5.0 MISCELLANEOUS PERIPHERALS

The Alto can have a number of slow peripherals which appear to programs as memory locations in the range 177000-177777B. The standard peripherals are described here.

5.1 Keyboard

The Alto keyboard contains 61 or 64 keys. It appears to the program as four 16 bit words in 4 adjacent locations starting at KBDAD (177034B). Depressed keys correspond to zeroes in memory, idle keys correspond to ones. Figure 6 shows layouts of the Microswitch and ADL keyboards, including keytops and the word number, bit number corresponding to each key. All Alto Is and the more recent Alto IIs have Microswitch keyboards; earlier Alto IIs have ADL keyboards, which are somewhat larger and have columns of function keys on the left and right sides.

MICROSWITCH KEYBOARD

Bit	KBDAD (177034B)	KBDAD+1 (177035B)	KBDAD+2 (177036B)	KBDAD+3 (177037B)
0	5	3	1	R
1	4	2	ESC	T
2	6	W	TAB	G
3	E	Q	F	Y
4	7	S	CTRL	H
5	D	A	C	8
6	U	9	J	N
7	V	I	B	M
8	0 (zero)	X	Z	LOCK
9	K	O	<shift-left>	SPACE
10	-	L	. (period)	[
11	P	, (comma)	;	+
12	/	" (quote)	RETURN	<shift-right>
13	\	]	←	<blank-bottom>
14	LF	<blank-middle>	DEL	xxx
15	BS	<blank-top>	xxx	xxx

ADL KEYBOARD

Bit	KBDAD (177034B)	KBDAD+1 (177035B)	KBDAD+2 (177036B)	KBDAD+3 (177037B)
0	5	3	1	R
1	4	2	ESC	T
2	6	W	TAB	G
3	E	Q	F	Y
4	7	S	CTRL	H
5	D	A	C	8
6	U	9	J	N
7	V	I	B	M
8	0 (zero)	X	Z	LOCK
9	K	O	<shift-left>	SPACE
10	-	L	. (period)	[
11	P	, (comma)	;	+
12	/	" (quote)	RETURN	<shift-right>
13	\ (FR2)	]	← (FR3)	FR1
14	LF (FL2)	FR4	DEL (FL1)	FL4
15	BS	BW	FL3	FR5

FL stands for the function keys at the left of the keyboard; FR for those at the right.

Figure 6

Note: Connecting an Alto I keyboard to an Alto II or an Alto II Microswitch keyboard to an Alto I requires rewiring a connector or installing an adaptor cable. An ADL keyboard requires additional logic to connect to an Alto I.

5.2 Mouse

The mouse is a hand-held pointing device which contains two encoders which digitize its position as it is rolled over a table-top. It also has three buttons which may be read as the three low-order bits of memory location UTILIN (177030B), in the manner of the keyboard. The bit/button correspondences in UTILIN are (depressed keys correspond to 0's in memory):

UTILIN[13]	Top or Left Button (RED)
UTILIN[14]	Bottom or Right Button (BLUE)
UTILIN[15]	Middle Button (YELLOW)

The mouse coordinates are maintained by the MRT microcode in locations MOUSELOC(424B)=X and MOUSELOC+1(425B)=Y in page one of the Alto memory. These coordinates are relative, i.e., the hardware only increments and decrements them. The resolution of the mouse is approximately 100 points per inch.

5.3 Keyset

The standard Alto includes a five-finger keyset which is presented to the program as 5 bits of memory location UTILIN (177030B), similar to the keyboard. The bit/key correspondences in UTILIN are (depressed keys correspond to 0's in memory):

UTILIN[8]	Key 0 (left-most)
UTILIN[9]	Key 1
UTILIN[10]	Key 2
UTILIN[11]	Key 3
UTILIN[12]	Key 4 (right-most)

5.4 External Device Interface

Two memory locations, UTILIN (177030B) and UTILOUT (177016B), provide an interface to external devices through a connector on the rear of the Alto. If a quantity is stored into UTILOUT, it is latched and appears as 16 output signals; if a 1 bit is stored, a more negative logic level is generated (TTL "low"). For input, bits 0 to 5 and bit 7 of UTILIN are available; more positive logic levels (TTL "high") are reported as 1 bits. The remaining bits of this location are used by the mouse, keyset and memory configuration switch.

On the Alto I, this connector also provides various power supply voltages. These are absent on Alto II.

The Alto II provides an additional 16-bit input port (the X bus), which can be read by accessing memory locations 177020B-177023B. The connector on the rear of the Alto II provides the low 2 bits of memory address and a signal that indicates the X bus is being read, together with the 16 input data signals. More positive logic levels (TTL "high") are reported as 1 bits.

The two sections below describe two common devices connected to UTILIN/UTILOUT, the Diablo HyType printer and Versatec printer/plotter. The descriptions are for the programmer: the bit values (0 or 1) refer to values that will be stored into UTILOUT or read from UTILIN by an Alto program.

5.4.1 Diablo Printer

The Diablo HyType printer plugs into a connector on the rear of the Alto, and is controlled by referencing two locations in Alto memory. None of the timing signals required by the printer are generated automatically--all must be program generated. For detailed information on the printer, refer to the Diablo manual.

Location UTILIN (177030B):

UTILIN[0] Paper ready bit. 0 when the printer is ready for a paper scrolling operation.

UTILIN[1] Printer check bit. Should the printer find itself in an abnormal state, it sets this bit to 0.

UTILIN[2] Unused.

