Inter-Office Memorandum

To	Distribution	Date	October 26, 1980
From	Roy Levin, Richard Johnsson	Location	Palo Alto
Subject	On the Harmonious Cooperation of Mesa and User Microcode	Organization	PARC/CSL, OPD/SDD

XEROX

Filed on: <Mesa>Doc>Microcode.bravo, .press

This document describes an interface that permits users to load their own microcode into the RAM of an Alto II XM and execute it without interfering with the Mesa emulator. These facilities are available only on Alto II XM's with XMesa PROMs or with 3K RAM boards. When running on an Alto with the 3K RAM option, the following substitutions should be made in the remainder of this memo: RAM1 replaces ROM1. RAM0 replaces RAM.

Loading User Microcode

The Mesa emulator occupies all of ROM1 and about 143 words of RAM. The remainder of the RAM is available for use by client programs. To simplify loading of user microcode, a file containing the overflow (RAM) portion of the Mesa emulator is included in the user's assembly, producing a single output file that can be loaded in its entirety. Addressing requirements are specified in the file.

The microcode normally loaded in the RAM by RunMesa includes some optional extensions so that many clients who loaded private microcode in the past may no longer need to do so. The included functions are: Pup Checksum, IEEE floating point, and HBLT (used by Griffin). These microcode packages use up all of the remaining RAM so that clients desiring additional microcode will have to eliminate one or more of these extensions.

Assembly Procedure

Users should carefully follow the steps below in constructing RAM images to be loaded under Mesa.

1) Obtain XMesaOverflow.mu, XMesaRAM.mu, and Mesab.mu from the dump file <Mesa>System>MesaMu.dm. (The dump file contains other files which should be ignored.) The first two files are the actual microcode, the third contains the definitions for all registers and other symbols used by the Mesa emulator. If you want to retain any of the standard extensions you must get them from the dump file as well. The file names are Float.mu, HBlt.mu and Checksum.mu.

2) Rename XMesaOverflow.mu to be Driver.mu and add to the end a line of the form:

```
#UserMicrocode.mu;                    user microcode source
```

3) Delete one or more of the optional included files. If you delete Float.mu you must either uncomment the line ahead of it or provide your own implementation of the MISC instruction.

4) Assemble Driver.mu with MU, obtaining Driver.mb.

5) Use the PackMU subsystem to convert Driver.mb to Driver.br.

Driver.mu contains dummy instructions which occupy the first 20B locations of the RAM. If the user microcode includes the code to control a non-standard I/O device (e.g., a Trident disk), at least one of these instructions will be replaced by an instruction used as part of the silent boot sequence (see Alto Hardware Manual, sections 2.4, 8.4, and 9.2.2). Prudence suggests that the locations corresponding to tasks that are not intended to execute in the RAM befilled in with dummy instructions of the form

```
Taski:      TASK, :Taski;
```

Thus, if Driver.mu contains no device control microcode, Task0-Task17 should all have this form. If there is a space crunch, locations 0-17B may be used for ordinary microcode, though this is usually unnecessary.

Loading Procedure

Driver.br can now be loaded using the (Mesa) RamLoad package. The facilities of this package are defined in RamDefs.bcd, and are exported by the module RamLoad.bcd. These files may all be found on <Mesa>Utilities>. RamDefs.mesa contains adequate documentation for the use of this package; the following Mesa code illustrates a typical use.

```
BEGIN OPEN RamDefs;
driver: MuImage _ ReadPackedMuFile["Driver.br"];
IF LoadRamAndBoot[driver,FALSE] # 0 THEN SIGNAL BogusMicrocode;
ReleaseMuImage[driver];
END;
```

The second parameter to **LoadRamAndBoot** controls whether a silent boot is performed and should be **FALSE** unless it is necessary to alter the set of running tasks (as is required when additional device control microcode is initialized). **LoadRamAndBoot** returns the number of mismatches between the constants specified in the microcode file and those present in the Alto's constants ROM. If this number is non-zero, the microcode cannot execute properly on this machine.

Note to the nervous: Since RamLoad is a Mesa program, it must exercise caution to avoid smashing the emulator on which it is running. As long as the procedure above is meticulously followed in constructing Driver.br, RamLoad will be able to load the RAM without committing suicide.

