Inter-Office Memorandum

To	Pilot Users	Date	September 12, 1979
From	Paul McJones	Location	Palo Alto
Subject	Pilot 3.0e Release - How To Use	Organization	SDD/SD

XEROX

Filed on: <WPilot>HowToUse.memo, .press

This memo describes how to use the Pilot 3.0e internal release. A short summary of the various releases of Pilot since Oak (2.0) may benefit some readers:

3.0a: maintenance release on 2.0 (Mesa 4; no public defs modules recompiled; bug fixes, some performance improvements, internal restructuring)

3.0a': conversion to Mesa 5; Pilot volume format changed

3.0b: Pilot microcode diverges from Alto/Mesa compatibility; microcode swapping to get to debugger (which still runs on compatible microcode); XDebug replaced by CoPilot; RunPilot added

3.0c: Nova emulator deleted from Pilot microcode; Pilot boot files used both as output of StartPilot and to save state on way to CoPilot (i.e. Swatee)

3.0d: Communications added; Pilot volume format changed.

3.0e: Bug fixes; preparations for running CoPilot on Pilot; Pilot volume format changed.

Pilot 3.0a' and later runs only on D0's with hardware updates of 8G or later.

Where to get it

Unless otherwise mentioned, all files mentioned in this memo as being part of the release should be obtained from the *release directory*. The release directory is specified for each particular version of Pilot in the accompanying release message. In this memo we will simply speak of the (Pilot) release directory or the microcode release directory; note that for external releases (n.0) they will be the same. The source of other files is mentioned explicitly in the text.

Compiling

Obtain the Pilot definitions from the release directory. (The command file **PilotInterfaces.cm** will fetch the .bcd's of all the public definitions modules; the file **PilotInterfaces.press** *is* the concatenation of these definitions modules.) Development Common Software (DCS) definitions may be obtained from the Mesa 5.0 release directory (**<Mesa>System>**) except for Magic, which is found on the release directory.

PilotInterfaces.press was not constructed for Pilot 3.0e; except for FileTypes.mesa it is the same as for Pilot 3.0d.

Always compile with the "-Alto/c" switch. For example:

```
Compile -Alto/c Mumble
```

The binder checks for intermixed -Alto/c and Alto/c code. (The interim compiler switch "Long/c" should *not* be used; it was used in an earlier phase of Pilot 3 development.)

Binding

Create and bind a configuration containing Pilot, DCS (optional), and your client or test module(s). In order to remain compatible with **Pilot.signals** and **DCS.signals**, you should make Pilot and DCS be the first two entries respectively in this configuration. One of your modules must export the procedure **Run** defined in the interface **PilotClient**. It is the first procedure called outside of Pilot and will cause a start trap in the exporting module. As such the first non-Pilot code to run is the initialization code of the module containing **PilotClient.Run** (or of its control module, as the case may be).

An experimental signal lister which takes as input a .bcd file (instead of a .image file) was used to produce the .signals files for this release; the program itself will be available in the future.

Setting up a disk for running on the D0

It is worthwhile to have a disk reserved for running on the D0; there are enough special programs needed that there probably won't be enough space left to do program development on the same disk. The files needed are listed in the table below. The first column, source, indicates the directory from which the file should be obtained. Two of the files must be named differently on your disk than on the release directory; that is indicated in the third column. A command file, **RunPilot.cm**, exists (on the Pilot release directory) to fetch these files and install CoPilot (see below).

source	*name*	*rename as*	*comment*
Microcode	AltoBoot.bt		disk-bootable Alto compatible microcode
<Alto>	MoveToKeys.run		delete after using--see below
Microcode	DMesa.sb	Pilot.sb	soft-bootable Pilot microcode
<Mesa>Temp>	RunMesa.run		runs image files
Pilot	CoPilot.image		Pilot debugger
<Mesa>	Binder.bcd		(optional; allows final binding on this disk
Pilot	StartPilot.image		builds Pilot boot files
Pilot	RunPilot.bcd		loads Pilot.sb and starts Pilot.germ
Pilot	GermAlto31.germ	Pilot.germ	loads Pilot boot file

"Pilot" and "Microcode" refer to the release directories--see "Where to get it" above.