UTILIN[3] Daisy ready bit. 0 when the printer is ready to print a character.

UTILIN[4] Carriage ready bit. 0 when the printer is ready for horizontal positioning.

UTILIN[5] Ready bit. Both this bit and the appropriate other ready bit (carriage, daisy, etc.) must be 0 before attempting any output operation.

UTILIN[6] (Memory configuration switch -- see section 5.5)

UTILIN[7] Unused.

Location UTILOUT (177016B):

Several of the output operations are invoked by "toggling" a bit in the output status word. To toggle a bit, set it first to 1, then back to 0 immediately.

UTILOUT[0] Paper strobe bit. Toggling this bit causes a paper scrolling operation.

UTILOUT[1] Restore bit. Toggling this bit resets the printer (including clearing the "check" condition if present) and moves the carriage to the left margin.

UTILOUT[2] Ribbon bit. When this bit is 1 the ribbon is up (in printing position); when 0, it is down.

UTILOUT[3] Daisy strobe bit. Toggling this bit causes a character to be printed.

UTILOUT[4] Carriage strobe bit. Toggling this bit causes a horizontal positioning operation.

UTILOUT[5-15] Argument to various output operations:

1. Printing characters. When the daisy bit is toggled bits 9-15 of this field are interpreted as an ASCII character code to be printed (it should be noted that all codes less than 40B print as lower case "w").
2. For paper and carriage operations the field is interpreted as a displacement (-1024 to +1023), in units of 1/48 inch for paper and 1/60 inch for carriage. Positive is down or to the right, negative up or to the left. The value is represented as sign-magnitude (i.e., bit 5 is 1 for negative numbers, 0 for positive; bits 6-15 are the absolute value of the number).

The printer is initialized by toggling the restore bit, then waiting for all ready bits to be 0. A typical output sequence, say printing a character, involves examining the check bit for abnormal status, waiting for both the ready and daisy ready bits to be 0, then writing in the printer output location the character code, the character code ORed with the daisy strobe bit, and the unmodified code again.

The device behaves more or less like a plotter, i.e. you must explicitly position each character in software; a print operation does not affect the position of either the carriage or the paper. All coordinates

in paper or carriage operations are relative; the device does not know its absolute position. Again, you must keep track of this in software.

WARNING: On Alto I, the printer cable should not be changed (connected or disconnected) while Alto power is on. The printer power is derived from the Alto power supplies; changing the cable causes a large transient which usually crashes the processor and does bad things to the disk drive. On Alto II, the printer is independently powered and may therefore be connected or disconnected at any time.

5.4.2 Versatec Plotters and Printer/Plotters

Because of their delightfully simple hardware interface, all manner of Versatec equipment may be driven from the Alto with ease. The description below gives the signal assignments and a small number of coding tricks; the programmer should consult a Versatec manual for details (bulletin 6002, Matrix Basic Interface Description is particularly helpful). The notation * is used below to indicate a signal whose sense is inverted.

Location UTILIN (177030B):

UTILIN[1]	ONLINE*	On-line (inverted).
UTILIN[2]	NOPAP	No paper.
UTILIN[3]	READY*	Ready (inverted).

Location UTILOUT (177016B):

UTILOUT[0]	RFFED	Remote form feed.
UTILOUT[1]	CLEAR	Clear print line.
UTILOUT[2]	RLTER	Remote line terminate.
UTILOUT[3]	PICLK*	Print clock (inverted).
UTILOUT[4]	PRINT*	Print select (inverted) -- print=0, plot=1
UTILOUT[5]	SPP	Simultaneous print/plot.
UTILOUT[6]	RESET	Remote reset.
UTILOUT[7]	REOTR	Remote end of transmission.
UTILOUT[8-15]	IN08* to IN01*	Data bits to be sent to the Versatec (inverted). Bit 8 is the most significant bit of the nibble; bit 15 is the least significant.

None of the timing signals (PICLK) are generated automatically by the Alto--the programmer must cause the signals to wave appropriately. The Alto II DIAGNOSE2 instruction is particularly helpful for generating the clock signals. The control functions (RFFED, CLEAR, RLTER, RESET, REOTR) are generated by raising and then lowering them:

```
LDA 0 FORMFEED
LDA 1 FORMTOGGLE
LDA 3 UTILOUTADR
DIAGNOSE2

FORMFEED:    114000     ; RFFED + PICLK* + PRINT*
FORMTOGGLE:  100000     ; RFFED
UTILOUTADR:  177016
```

Data bytes must be sent with care, because the UTILOUT data lines take a little time to set up. The data is first set, then the clock bit is toggled, and then the clock bit is toggled again:

```
LDA 0 DATA
COM 0 0                 ; Note that data must be inverted.
LDA 1 DATAMASK
AND 1 0                 ; Save IN08*-IN01*,PICLK*,PRINT*. We're plotting
LDA 3 UTILOUTADR
STA 0 0 3               ; Let data settle--clock is "off"
LDA 1 DATATOGGLE
DIAGNOSE2               ; Toggle clock "on" then "off"

DATA:         111       ; ASCII code for "I"
DATAMASK:     014377    ; PICLK* + PRINT* + data mask
DATATOGGLE:   010000    ; PICLK*
UTILOUTADR:   177016
```

On Alto I, DIAGNOSE2 is not available, but its effect may be emulated.

