Inter-Office Memorandum

To Debugger Planners

Date June 9, 1978

From Barbara Koalkin

Location Palo Alto

Subject Pilot Debugger

Organization SDD/SD/DE

XEROX

Filed on: [Iris]<Koalkin>XD>PilotDebugger6-8.bravo

The following lists the set of debugger problems proposed by Richard Johnsson in his memo of 6/7 along with suggestions as to how these problems may be solved as discussed in a meeting of Pilot Implementors (Johnsson, Kierr, Koalkin, Lauer, Lynch, McJones, Redell, Sandman, and Wick) on Thursday, June 8th.

For puposes of discussion, it was decided to call stage 1 of the Pilot debugger the Extended Memory Debugger (with long pointers and codebase, virtual = real, MDS starts at 0, no page faults), and call stage 2 the Paging Debugger (with virtual memory).

It was pointed out many times during the course of this meeting, the importance of remembering that *the debugger wants to have no knowledge of the underlying Pilot data structures and not to have to rely on Pilot for any information* (since its early mode of use will primarily be debugging Pilot itself).

Problems to be solved for Pilot Debugger:

1. Full integration of LONG POINTERs into the interpreter.
   Can the debugger's type calculus deal with them?
   What about LONG INTEGERs and long type-in?

The debugger interpreter is prepared to handle long pointers in terms of dereferencing them and could easily be modified to do long arithmetic. However, extending the grammar to generate variables of type LONG is much more difficult (but there needs to be a way of typing in a long number).

2. Debugger's memory cache must deal with LONG POINTERS.
   LongRead and **LongWrite** needed.
   Interpretation of Pilot's VM to DA log file.
      Format, interface, how to find.

In order for the debugger to be able to understand and manipulate the Pilot virtual memory, the debugger's memory cache (which currently defines all memory access using a READ/WRITE procedural interface), must use LONG POINTERs instead. **LongRead** and **LongWrite** operations are needed, not specific to Pilot, and can be done once the mapping is well defined. The short READ and WRITE operations in the Mesa 4.0 debugger will have to be changed to call the long read and write operations after doing the MDS calculation. This change is necessary for the Extended Memory Debugger.

As far as the Pilot VM to DA log file is concerned, we need to nail down the interface

between Pilot and the debugger to answer questions such as where the log file will be kept, how to turn it on and off, how to keep the log file from getting too big (and who should do this), exactly what information it will contain, and how to find it (it was decided to define the details of this interface at another meeting). This needs to be done before work can proceed on the Paging Debugger.

3. Changing code files for breakpoints:
   When do we make a copy of the code?
   How do we tell Pilot about it?
   Store broken instruction in code? How many? Packed code?
   Pin affected pages in real memory?

We discussed 5 possibilities for solving the breakpoint problem:

(1) copy code and fix user pointers as in the Mesa 4.0 debugger
(no way to tell this to Pilot)

(2) lock page in memory when you set the breakpoint there
(but how to get it there first? performance?)

(3) leave a half page empty at the end of each codesegment to store the break information
(what about packed code? what about Trace All Entries?)

(4) get new disk address for any virtual page
(need a reverse VM to DA map on a page basis)

(5) the diamond solution: just change the files themselves and append breakpoint information to code file
(must be undone before leaving or else the code is clobbered)

McJones suggested that Pilot could reserve a segment of virtual memory (ie. 64K, the size of Swatee) for the debugger to put code segments for breakpoints. This would be copied into backing store and change the code pointer in the global frame. It was decided that this was the best way for it to be done.

Lynch pointed out that we need to watch the performance issue (changing things out from under the user, not space/time), since this strategy may change the nature of the bug that is being tracked. The problem is moving the code in the virtual address space with the codepointer possibly being on the stack. The restriction that will have to be placed in order to get around this problem, is that we do not garbage collect this region until the end of a session (ie., the sum of all of the codesegments in which breaks have been set must be < 64K).

If this seems to cause undue hardship, we could introduce a mode similar to worry mode, which puts the breaks directly in the code, and lets you take your chances on having the code get clobbered. (Note with the current **StartUp** strategy of the file being recreated from the bcds at startup time, this is not as much of a problem as it was first thought to be).

