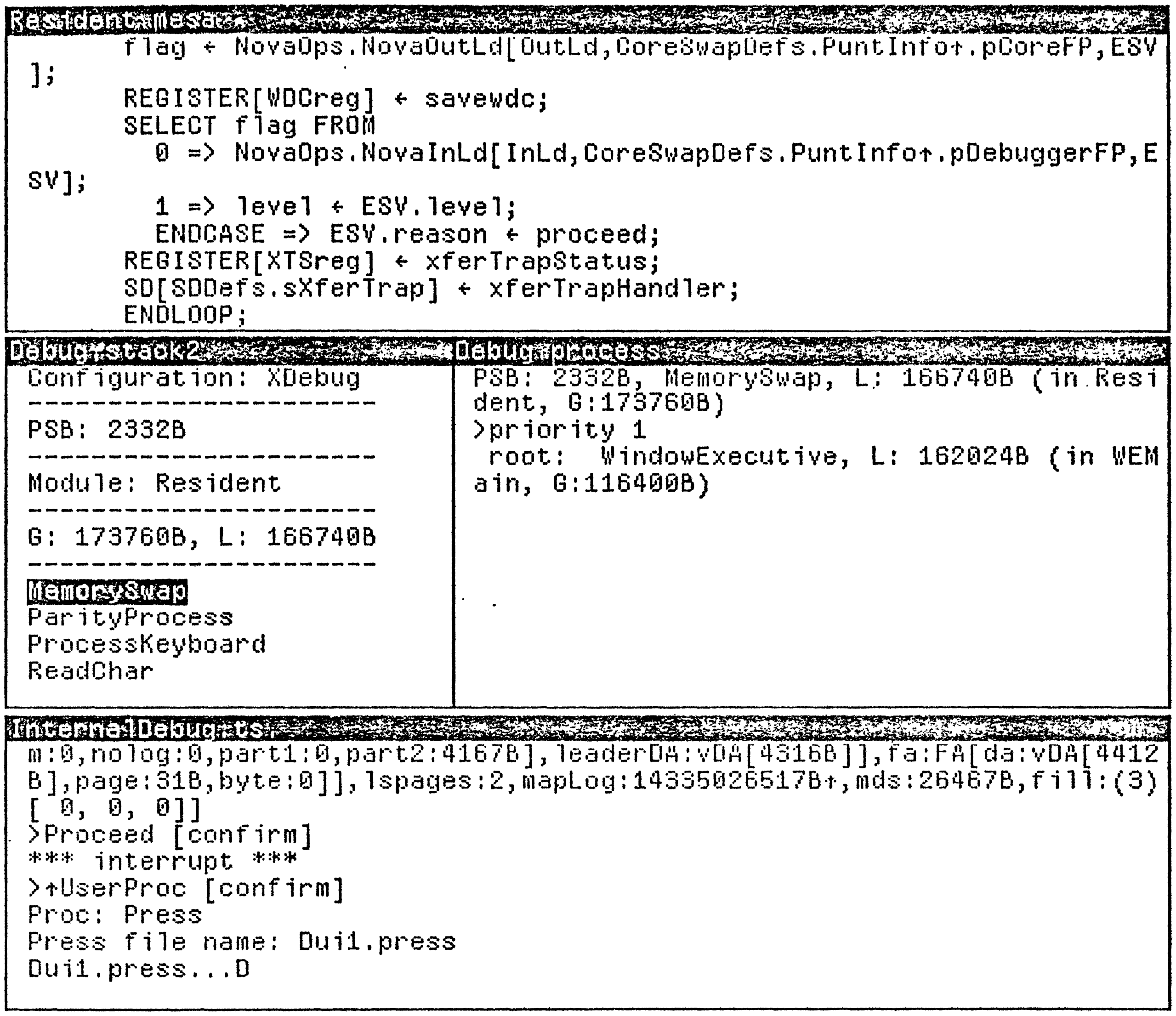
```
    bbptr: BitBltDefs.BBptr = EVEN[BASE[bbtable]];
    mapaddr: BMptr ← rectangle.bitmap.addr;
    wordsperline: CARDINAL = rectangle.bitmap.wordsperline;
    dlx: xCoord ← rectangle.x0+x0;
    dty: yCoord ← rectangle.y
    dw: xCoord ← MIN[rectangl
    dh: yCoord ← MIN[rectangl
    bbptr↑ ← [0, FALSE, FALSE                              r, wordspe
rline,
      dlx, dty, dw, dh, mapad
    BitBltDefs.BITBLT[bbptr];
```

```
x0<>y0
```

```
Configuration: XDebug
----------------------
PSB: 2337B
----------------------
Module: RectanglesA
----------------------
G: 172570B, L: 164320B
----------------------
ClearBoxInRectangle
SetJumpStripe
DoWork
WindowExecutive
```

```
ClearBoxInRectangle, L: 164320B (in Rectangles
A, G:172570B)
 >rectangle=160737B↑
  x0=1
  width=9
  y0=13
  height=434
  gray=164124B↑
  bbtable=(17)[ 1, 14B, ... , 0]
  bbptr=164252B↑
  mapaddr=122000B↑
  wordsperline=40B
```

```
ClearBoxInRectangle, L: 164230B (in RectanglesA, G:172570B)     >s
 Source:     <>BitBltDefs.BITBLT[bbptr];
      >q
>Proceed [confirm]
*** interrupt ***
>↑UserProc [confirm]