5.5 *Parity Error Detection*

The detection and reporting of parity errors is accomplished somewhat differently on Alto I and Alto II. In both machines, the processing of errors is undertaken by a high-priority microtask, which is invoked very soon after an error occurs. The microtask reports a parity error by causing an interrupt on emulator interrupt channel 15, i.e., by ORing a one into NWW[15]. Bear in mind that parity errors can be generated by memory references undertaken by any microtask; as a result, it may be some time between the occurrence of the error and the next execution of the emulator task and consequent servicing of the interrupt.

When a parity error happens, the parity task stores the contents of various R registers into some page 1 reserved locations given below. Unfortunately, the information recorded by the parity task is not sufficient to determine precisely where the parity error occurred. The intent of the collection is to save values of the R registers most likely to be used as a source of memory addresses.

Address	R-Register	Use
614B	DCBR	Disk control block fetch pointer
615B	KNMAR	Disk word fetch/store pointer
616B	DWA	Display word fetch address
617B	CBA	Display control block fetch address
620B	PC	Current program counter in the emulator
621B	SAD	Temporary register for indirection in emulator

Alto II

The Alto II memory contains circuitry for correcting single-bit errors and detecting double-bit errors. The logic expects a good deal of set-up and in turn reports copious error information. Interaction with the error control is effected through three memory locations (177024B, 177025B and 177026B). Detailed information on the operation of the error correction mechanism is best obtained from the logic drawings.

Memory Error Address Register (MEAR = 177024B). This register is a 'shadow MAR': it holds the address of the first error since the error status was last reset. If no error has occurred, MEAR reports the address of the most recent memory access. Note that MEAR is set whenever an error of *any kind* (single-bit or double-bit) is detected.

Memory Error Status Register (MESR = 177025B). This register reports specifics of the first error that occurred since MESR was last reset. Storing anything into this register resets the error logic and enables it to detect a new error. Bits are "low true," i.e. if the bit is 0, the condition is true.

MESR[0-5]	Hamming code reported from error
MESR[6]	Parity Error
MESR[7]	Memory parity bit
MESR[8-13]	Syndrome bits
MESR[14-15]	Bank number in which error occurred

MESR[14-15] is an extension to the most significant end of MEAR. This field is only present if the extended memory option is installed (see section 2.3), otherwise it reads out -1.

Memory Error Control Register (MECR = 177026B). Storing into this register is the means for controlling the memory error logic. This register is set to all ones (disable all interrupts) when the Alto is bootstrapped and when the parity error task first detects an error. When an error has occurred, MEAR and MESR should be read before setting the MECR. Bits are "low true," i.e. a 0 bit enables the condition.

MECR[0-3]	Spare
MECR[4-10]	Test Hamming code (used only for special diagnostics)
MECR[11]	Test mode (used only for special diagnostics)
MECR[12]	Cause interrupt on single-bit errors if zero
MECR[13]	Cause interrupt on double-bit errors if zero
MECR[14]	Do not use error correction if zero
MECR[15]	Spare

Note that MECR[12] and [13] govern only the initiation of interrupts; MEAR and MESR hold information about the first error that occurs after reseting MESR regardless of what kind of errors are to cause interrupts.

ADDRESS MAPPING

The mapping of addresses to memory chips can be altered by the setting of the "memory configuration switch." This switch is located on the front of Alto I's, and at the top of the backplane of the Alto II. The current setting of the switch is reported in bit 6 of UTILIN (location 177030B): if this bit is 0, the switch is in the "normal" position ("up" on Alto I, "back" on Alto II), otherwise the switch is in the "alternate" position. On Alto I, if the switch is in the alternate position, the first two 16K portions of memory are exchanged (i.e., the memory address is modified by the algorithm: if MAR[0]=0 then MAR[1]←MAR[1] XOR 1). On Alto II, if the switch is in the alternate position, the first and second 32K portions of memory are exchanged (i.e., the memory address is modified by the algorithm: MAR[0]←MAR[0] XOR 1).

In order to fix many memory problems, it is necessary to know the mapping between memory addresses (and bit numbers) to actual memory chips on the memory boards. Herewith the mapping, given in the style of a program: the algorithm is given the memory address (*address*) and the bit position in the word (*bit*). The function odd(x) returns true if the 16-bit number x is odd. The variable *switch* corresponds to the setting of the memory configuration switch (i.e., switch←UTILIN[6]).

Alto I

The variables *row* and *column* are the "coordinates" of the memory chip on the given *cardSlot*, as printed by the memory diagnostic. The *chipNumber* is the chip number on the memory board. Bit 16 is the parity bit.

```
if address[0]=0 then (if switch=1 then address[1]←address[1] xor 1)
row←address[2-4]
cardSlot←(address[0-1])*4 + 13
if odd(address) then card←card+2
column←bit
if bit ≥ 12 then [ card←card+1; column←bit-5 ]
```

```
chipNumber← 15 + column + 14*row
```

Alto II

The Alto II memory system is organized around 32-bit doublewords. Stored along with each double word is 6 bits of hamming code and a parity bit for a total of 39 bits:

bits 0-15	even data word
bits 16-31	odd data word
bits 32-37	Hamming code
bit 38	parity bit

Things are further complicated by the fact that two types of memory chips are used: 16K chips in machines with the extended memory option (see section 2.3), and 4K chips for all others.