{There is a patched version of RunMesa.run on <Mesa>Temp which avoids the tendency of the unpatched one to try to go to Swat.}

In addition your disk will have a Pilot boot file, together with the constituent bcd's and code, plus any source and symbols necessary in debugging. If necessary, some of the above can be deleted part way through a given debugging session in order to have more room for source and symbols.

Making your disk D0 bootable

In order to make your vanilla Alto disk D0 bootable you must obtain **AltoBoot.bt** from the microcode release directory and **<Alto>MoveToKeys.run** and then run the latter by typing

```
MoveToKeys AltoBoot.bt 56e
```

to the Alto Exec. MoveToKeys.run may then be deleted from your disk. It is not necessary to rerun it (although it does not hurt) if the same or a newer copy of AltoBoot.bt is fetched on top of the existing one. However, it will be necessary to rerun MoveToKeys.run if AltoBoot.bt is deleted completely from your disk and then re-fetched. (MoveToKeys must ordinarily run on an Alto since it is hard to boot the D0 until it has been done.)

Further, MoveToKeys.run is actually a self-contained Mesa program, and must be run on a D0 of the correct hardware version.

CoPilot: the Pilot debugger

As mentioned above, versions of Pilot starting with 3.0b require a special version of the Mesa debugger, called CoPilot. CoPilot behaves essentially identically like XDebug, so there is no new documentation on it other than these notes (in particular, the Fetch released for XDebug works with CoPilot). CoPilot should be obtained from the release directory and installed (by typing CoPilot to the Alto Exec) before creating a client image file. When installed, CoPilot extends Swatee to about 800 pages (the exact size depends on the amount of real memory on the D0; reinstall CoPilot if the amount of real memory changes).

CoPilot will NOT run on an Alto, even to install itself.

Beware of programs such as Empress which reset Swatee to 257 pages; the next time Pilot swaps to CoPilot you will get a DNP code of 905 (see below).

Creating a Pilot boot file

Obtain StartPilot.image from the release directory. If your outermost configuration is ClientOnPilot.config, then the image building command is:

```
StartPilot.image Build["ClientOnPilot"]
```

The result will be **ClientOnPilot.boot** and **ClientOnPilot.loadmap**. This boot file is specific to the disk upon which it is created. Do not try to copy it to another disk. Do not run it except on a D0.

The old DontRun[] command is gone.

The reason the boot file cannot be moved is that it has built into it FP's (Alto-style file identifiers) for Swatee and MesaDebugger, which in general vary from disk to disk.

{Implementors can save a little more disk space by running **StartPilot.bcd** (rather than .image); one must also have **Pilot.bootmesa**, and incant: StartPilot Pilot Build["Mumble"].}

Running the boot file

Load your D0 bootable disk into drive zero (i.e., DP0) of an 8G D0. Put a Pilot volume or a scratch disk into drive one. Any disk that has headers (has run on an Alto) can be used as a Pilot volume. New disks, right out of the box, cannot be used until headers are placed on them. Pilot will detect non-Pilot volumes and proceed to turn them into Pilot volumes.

Note: Between Pilot 3.0e and 3.0d (as well as changing from Pilot 3.0c to 3.0d, and 3.0a to 3.0a'), Pilot volumes underwent an incompatible change; older volumes must be rebuilt to run under the new system.

Press the red boot button on the D0. A number representing the current version of the microcode (140 for AltoBoot.bt) should appear in the Diagnostic Notification Panel (DNP) (three-digit LED display on the front panel). The boot should then proceed as a normal Alto boot and you'll come up in Alto emulation mode.