Proc: Press
Press file name: dui1.press
dui1.press...D
```

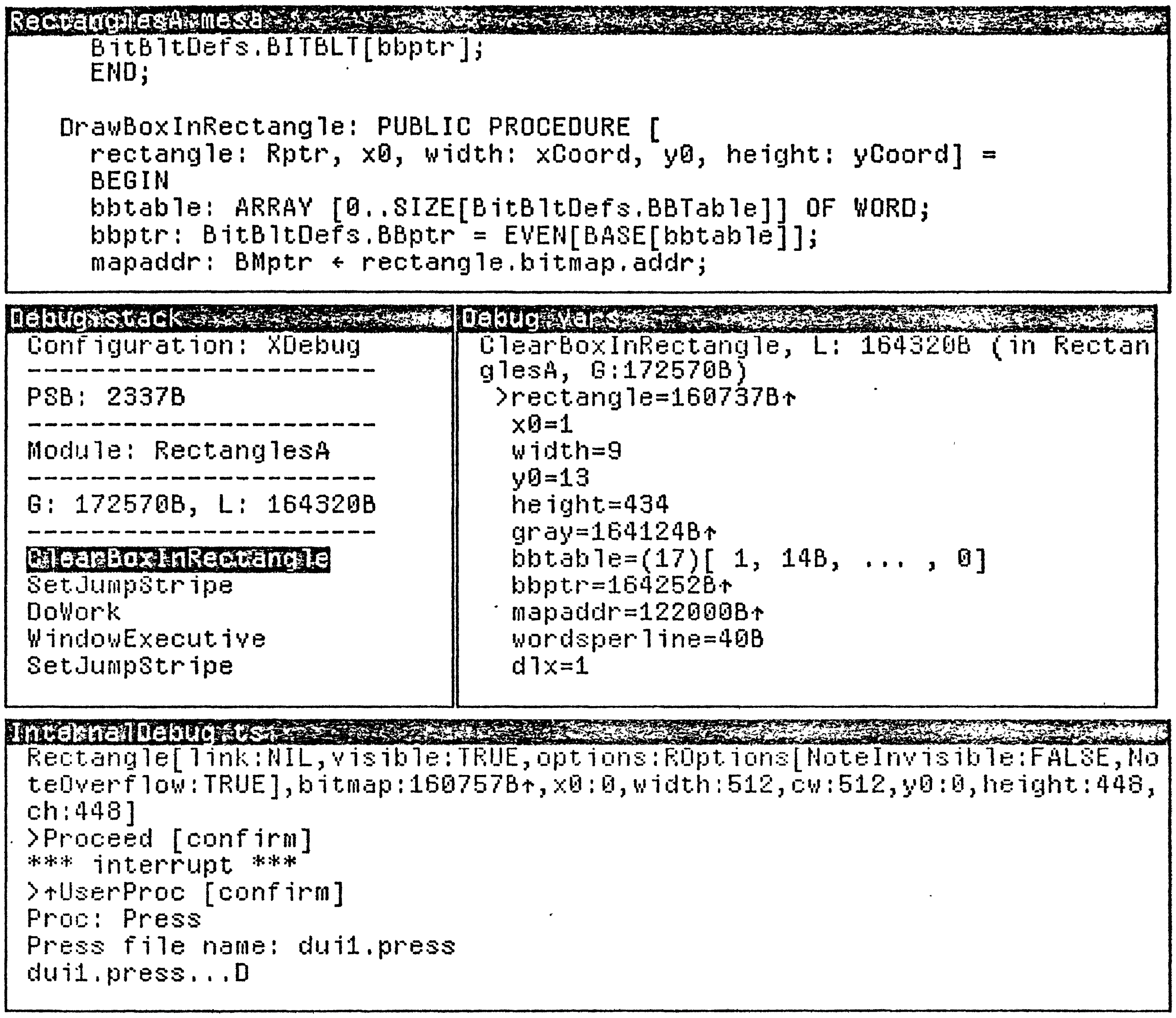


```
      flag ← NovaOps.NovaOutLd[OutLd,CoreSwapDefs.PuntInfo↑.pCoreFP,ESV
];
      REGISTER[WDCreg] ← savewdc;
      SELECT flag FROM
        0 => NovaOps.NovaInLd[InLd,CoreSwapDefs.PuntInfo↑.pDebuggerFP,E
SV];
        1 => level ← ESV.level;
        ENDCASE => ESV.reason ← proceed;
      REGISTER[XTSreg] ← xferTrapStatus;
      SD[SDDefs.sXferTrap] ← xferTrapHandler;
      ENDLOOP;
```

```
Configuration: XDebug
----------------------
PSB: 2332B
----------------------
Module: Resident
----------------------
G: 173760B, L: 166740B
----------------------
MemorySwap
ParityProcess
ProcessKeyboard
ReadChar
```

```
PSB: 2332B, MemorySwap, L: 166740B (in Resi
dent, G:173760B)
>priority 1
 root:  WindowExecutive, L: 162024B (in WEM
ain, G:116400B)
```

```
m:0,nolog:0,part1:0,part2:4167B],leaderDA:vDA[4316B]],fa:FA[da:vDA[4412
B],page:31B,byte:0]],lspages:2,mapLog:14335026517B↑,mds:26467B,fill:(3)
[ 0, 0, 0]]
>Proceed [confirm]
*** interrupt ***
>↑UserProc [confirm]
Proc: Press
Press file name: Dui1.press