The bits in a 1-chip deep slice of memory are called a *group*. A group contains 4K or 16K double words, depending on chip type. The bits of a group on a single board are called a *subgroup*. Thus a subgroup contains 10 of the 40 bits in a group. There are 8 subgroups on a memory board. Subgroups are numbered from the high 3 bits of the address: for 4K chips this means MAR[0-2]; for 16K chips (i.e. an Alto with extended memory) this means BANK.MAR[0]:

Subgroup	chip positions	
7	81-90	
6	71-80	
5	61-70	
4	51-60	
3	41-50	
2	31-40	
1	21-30	
0	11-20	Nearest the edge connector

The location of the bits in group 0 is:

```
    CARD 1          CARD 2          CARD 3          CARD 4
32 24 16 08 00 | 33 25 17 09 01 | 34 26 18 10 02 | 35 27 19 11 03
36 28 20 12 04 | 37 29 21 13 05 | 38 30 22 14 06 | XX 31 23 15 07
               ↑                ↑                ↑                ↑
          chip position 11
```

Chips 15, 25, 35, 45, 55, 65, 75, and 85 on board 4 aren't used. If you are out of replacement memory chips, you can use one of these, but then the board with the missing chips will only work in Slot 4.

The algorithm for converting *address* and *bit* into *cardSlot* and *chipNumber* is (the variable 'xm' is true if the Alto has extended memory):

```
    if odd(address) then bit←bit+16
α:  if switch=1 then address[0]←address[0] xor 1
    cardSlot← (bit mod 4) +1
    chipNumber← bit/8 + 16 - (if odd(bit/4) then 5 else 0) +
        10 * (if xm then address[0] else address[0-2]) +
        (if xm then bank*20 else 0)
```

A second entry to this algorithm is with an *address* (usually read from MEAR), and a *syndrome* (usually read from MESR, but remember that it must be complemented: syndrome←(rv(MESR))[8-13] XOR 77B)).

```
bit←syndromeMapping[syndrome]      (see table below)
if bit=-1 then error ("impossible" syndrome)
enter the algorithm above at α.
```

The syndromeMapping maps a 6-bit number (range 0 to 63) into the number of the bad bit (0 to 38) or -1 if the syndrome is incorrect:

	0	1	2	3	4	5	6	7	
0	38	37	36	-1	35	-1	18	-1	(syndrome values 0 to 7)
10	34	29	14	-1	7	-1	22	-1	
20	33	27	12	-1	5	-1	20	-1	
30	2	31	16	-1	9	-1	24	-1	
40	32	26	11	-1	4	-1	19	-1	
50	1	30	15	-1	8	-1	23	-1	
60	0	28	13	-1	6	-1	21	-1	
70	3	-1	17	-1	10	-1	25	-1	

6.0 DISK AND CONTROLLER

The disk controller is designed to accommodate one of a variety of DIABLO disk drives, including models 31 and 44. Each drive accommodates one or two disks. Each disk has two heads, one per side. Information is recorded on each disk in a 12-sector format on each of up to 406 (depending on the disk model) radial track positions. Thus, each disk contains up to 9744 recording positions (2 heads x 12 sectors x 406 track positions). Figure 7 tabulates various useful information about the performance of the disk drives.

DEVICE	DIABLO 31	DIABLO 44	
Number of drives/Alto	1 or 2	1	
Number of packs	1 removable	1 removable 1 fixed	
Number of cylinders	203	406	
Tracks/cylinder/pack	2	2	
Sectors per track	12	12	
Words per sector	2 header 8 label 256 data	2 header 8 label 256 data	
Data words/track	3072	3072	
Sectors/pack	4872	9744	
Rotation time	40	25	ms
Seek time (approx.)	15+8.6*sqrt(dt)	8+3*sqrt(dt)	ms *
min-avg-max	15-70-135	8-30-68	ms
Average access to 1 megabyte	80	32 (using both packs)	ms
Transfer rate:			
peak/avg	1.6/1.22	2.5/1.9	MHz
peak/avg	10.2/13	6.7/8.	μs/word
per sector	3.3	2.1	ms
for full display	.460	.266	sec
for 64k memory	1.03	.6	sec
whole drive	19.3	44(both packs)	sec

* The notation dt stands for the number of tracks traveled during the seek.

Figure 7

The disk controller records three independent data blocks in each sector. The first is two words long, and is intended to include the address of the sector. This block is called the *Header block.* The second block is eight words long, and is called the *Label block.* The third block is 256 words long, and is the *Data block.* Each block may be independently read, written, or checked, except that *writing, once begun, must continue until the end of the sector.*

When a block is checked, information on the disk is compared word for word with a specified block of main memory. During checking, a main memory word containing 0 has special significance. When this word is encountered, the matching word read from the disk is stored in its place and does not take part in the check. This feature permits a combination of reading and checking to occur in the same block. (It also has the drawback of making it impossible to use the disk controller to check for words containing 0 on the disk.)

The Alto program communicates with the disk controller via a four-word block of main memory beginning at location KBLK (521B). The first word is interpreted as a pointer to a chain of disk command blocks. If it contains 0, the disk controller will remain idle. Otherwise, the disk controller will commence execution of the command contained in the first disk command block. When a command is completed successfully, the disk controller stores in KBLK a pointer to the next command in the chain and the cycle repeats. If a command terminates in error, a 0 is immediately stored in KBLK and the disk

controller idles. At the beginning of each sector, status information, including the number of the current sector, is stored in KBLK+1. This can be used by the Alto program to sense the readiness of the disk and to schedule disk transfers, for example. When the disk controller begins executing a command, it stores the disk address of that command in KBLK+2. This information is later used by the disk controller to decide whether seek operations or disk switches are necessary. It can be used by the Alto program for scheduling disk arm motion. If the Alto program stores an illegal disk address (like -1) in this word, the disk controller will perform a seek at the beginning of the next disk operation. (This is useful, for example, when a disk driver wants to force a restore operation.) The disk controller also communicates with the Alto program by interrupts (see Section 3.2). At the beginning of each sector interrupts are initiated on the channels specified by the bits in KBLK+3.