The program **RunPilot.bcd** must be used to load the Pilot microcode and start the Pilot germ (boot loader). To invoke RunPilot, type:

```
RunPilot ClientOnPilot // if you have Mesa.image on your disk
```
or
```
StartPilot Load["RunPilot"] ClientOnPilot // if you don't
```

RunPilot performs the following three actions:

1. Load the Pilot germ into memory from the file **Pilot.germ**
2. Reload the control store from the file **Pilot.sb**.
3. Start executing the germ.

When a block cursor appears in the middle of the screen, hit the space bar (while holding any desired key switches; see below); Pilot will begin initialization. (The DNP displays 000 while Pilot awaits key switches.)

It takes 30 seconds or more for Pilot to initialize itself and start your program. To indicate progress and help with trouble-shooting, Pilot displays a series of codes in the DNP. These codes are:

900 - Boot file being loaded by germ
910 - Mesa runtime being initialized
920 - Storage drivers being initialized
930 - Filer being initialized
940 - Swapper being initialized
950 - FileMgr being initialized
960 - VMMgr being initialized
970 - Client and other non-boot loaded code being mapped
980 - Temporary files being deleted (if T key was down)
990 - Communication being initialized
000 - PilotClient.Run called

If your Pilot volume has never been a Pilot volume before, or it was interrupted during a file operation the last time you ran, you will end up immediately in the debugger with an explanatory message (initializing a volume asks for two confirmations, first for the physical volume and second for the logical volume). This is normal and you should Proceed. (In the former case Pilot will initialize the volume, a process which takes an extra couple of minutes. In the latter case it will scavenge the volume, the amount of time required depending on the number of files and how fragmented the volume is.)

If you call **System.PowerOff** as the last thing in your program (and you should), then you will terminate normally with the word OFF in the cursor and the Pilot volume closed cleanly. To quit from the debugger use Kill Session (K). In all cases you may continue only by rebooting; there is no way to return from Pilot to the Alto emulator mode.

Pilot key switches

These keys affect various aspects of Pilot initialization. Because of the hardware and microcode, keys depressed before the microcode is loaded are lost. Thus Pilot gives the user a chance to enter key switches by displaying a block cursor centered in the screen and then waiting. To cause Pilot to continue depress the space bar WHILE HOLDING DOWN ALL DESIRED KEYS.

- E Erase physical volume
- L erase Logical volume
- S Scavenge logical volume
- M disable Map logging (see subsequent section on virtual memory)
- 2 call CoPilot just before **PilotClient.Run** is called (clients use this rather than 0 or 1)

The following are intended for Pilot implementors, but some clients may find them useful:

- F call CoPilot at very beginning of FileMgr
- V call CoPilot before initiating the VMMgr
- 1 call CoPilot after noninitially resident code is mapped
- 0 call CoPilot as soon as possible

Interpreting D0 DNP codes

If the microcode encounters certain problems during D0 booting or operation, it stops executing Mesa instructions and displays an error code in the maintenance panel. As mentioned above, 140 means the last boot was of Alto compatible microcode and 130 means the last boot was DMesa microcode. (During microcode booting a series of other numbers may be displayed.) The document *???* lists all standard error codes, but occasionally the machine stops with an undocumented error code.

Several values one is likely to see are:

003 - (during push-button boot) drive zero is not ready (if it looks ready, try spinning it down and back up again)
??? - (while running) an I/O device controller has touched a page of virtual memory not currently contained in real memory.

Additionally, Pilot and the germ display DNP codes when it is not possible to get to the debugger to report an error:

90X - Germ problem
- 901 - Alloc trap (bug)
- 902 - Control trap (bug)
- 903 - Start fault (bug)
- 904 - page or write protect fault (bug)
- 905 - device error (e.g. hard disk error, Swatee too short)
- 906 - Bad boot file (e.g. wrong version of StartPilot)
- 909 - Unnamed ERROR (bug)

91X - Pilot initialization problem
- 912 - Trap or KFCB before Mesa runtime initialized (bug)
- 919 - Attempt to swap to CoPilot before Mesa runtime initialized (bug)

{Implementors see the defs module PilotDNP for the actual definitions.}

Virtual memory and the Mesa debugger

In order for the Mesa debugger to access the Pilot virtual memory (e.g. to display a variable or to set a breakpoint), it must be aware of the current mappings between virtual address space and file windows. It does this by consulting the *virtual memory map log*, written by Pilot as a part of executing `Space.Map`, `Space.Remap`, and `Space.Unmap` operations.