There and Back Again

This section describes mechanisms for transferring control between the Mesa emulator and user-supplied microcode. It assumes that this microcode is, in effect, implementing "extended instructions", and the linkage mechanisms described are suitable for this purpose. Device driver microcode (*e.g.*, for a Trident disk) executes in a different hardware task and therefore does not require these linkages.

The hardware-defined linkage mechanism between control memory banks is somewhat precarious. Opportunities for errors arise because of the way the "next address" field of the microinstruction is interpreted when SWMODE has been invoked (see section 8.4 of the Alto Hardware Manual). The thing to remember is that whether you are in ROM1 or RAM, to get to the other memory you must specify an address with the 400 bit on (*i.e.*, **BITAND[address,400B] = 400B**). If you preserve this invariant, your Alto will probably avoid most black holes. Additional address restrictions are in effect on 3K RAM Altos; see *Alto Hardware Manual* section 8.4.

The Mesa MISC Instruction

Mesa has a bytecode, MISC, that is used to extend the Mesa bytecode set with special purpose instructions. MISC takes an a-byte which specifies the operation to be performed. Some a values already have defined meaning. These are recorded in <Mesa>System>MiscAlpha.mesa. To avoid conflict with "standard" extensions, *e.g.* floating point, users implementing strictly private functions should use a values larger than 200B. The a value 11B is filtered out by ROM1 and returns the Alto Real Time Clock. All other values are passed to microcode in the RAM. The MISC instruction operates this way only on version 41 microcode. In version 39 (released with Mesa 5.0) the MISC instruction always reads the clock regardless of the a value.

The normal RAM microcode for MISC simply zeros the stack pointer and returns. The instruction labeled MISC: in Driver.mu may be replaced by code to decode the a value. When control reaches MISC, the a value is in T. Control is returned to ROM1 by the sequence

```
SWMODE;                 sigh...SWMODE and TASK are both F1s
:romnext;               (... and we can't TASK here)
```

romnext is defined in Mesab.mu, as are all the registers and other symbols used by the Mesa emulator.

Setting Up a MISC in Mesa

The easiest way to access microcode in the RAM is by declaring a procedure to invoke it as follows:

```
alphaValue: CARDINAL = 100B;

MyCode: PROCEDURE[arg1: AType, arg2: AnotherType]
        RETURNS [result: AThirdType] =
        MACHINE CODE BEGIN
        Mopcodes.zMISC, alphaValue
        END;
```

When **MyCode** is invoked, the arguments are pushed on the evaluation stack. Control will then wind up at the location MISC in the RAM with alphaValue in T. The microcode is responsible for computing **result** and placing it on the stack (details appear below). The return sequence described above will then cause execution to resume immediately after the invocation of **MyCode**.

The Mesa JRAM Instruction

Note: The MISC bytecode provides the function for which JRAM was originally intended. While JRAM is retained for compatibility, its use is discouraged.

Mesa also has a bytecode, JRAM, that is a straightforward mapping of the Alto JMPRAM instruction. JRAM takes the top element off the stack and dispatches on it, doing a SWMODE in the same instruction. The microcode therefore looks approximately like this:

```
JRAM:   IR_sr17, :Pop;              pops stack into L, T; returns to JRAMr
JRAMr:  SINK_M, BUS, SWMODE;
        L_0, :zero;
```

Thus, at entry to whatever microcode JRAM jumps to, L=0 and T has the target address of the jump.

Getting back to ROM1 is equally straightforward. User microcode should terminate with:

```
SWMODE;                 sigh...SWMODE and TASK are both F1s
:romnextA;              (... and we can't TASK here)
```

romnextA is defined in MesabROM.mu, as are all the registers and other symbols used by the Mesa emulator.

The safest way to force a microinstruction to a specific address in the RAM is to use MU's '%' pre-definition facility. For example,

```
%1,1777,540,MyCode;

MyCode:  . . . ;            start of user-written microcode
```

would enable the Mesa procedure **MyCode** (defined just below) to transfer control to the microcode at MyCode.

Setting Up a JRAM in Mesa

The easiest way to access microcode in the RAM is by declaring a procedure to invoke it as follows:

```
locationInRAM: CARDINAL = 540B;

MyCode: PROCEDURE[arg1: AType, arg2: AnotherType]
        RETURNS [result: AThirdType] =
        MACHINE CODE BEGIN
        Mopcodes.zLIW, locationInRAM/256, locationInRAM MOD 256;
        Mopcodes.zJRAM
        END;
```

When **MyCode** is invoked, the arguments are pushed on the evaluation stack, then a transfer to location 540B in the RAM occurs. The microcode is responsible for computing **result** and placing it on the stack (details appear below). The return sequence described above will then cause execution to resume immediately after the invocation of **MyCode**.