KBLK (521B): Pointer to first disk command block.
KBLK+1 (522B): Status at beginning of current sector.
KBLK+2 (523B): Disk address of most-recently started disk command.
KBLK+3 (524B): Sector interrupt bit mask.

A disk command block is a ten-word block of memory which describes a disk transfer operation to the disk controller, and which is also used by the controller to record the status of that operation. The first word is a pointer to the next disk command block in this chain. A 0 means that this is the last disk command block in the chain. When the command is complete, the disk controller stores its status in the second word. The third word contains the command itself, telling the disk controller what to do. The fourth word contains a pointer to the block of memory from/to which the header block will be transferred. The fifth word contains a similar pointer for the label block. The sixth word contains a similar pointer for the data block.

The seventh and eighth words of the disk command block control the initiation of interrupts when the command block is finished. If the command terminates without error, interrupts are initiated on the channels specified by the bits in DCB+6. However, if the command terminates with an error, the bits in DCB+7 are used instead.

The ninth word is unused by the disk controller, and may be used by the Alto program to facilitate chained disk operations. The tenth word contains the disk address at which the current operation is to take place.

DCB: Pointer to next command block.
DCB+1: Status.
DCB+2: Command.
DCB+3: Header block pointer.
DCB+4: Label block pointer.
DCB+5: Data pointer.
DCB+6: Command complete no-error interrupt bit mask.
DCB+7: Command complete error interrupt bit mask.
DCB+8: Currently unused.
DCB+9: Disk address.

A disk address word A contains the following fields:

FIELD	RANGE	SIGNIFICANCE
A[0-3]	0-13B	Sector number.
A[4-12]	0-625B (Model 44) 0-312B (Model 31)	Cylinder number.
A[13]	0-1	Head number.
A[14]	0-1	Disk number (see also C[15]). 0 is removable pack on Model 44. 1 is optional second Model 31 drive.

A[15]	0-1	0 normally. 1 if cylinder 0 is to be addressed via a hardware "restore" operation.

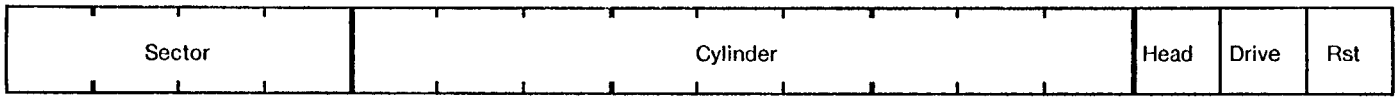
A disk command word C contains the following fields:

FIELD	RANGE	SIGNIFICANCE
C[0-7]	110B	Checked to verify that this is a valid disk command.
C[8-9]	0-3	0 if Header block to be read. 1 if Header block to be checked. 2 or 3 if Header block to be written.
C[10-11]	0-3	0 if Label block to be read. 1 if Label block to be checked. 2 or 3 if Label block to be written.
C[12-13]	0-3	0 if Data block to be read. 1 if Data block to be checked. 2 or 3 if Data block to be written.
C[14]	0-1	0 normally. 1 if the command is to terminate immediately after the correct cylinder position is reached (before any data is transferred).
C[15]	0-1	XOR'ed with A[14] to yield hardware disk number.

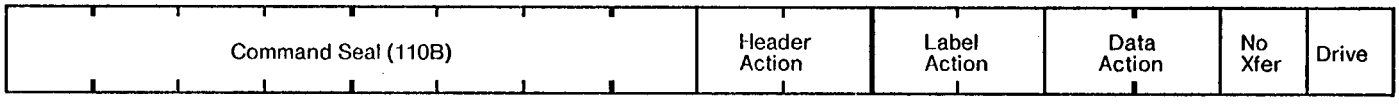
A disk status word S has the following fields:

FIELD	VALUES	SIGNIFICANCE
S[0-3]	0-13B	Current sector number.
S[4-7]	17B	One can tell whether status has been stored by setting this field initially to 0 and then checking for non-zero.
S[8]	0-1	1 means seek failed, possibly due to illegal cylinder address.
S[9]	0-1	1 means seek in progress.
S[10]	0-1	1 means disk unit not ready.
S[11]	0-1	1 means data or sector processing was late during the last sector. Data and current sector number unreliable.
S[12]	0-1	1 means disk interface was not transferring data last sector.
S[13]	0-1	1 means checksum error. Command allowed to proceed.
S[14-15]	0-3	0 means command completed correctly. 1 means hardware error (see S[8-11]) or sector overflow. 2 means check error. Command terminated instantly. 3 means disk command specified illegal sector.