Whenever the debugger is entered from the Pilot world, it first reads all new virtual memory map log entries, using the information to record in its internal data structures the current state of the Pilot virtual memory-file window mapping. This can take up to seven seconds depending on how many log entries it must read.

Pilot writes the virtual memory map log into a fixed-size area of its own virtual memory, and this area can fill up. When this happens Pilot automatically calls the debugger using a special entry point which displays the message `*** Processing VM Map ***`, processes the map log, and then returns to Pilot without checking for user input.

As mentioned before, virtual memory map logging can be disabled by holding down the `M` key during Pilot initialization. If you do this and then have to use the debugger, you will find that it responds to certain commands with the message `Command not allowed.`

Address and write protect faults

In contrast with systems on the Alto, Pilot and the D0 permit programs to access only those locations in virtual memory which are contained within *mapped spaces*. Furthermore, a space in virtual memory is read-only if the file window to which it is mapped is immutable or read-only. Programs which try to reference unmapped locations in virtual memory or which try to store into read-only locations will enter the debugger with the indications, `AddressFault` or `WriteProtectFault`, respectively. These are *not* signals and neither Pilot nor the client program can be usefully restarted. They are, instead, to be regarded as fatal errors. The following sample debugger script illustrates how to track down one of these faults.

The following scenario needs to be redone to show BytePC's.

Also, obtaining Swapper.symbols and VMMgr.symbols from the release directory would simplify things; when displaying the stack, one can look for the variable "page".

```
CoPilot 0.1 of 12-Jul-79 16:12
12-Jul-79 18:58

AddressFault
>-- Start by finding out where you are; you should be somewhere in SpaceImpl.
>Display Stack
No symbols for SpaceImpl, G: 164174B, L: 147664B, PC: 3061B,E >n
No symbols for SpaceImpl, G: 164174B, L: 147724B, PC: 3024B,O >n
No previous frame!
>-- The page fault actually happened in another process.
>-- Find out who is running.
>List Processes [confirm]
PSB: 2636B, waiting CV, No symbols for PupRouter, G: 156460B, L: 143130B, PC: 1117B,O
PSB: 2643B, waiting CV, No symbols for OisRouter, G: 157414B, L: 147740B, PC: 662B,O
PSB: 2650B, waiting CV, No symbols for Processes, G: 163424B, L: 144004B, PC: 256B,E
PSB: 2655B, waiting CV, No symbols for AltoEthernetDriver, G: 157754B, L: 143210B, PC: 1015B,O
PSB: 2662B, waiting CV, No symbols for AltoEthernetDriver, G: 157754B, L: 143550B, PC: 222B,O
PSB: 2667B, waiting CV, No symbols for Dispatcher, G: 160040B, L: 143014B, PC: 215B,O
PSB: 2674B, waiting CV, No symbols for StatsHot, G: 160270B, L: 143334B, PC: 136B,E
PSB: 2701B, No symbols for Traps, G: 163000B, L: 150060B, PC: 1235B,E
PSB: 2706B, waiting CV, No symbols for Model31CSPImpl, G: 155600B, L: 143320B, PC: 333B,E
PSB: 2713B, waiting CV, No symbols for CacheMissImpl, G: 172660B, L: 143024B, PC: 76B,O
PSB: 2720B, waiting CV, No symbols for MStoreImpl, G: 164634B, L: 147654B, PC: 254B,E
PSB: 2725B, waiting CV, No symbols for PageFaultImpl, G: 164444B, L: 143240B, PC: 173B,O
PSB: 2732B, waiting CV, No symbols for TransferImpl, G: 167470B, L: 150164B, PC: 454B,E
```

```
PSB: 2737B, waiting CV, No symbols for ServerImpl, G: 170520B, L: 143044B, PC: 232B,O
PSB: 2744B, waiting CV, No symbols for Model31CSPImpl, G: 155600B, L: 143064B, PC: 164B,O
PSB: 2751B, waiting CV, No symbols for SystemImpl, G: 161140B, L: 143074B, PC: 230B,O
PSB: 2756B, waiting CV, No symbols for InterruptKey, G: 163570B, L: 143104B, PC: 41B,O
PSB: 2770B, No symbols for PageFaultImpl, G: 164444B, L: 150034B, PC: 145B,O
>CUrrent context --
 Module: SpaceImpl, G: 164174B, L: 147664B, PSB: 2701B
 Configuration: VMMgr
>-- Look for a process which is running (not waiting CV)
>-- and is not the current PSB (2701B, in Traps).
>-- The offender should be in PageFaultImpl.
>SEt Process context: 2770B
>Display Stack
No symbols for PageFaultImpl, G: 164444B, L: 150034B, PC: 145B,O >n
PageFaultTest, L: 143034B (in PageFaultTest, G:150214B)   >v
 w = 0
 i = 47B
 p = 23400B↑
   >s
Source:  <>FOR i IN [0..256) DO
   >
>-- The offending page number is the fourth word of the local frame for PageFaultImpl ( 150034B).
>-- The offending program is indicated by the next frame on the stack (e.g., PageFaultTest)
>Octal Read @: 150034B, n(10): 4
150034B/ 164444B 177633B 143034B   47B
>-- I.e., it was page 47B in virtual memory
>Octal Read @: 143034B, n(10): 4
143034B/ 150214B 177765B 143710B 133020B
>-- The program counter for the offending program is in the second word of this local frame.
>--              (i.e., 177765B)
>Kill session [confirm]
```