Dui1.press...D
```



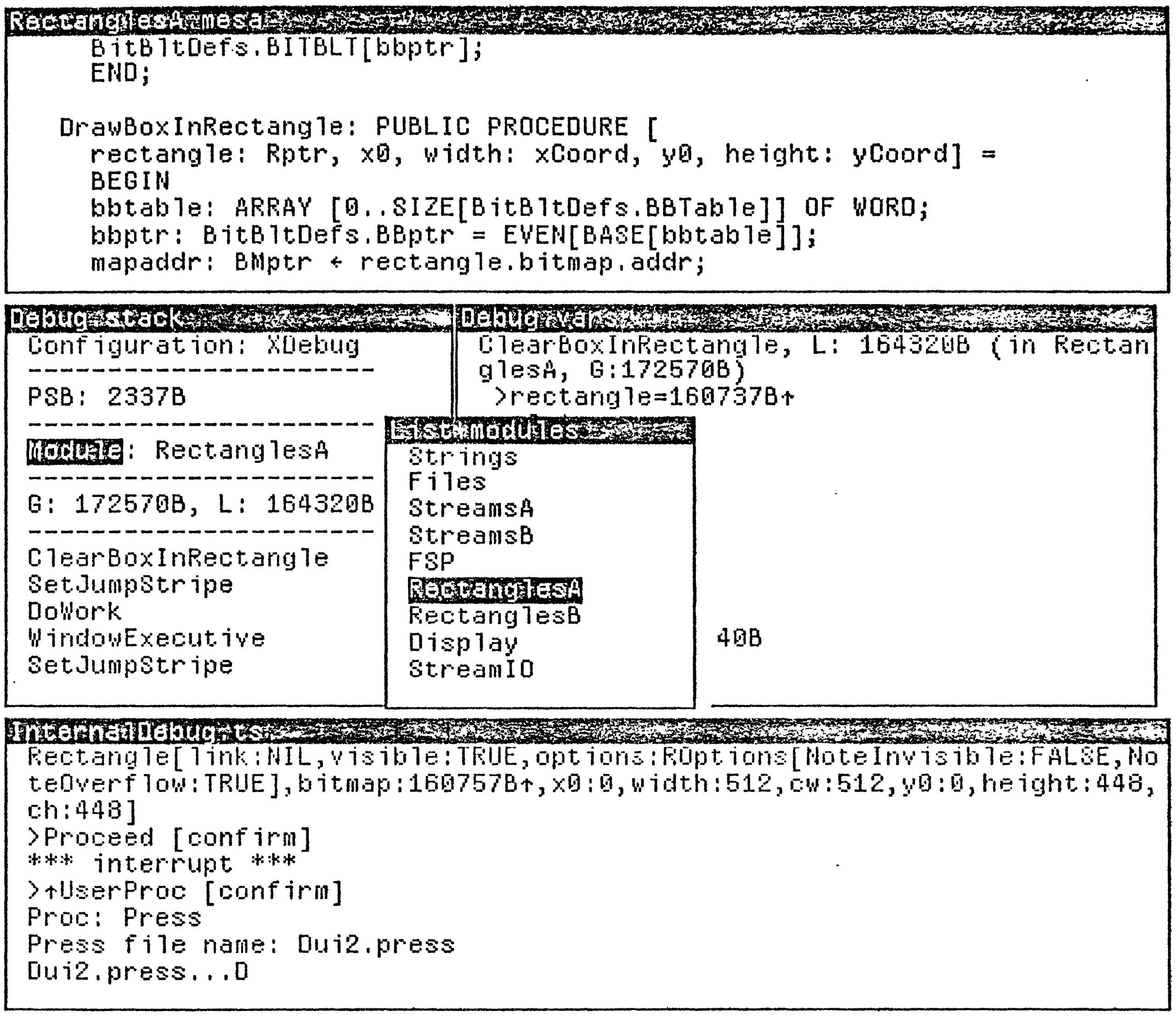
```
RectanglesA.mesa
    BitBltDefs.BITBLT[bbptr];
    END;

  DrawBoxInRectangle: PUBLIC PROCEDURE [
    rectangle: Rptr, x0, width: xCoord, y0, height: yCoord] =
    BEGIN
    bbtable: ARRAY [0..SIZE[BitBltDefs.BBTable]] OF WORD;
    bbptr: BitBltDefs.BBptr = EVEN[BASE[bbtable]];
    mapaddr: BMptr ← rectangle.bitmap.addr;
```

```
Debug.stack
 Configuration: XDebug
 ----------------------
 PSB: 2337B
 ----------------------
 Module: RectanglesA
 ----------------------
 G: 172570B, L: 164320B
 ----------------------
 ClearBoxInRectangle
 SetJumpStripe
 DoWork
 WindowExecutive
 SetJumpStripe
```

```
Debug.Vars
 ClearBoxInRectangle, L: 164320B (in Rectan
 glesA, G:172570B)
  >rectangle=160737B↑
   x0=1
   width=9
   y0=13
   height=434
   gray=164124B↑
   bbtable=(17)[ 1, 14B, ... , 0]
   bbptr=164252B↑
   mapaddr=122000B↑
   wordsperline=40B
   dlx=1
```

```
InternalDebug.ts
 Rectangle[link:NIL,visible:TRUE,options:ROptions[NoteInvisible:FALSE,No
 teOverflow:TRUE],bitmap:160757B↑,x0:0,width:512,cw:512,y0:0,height:448,
 ch:448]
 >Proceed [confirm]