Several clever programming tricks have been suggested to drive the disk controller. For an initial program load, KBLK should be set to point to a disk command block representing a read into location

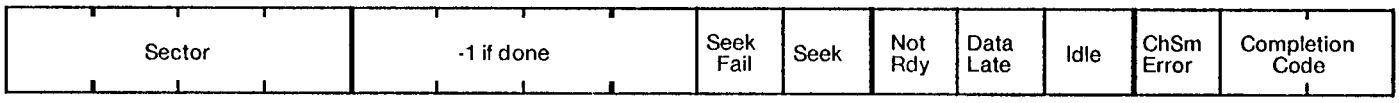


Disk Address



Disk Command

0: Read
1: Check
2 or 3: Write



Disk Status

0: Good status
1: Hardware error
2: Check error
3: Illegal sector

0	Pointer to next KCB
1	Disk status
2	Disk command
3	Header record memory address
4	Label record memory address
5	Data record memory address
6	No-Error Interrupt bit mask
7	Error Interrupt bit mask
10	Reserved
11	Disk address

Disk Command Block (KCB)

521	Pointer to next KCB
522	Status at beginning of sector
523	Disk address of most recent KCB
524	Sector interrupt bit mask

Reserved Page 1 Locations

Figure 8 -- Disk Data Structures

STRT. Before setting KBLK, the Alto program should put a JMP STRT instruction in STRT; afterward it should jump to STRT. The disk controller transfers data *downward*, from high to low addresses, so that when location STRT is changed the reading of the block is complete. (See section 3.4 on the standard bootstrap loading microcode.)

Another trick is to chain disk reads through their label blocks. That is, the label block for sector n contains part of the disk command block for reading sector n+1, and so on.

6.1 Disk Controller Implementation

The following walk-through of an average day in the life of the standard disk controller is not intended for the casual reader, but rather as a roadmap to ease the pain of learning the innermost workings of the controller. If you really want to benefit from this next section, you should have a copy of the standard disk controller microcode and logic drawings close at hand.

The disk controller consists of a modest amount of hardware and two microcode tasks (the sector task and the word task). Communication with the emulator is via the four special main memory words, the disk command blocks, and the interrupts described earlier. In following few paragraphs the actions of the standard disk controller microcode are described. Occasionally it may be unclear whether the actor is microcode or hardware. Referring to microcode listings and/or logic drawings will resolve any such questions.

The sector task is awakened by a sector signal from the disk. When awakened, it stores the status of the disk and controller in the special disk status word (KBLK+1). In addition, if this sector signal terminates a disk command (for example, a data transfer during the previous sector), the status of the disk and controller are stored in the status word of the disk command block containing the terminated command, and the command block pointer (KBLK) is advanced. If a command was terminated with an error, KBLK (DCB pointer) is set to 0 and KBLK+2 (current disk address) is set to -1. The effect of this is to cause the disk controller to abandon the current disk command chain and to forget where the disk arm is positioned.

Next, the sector task considers the first command on the disk command block chain (by using KBLK). If there is none, or if the disk unit is not ready to accept a command, the sector task goes to sleep until the next sector pulse. Otherwise, the sector specified in the new command is verified to be less than 13. Then, the disk and cylinder specified in the new command are compared with those stored in KBLK+2 (current disk address), and then the new disk address is stored in KBLK+2 and in the disk controller hardware. Part of the new command is also stored in the hardware. If the comparison is unequal, a seek is initiated and the sector task goes to sleep until the next sector pulse.

If the comparison was equal, the SEEKOK hardware flag is tested. If that is OK, then the no-transfer bit of the disk command (bit 14 of the command word of the current disk command block) is tested to see whether a data transfer is required. If not, the sector task goes to sleep such that the command will terminate at the next sector pulse. If a data transfer is required, the specified sector number and the current disk sector number are compared. If unequal, the sector task goes to sleep until the next sector pulse. If sector numbers are equal, awakening of the word task is enabled and the sector task goes to sleep such that the command will terminate at the next sector pulse.

The word task awakens when a word has been processed by the disk controller hardware and the word task has been enabled by the sector task. First, a starting delay is computed, based on whether the current record is to be read or written. Second, control is dispatched based on the current record number. A record length and main memory starting address are computed based on the record number. In addition, special starting delays are computed for record number 0. The disk unit is set into the delay mode appropriate for the operation (read/write) and the word task goes to sleep the appropriate number of times.

Then a sync word is written (if writing) or awaited (if reading). Finally the main transfer loop is entered. Here the word count is decremented, a memory operation is started, and control is dispatched on the transfer type. If read, the disk word is stored in memory. If write, the memory word is sent to the disk. If check, the memory word is compared with 0. If non-zero, the disk and memory words are compared. An unequal compare here terminates this sector's operation with an error immediately. If the memory word is 0, it is replaced by the disk word. In any case, the checksum is updated and control returns to the main transfer loop. Due to the ALU functions available, the main transfer loop moves in sequence from high to low main memory addresses.

After the word count reaches 0, the checksum is written or checked. A checksum error will be noted in the status word, but will not terminate this sector's operation. A finishing delay is computed, based on the current operation, the disk unit is set into a delay mode appropriate to the operation, and the delay happens. Finally, all disk transfers are shut off, the record number is incremented, and control returns to the beginning of the word task.

To accomplish all this, the disk controller hardware communicates with the microprocessor in four ways: first, by task wakeup signals for the sector and word tasks; second, by five task-specific F2's which modify the next microinstruction address; third, by seven task-specific F1's, four of which activate bus destination registers, and the remaining three of which provide useful pulses; and fourth, by two task-specific BS's. The following tables describe the effects of these.