Notes: Because the debugger doesn't really understand page faults, it may display the source line preceeding the offending one. To nail things down exactly, do two things:

1. Find the page that it faulted on; it is in the fourth word of the local frame of PageFaultImpl's trap handler (at 150037B in the example). In our case, it is page 47B, the value of POINTER **p** of our program divided by 256 (the size of a page).

2. If you're still not sure what's going on, find the PC (program counter) in the second word of the frame which caused the trap (143035B↑ = 177765B). Its absolute value (13B) is the word offset in the code; positive means the even byte, negative means the odd one. Now get a code listing (see the *Mesa User's Handbook*), and find the PC (13,Odd). E.g.,

```
PageFaultTest:  PROGRAM
   IMPORTS  DisplayDefs,  System
   EXPORTS  PilotClient  =
   BEGIN


            Frame  size:   7

   p:  LONG  POINTER  ←  NIL;


          7,E    16: [054]        LI0
          7,O    17: [154]        DUP
         10,E    20: [027]        SGDB              3

   FOR  i  IN  [0..256)  DO

          11,E   22: [054]        LI0
          11,O   23:              [161]        J3                    (26)
          12,E   24: [150]        PUSH
          12,O   25:              [267]        INC
          13,E   26: [024]        SG2

      w  ←  p↑;
```

```
            13,O   27:              [243]       RIGPL          [0,0]
                        -- This instruction is the guilty one.

            14,O   31:              [025]       SG3

    p ← p + 256;
    ENDLOOP;

END.
            15,E   32: [021]        LGDB              3
            16,E   34: [065]        LIW*            400
            20,E   40: [054]        LI0
            20,O   41:              [274]       DADD
            21,E   42: [027]        SGDB              3
            22,E   44: [012]        LG2
            22,O   45:              [064]       LIB                377
            23,O   47:              [000]       NOOP
            24,E   50: [220]        JULB            353  (24)
            25,E   52: [343]        RET

        Instructions:  20,  Bytes:  30
```