Note: A few locations in the RAM are already used as entry points from the emulator microcode in ROM1. Although no firm convention has been established, user microcode that avoids RAM addresses in the ranges 400B-477B and 600B-677B is unlikely to experience compatibility problems in the future.

The Mesa Stack

The Mesa stack is implemented by 8 S-registers named stk0, stk1, ..., stk7, with stk0 being the base of the stack. An R-register, stkp, indicates the number of words on the stack and is thus in the range [0..8]. To obtain values from the stack in the general case, therefore, requires a dispatch on stkp, but there is an important special case. If **MyCode** is invoked in a statement context (*i.e.*, one in which it stands alone and is not a term of an expression), the stack will be empty except for **arg1** and **arg2**, which will therefore appear in `stk0` and `stk1`, respectively. In this case `stkp` will be 2 at entry, and the microcode should set it to 1 before returning to ROM1. **result** should also be stored in stk0. It is frequently useful to restrict the use of **MyCode** to statement contexts so that the microcode can take advantage of the known stack depth. Note: this assumes that values of type **AType**, **AnotherType**, and **AThirdType** are each represented in a single word. Multi-word quantities are stored in adjacent stack words, with consecutive memory cells being pushed in increasing address order.

An argument or return record may not exceed 5 words in length. If you need more arguments you should pass a pointer. We believe that it is relatively easy to allow arguments up to 6 words. It may be possible to allow up to 8 words in the future.

Other Emulator State

The Mesa emulator has a number of temporary registers that may be freely used by code entered via MISC or JRAM. The following list identifies all registers used by the Mesa emulator, their intended function, and whether they may be used as destroyable temporaries by user-supplied RAM microcode:

Register	Type	Destroyable by RAM code	Function
mpc	R	No	program counter
stkp	R	No	stack pointer
XTSreg	R	No	Xfer trap status
ib	R	No	instruction byte
brkbyte	R	No	Xfer state variable; overlaps AC3
clockreg	R	No	high resolution clock bits
mx	R	Yes	temporary during Xfer; overlaps AC2
saveret	R	Yes	holds return dispatch values; overlaps AC1
newfield	R	Yes	used by field instructions; overlaps AC0
count	R	Yes	temporary for counting
taskhole	R	Yes	temporary for holding things across TASKs
temp	R	Yes	general temporary
temp2	R	Yes	general temporary
lp	S	No	local frame pointer (+6)
gp	S	No	global frame pointer (+4)
cp	S	No	code segment pointer
stk0-7	S	No	8 evaluation stack registers
wdc	S	No	wakeup disable counter (interrupt control)
ATPreg	S	Yes	alloc trap parameter
XTPreg	S	Yes	xfer trap parameter
OTPreg	S	Yes	other trap parameter
mask	S	Yes	used by field instructions; smashed by BITBLT
index	S	Yes	used by field instructions; smashed by BITBLT
alpha	S	Yes	temporary for alpha byte; smashed by BITBLT
entry	S	Yes	Xfer and field temporary; smashed by BITBLT
frame	S	Yes	ALLOC/FREE temporary; smashed by BITBLT
my	S	Yes	Xfer temporary; smashed by BITBLT
unused1	S	Yes	smashed by BITBLT
unused2	S	Yes	smashed by BITBLT
unused3	S	Yes	smashed by BITBLT

User microcode should endeavor to TASK within 5 microcycles of entry. It should also TASK as close to the end as possible. Ideally, the penultimate instruction before returning to the emulator would TASK, but unfortunately SWMODE must appear in the same instruction and both are F1s. The Mesa emulator tries to TASK at least every 12 microcycles; user microcode should observe the same guideline.

The Mesa emulator does not check for pending interrupts on every instruction. It does so only when it must fetch a new instruction word from memory. Therefore, user microcode that sets a pending interrupt condition must not expect that the interrupt will be noticed by the emulator

immediately upon return to ROM1.

Distribution:
Mesa Users
Cedar Group
Mesa Group