F1 VALUE	NAME	EFFECT
17B	KDATA←	The KDATA register is loaded from BUS[0-15]. This register is the data output register to the disk, and is also used to hold the disk address during KADR← and seek commands. When used as a disk address it has the format of word A in section 6.0 above.
16B	KADR←	This causes the KADR register to be loaded from BUS[8-14]. This register has the format of word C in section 6.0 above. In addition, it causes the head address bit to be loaded from KDATA[13].
15B	KCOM←	This causes the KCOM register to be loaded from BUS[1-5]. The KCOM register has the following interpretation: (1) XFEROFF = 1 inhibits data transmission to/from the disk. (2) WDINHIB = 1 prevents the disk word task from awakening. (3) BCLKSRC = 1 takes bit clock from disk input or crystal clock, as appropriate. BCLKSRC = 1 forces use of crystal clock. (4) WFFO = 0 holds the disk bit counter at -1 until a 1-bit is read. WFFO = 1 allows the bit counter to proceed normally. (5) SENDADR = 1 causes KDATA[4-12] and KDATA[15] to be transmitted to disk unit as track address. SENDADR = 0 Inhibits such transmission.
14B	CLRSTAT	Causes all error latches in disk controller hardware to reset, clears KSTAT[13].
13B	INCRECNO	Advances the shift registers holding the KADR register so that they present the number and read/write/check status of the next record to the hardware.
12B	KSTAT←	KSTAT[12-15] are loaded from BUS[12-15]. (Actually, BUS[13] is ORed into KSTAT[13].) This enables the microcode to enter conditions it detects into the status register.

11B	STROBE	Initiates a disk seek operation. The KDATA register must have been loaded previously, and the SENDADR bit of the KCOMM register previously set to 1.
F2 VALUE	NAME	EFFECT
10B	INIT	NEXT←NEXT OR (*if* WDTASKACT AND WDINIT) *then* 37B *else* 0)
11B	RWC	NEXT←NEXT OR (*if* current record to be written *then* 3 *elseif* current record to be checked *then* 2 *else* 0)
12B	RECNO	NEXT←NEXT OR MAP (current record number) where MAP(0) = 0 MAP(1) = 2 MAP(2) = 3 MAP(3) = 1
13B	XFRDAT	NEXT←NEXT OR (*if* current command wants data transfer *then* 1 *else* 0)
14B	SWRNRDY	NEXT←NEXT OR (*if* disk not ready to accept command *then* 1 *else* 0)
15B	NFER	NEXT←NEXT OR (*if* fatal error in latches *then* 0 *else* 1).
16B	STROBON	NEXT←NEXT OR (*if* seek strobe still on *then* 1 *else* 0).
BS VALUE	NAME	EFFECT
3	←KSTAT	The KSTAT register is placed on BUS. It has the format of a disk status word.
4	←KDATA	The disk input data register is placed on BUS.

A feature of interest mostly to the diagnostic microcode writer is that if one reads the disk input data register while writing, what should appear is delayed written data correctly aligned on word boundaries. This is a painless way of checking most of the data paths in the disk controller hardware.

7.0 ETHERNET

An Ethernet is the principal means of communications between an Alto and the outside world. The object was to design a communication system which could grow smoothly to accommodate several buildings full of personal computers and the facilities needed for their support. The Ethernet is a broadcast, multi-drop, packet-switching, bit serial, digital communications network: it connects up to 256 nodes, separated by as much as 1 kilometer, with a 2.94 megabits/sec channel. Control of the Ethernet is distributed among the communicating computers to eliminate the reliability problems of an active central controller, to avoid a bottleneck in a system rich in parallelism, and to reduce the fixed costs which make small systems uneconomical.

The Ethernet is intended to be an efficient, low-level packet transport mechanism which gives its best efforts to delivering packets, but *it is not error free.* Even when transmitted without source-detected interference, a packet may not reach its destination without error; thus, *packets are delivered only with high probability.* Stations requiring a residual error rate lower than that provided by this bare packet transport mechanism must follow mutually agreed upon packet protocols.

Alto Ethernets come in three pieces: the transceiver, the interface, and the microcode. The transceiver is a small device which taps into the passing Ether, inserting and extracting bits under the control of the interface while disturbing the Ether as little as possible. The same device is used to connect all types of Ethernet interfaces to the Ether, so the transceiver design is not specific to the Alto, and will not be described here. The following sections describe the programming characteristics of the Alto Ethernet, and then the implementations of the interface and microprogram.

7.1 Programming Characteristics

Programs communicate with the interface and the microcode via the emulator instruction SIO and 9 reserved locations in page 1. Word counts, buffer addresses, etc., are put in the appropriate locations and then SIO is executed with an Ethernet command in AC0.

The special page 1 memory locations and their functions are:

EPLOC = 600B: Post location. Microcode and interface status information is posted in this location when a command completes.

EBLOC = 601B: Interrupt bit location. The contents of this location is ORed into NWW when a command completes, thereby causing interrupt(s) on the channels corresponding to the one bits in EBLOC.

EELOC = 602B: End count location. The number of words remaining in the main memory buffer at command completion is stored here as part of the posting operation.

ELLOC = 603B: Load location. This location is used by the microcode to hold a mask of ones shifted in from the right for generating random retransmission intervals. ELLOC should be zeroed before starting the transmitter.

EICLOC = 604B: Input count location. The emulator program should put the size of the input buffer (in words) into this location before starting the receiver. If a packet arrives that is longer than EICLOC, the receiver will post an Input Buffer Overrun error status.

EIPLOC = 605B: Input pointer location. The emulator program should put a pointer to the beginning of the input buffer into this location before starting the receiver.

EOCLOC = 606B: Output count location. The emulator program should put the size of the output buffer (in words) into this location before starting the transmitter. By convention, packets should not be substantially longer than 256 words.

EOPLOC = 607B: Output pointer location. The emulator program should put a pointer to the beginning of the output buffer into this location before starting the transmitter.