4. Compatibility - evolution
   How do the Pilot and Mesa 4.0 (Alto or D0) debuggers differ?
   Is the difference at compile time, install time, or **InLoad** time?

It was decided that is would be easier (both for implementing and maintaining), to have one version of the debugger with the decision being made at **Inload** time. This does have the disadvantage that everybody pays for the added code required for the Pilot debugger, but it is thought that this is not too much overhead too pay.

5. Communications

   InLoad and **OutLoad** pass a message (18 words) back and forth. What is in it? How will **InLoad** be done? What about microcode swapping? Map swapping?

Resident changes should be limited to **InLoad** and **OutLoad** (which require changes to Nova code and microcode), alloc trap (needs to call Pilot), and the ability to handle worry mode breakpoints. Eight words of the message are currently unused; this should be enough space for all of the additional information that we decide is neccesary. It is hoped that the microcode will be able to tell what is going on and switch the map (exchange the first and last 64K of map) while the Nova code is running. We will have to talk to Garner about this.

Ths issue of microcode swapping was postponed for now since it was decided that it is not on the critical path for completing the Pilot debugger and it can be a time sink.

6. How does the debugger know what it is debugging?

   Install time or **InLoad** time?
   If D0/Pilot, where is the **MDS**?
   What about multiple **MDSs** . . . ?

The debugger knows what machine it is debugging from the microcode and where to find the MDS from **PilotNub** (who gets the information from **Startup** (**BootMesa**)).

Lynch said that there will not be multiple MDSs until at least Pilot 3.0; therefore we will not worry about it in the Pilot debugger (however, the design will include provision for adding this later without undoing what already has been done).

There is a need for a `SEt MDS` command, which takes a page number; useful on bootloading the debugger and moving the MDS to a different place (note that this means that each time the MDS is changed, many of the debugger's caches will have to be flushed due to a new global frame table, etc.). This goes along with extending the notion of current context to include the MDS as well; however, this may be postponed for a while until we are dealing with multiple MDSs.

Problems arise on bootloading the debugger, when real memory (and the map) is gone. There was a lengthy discusion of alternative solutions to this problem; many questions remain, including investigating whether the memory map could survive booting, the difference between the soft boot and power up/off, and getting a dumper.boot to save memory (as in the current world). An immediate solution is to resort to debugging with Midas when things get this bad. (It was decided to continue discussion on this issue at a later time).

7. **DebugNub**'s responsibilities

   Finding the debugger
   LoadState
   Breakpoints

PilotNub can be given the file information about where to find the debugger from **Bootmesa**.

Since there probably will not be a loader in Pilot 2.0, there is no need for a loadstate (there will be only one bcd). But **Bootmesa** can initialize the loadstate for now.

There will be a problem in finding files when the debugger has been bootloaded; the solution to this seems to be to force the user to enter restricted debugging mode whenever the files cannot be found easily. Since the file information can be gotten from **Wart (StartUp)**, it seems better than making the decision at install time or having separate debuggers.

8. When do things start moving into Pilot File System?
   What are they?

As soon as possible, things will start moving into the Pilot File System, beginning with the anonymous files.

The going inposition is that we will try to accomplish all of this without changing the Mesa definitions files, and confine the changes to the debugger to those modules that deal with file manipulation, breakpoints, and interpreting long variables.

The discussion then turned to the issue of scheduling. Wick guessed that it would take 2 weeks of Sandman, Johnsson, and Koalkin working full time to get stage 1 complete, plus about 1 1/2 weeks of debugging time on the D0 (assuming all of the long pointer things have been checked out). The stage 2 debugger can be completed in a shorter time, about 1 week of design plus 1 week of debugging.

Lynch and the Pilot group need to decide if, with this schedule, the debugger will be useful to them when they get it. Wick offered another debugger, with long pointer READ and WRITE as a first step if that would be useful. It could be used for examining memory > 64K but not any code out there. This brought up the discussion of the hybrid compiler (which Wick guessed would take Sweet about 3 days to do), vs completion of the Pilot Runtime. One of the big problems in all of this remains the availability of a working D0 as soon as possible.