 *** interrupt ***
 >↑UserProc [confirm]
 Proc: Press
 Press file name: dui1.press
 dui1.press...D
```



```
    BitBltDefs.BITBLT[bbptr];
    END;

  DrawBoxInRectangle: PUBLIC PROCEDURE [
    rectangle: Rptr, x0, width: xCoord, y0, height: yCoord] =
    BEGIN
    bbtable: ARRAY [0..SIZE[BitBltDefs.BBTable]] OF WORD;
    bbptr: BitBltDefs.BBptr = EVEN[BASE[bbtable]];
    mapaddr: BMptr ← rectangle.bitmap.addr;
```

```
Configuration: XDebug
----------------------
PSB: 2337B
----------------------
Module: RectanglesA
----------------------
G: 172570B, L: 164320B
----------------------
ClearBoxInRectangle
SetJumpStripe
DoWork
WindowExecutive
SetJumpStripe
```

```
ClearBoxInRectangle, L: 164320B (in Rectan
glesA, G:172570B)
 >rectangle=160737B↑

40B
```

```
Strings
Files
StreamsA
StreamsB
FSP
RectanglesA
RectanglesB
Display
StreamIO
```

```
Rectangle[link:NIL,visible:TRUE,options:ROptions[NoteInvisible:FALSE,No
teOverflow:TRUE],bitmap:160757B↑,x0:0,width:512,cw:512,y0:0,height:448,
ch:448]
>Proceed [confirm]
*** interrupt ***
>↑UserProc [confirm]
Proc: Press
Press file name: Dui2.press
Dui2.press...D
```