EHLOC = 610B: Host address location. This location must contain zero in the left byte and the host address in the right byte. The microcode matches this host address against the first byte of a passing packet to decide whether to accept it.

SIO passes commands to the interface and returns the host address of the Alto. Commands to the Ethernet interface are encoded in the two low order bits of AC0 and have the following meaning (the remaining bits of AC0 may be interpreted by other devices and thus should be zero):

AC0[14-15]: 0 Do nothing
1 Start the transmitter
2 Start the receiver
3 Reset the interface and microcode.

The host address, returned in AC0[8-15] by SIO, is set by wires on the Alto backpanel. This number is normally put in EHLOC thereby causing packets with destination addresses matching the address set with the wires to be accepted by the receiver. For more on addressing, see below.

Upon completion of a command, EPLOC contains the status of the microcode in the left byte and the status of the interface in the right byte. The possible values of the microcode status byte, EPLOC[0-7], and their meanings are:

EPLOC[0-7] = 0: Input done. If the hardware status byte is 377B, the interface believes the packet was received without error.

EPLOC[0-7] = 1: Output done. If the hardware status byte is 377B, the interface believes the packet was sent without error. The number of collisions experienced while sending the packet is $\log_2(\text{ELLOC}/2+1)-1$.

EPLOC[0-7] = 2: Input buffer overrun. The received packet was longer than the buffer, and the excess words were lost. Buffer overrun causes an early exit from the microcode input main loop, so it is likely that the CRC error and Incomplete transmission bits in the hardware status byte will be set.

EPLOC[0-7] = 3: Load overflow. The transmitter experienced 16 consecutive collisions (assuming ELLOC was zeroed before starting the transmitter) while trying to transmit the packet described by EOPLOC and EOCLOC. ELLOC[0] will be one.

EPLOC[0-7] = 4: The command (input or output) specified a zero length buffer.

EPLOC[0-7] = 5: Reset. Generally indicates that a reset command (SIO with AC0[14-15] = 3) was issued to the interface when it was idle or *any* command was issued when it was not idle.

EPLOC[0-7] = 6: Microcode branch conditions that should never happen cause this code to be posted if they do happen.

EPLOC[0-7] = 7-377B: The microcode does not generate these values for status.

Note that the microcode statuses are *small integers* and not individual bits as in the interface status byte. Bits in the interface status byte, EPLOC[8-15], are *low true*. When *zero*, their meanings are:

EPLOC[8-9]	Unused. These should always be one.
EPLOC[10]	Input data late. The interface did not get enough processor cycles.
EPLOC[11]	Collision.
EPLOC[12]	Input CRC bad.
EPLOC[13]	Input command issued. (AC0[14] in last SIO)
EPLOC[14]	Output command issued. (AC0[15] in last SIO)
EPLOC[15]	Incomplete transmission. The received packet did not end on a word boundary.

Command completion can be detected in two ways: (1) zero EPLOC and wait for it to go non-zero, or (2) set bits in EBLOC corresponding to the channels on which interrupts are desired at command completion.

When a program wishes to send a packet, it must first turn off the receiver if it is on. If the receiver is actively copying a packet into memory, the transmitter should wait for the receiver to finish (a maximum of about 1.5 ms. assuming 250-300 word packets). The program can tell whether the receiver is actively transferring or idle by zeroing the first word of the input buffer before starting the receiver. When the program wants to start the transmitter, it checks the first word of the input buffer: if it is still zero, input has not yet begun and the interface may be reset and the transmitter started with a high probability of not missing an incoming packet. There is still a small window between testing the word and starting the transmitter when a packet can arrive and be missed, but paragraph two of this chapter warned that the Ethernet is not error free anyway, so missing a few more packets should be harmless.

A program can determine the size of an input message (and though not too useful, the number of words transferred to the interface by the output microcode) by subtracting the contents of EELOC from the original buffer count in EICLOC or EOCLOC. The microcode never modifies the buffer count or pointer locations.

To keep the receiver listening as much of the time as possible, if EICLOC is non-zero when an output command is issued, the microcode will start the receiver 'under' the transmitter: while the transmitter is counting down a random retransmission interval after a collision, the receiver is listening. If a message arrives addressed to the receiver, the transmission attempt is aborted and the incoming message is received into the buffer described by EICLOC and EIPLOC. The transmit command is *not* executed in this case, and must be reissued. The microcode status byte in EPLOC will have an 'input done' status value if the transmission attempt was aborted by an incoming packet.

The first word of all Ethernet packets must contain the address to which the packet is destined in the left byte, and the address of the sender (or 'source') in the right byte. Receivers examine at least the destination byte, and in some cases (not in Altos) the source byte to determine whether to copy the message into memory as it passes by. Address zero has special meaning to the Ethernet. Packets with destination zero are *broadcast* packets, and all active receivers will receive them. If a program wishes to receive *all* packets on the Ether regardless of address (useful for debugging and diagnostic programs), it should put zero into EHLOC instead of the host number returned by SIO. A host which does this is said to be *promiscuous*. Address 377B is reserved for Ethernet booting (see section 3.4). Address 376B is reserved as the destination for diagnostic messages.

By convention, the second word of all Ethernet packets is the *packet type*. Communication protocols using the Ethernet should set the type word to describe the protocol to which the packet belongs (for example Pup protocol packets have 1000B in the type word). The type word is purely a software convention; no Ethernet hardware or microcode interprets it.

7.2 Ethernet Hardware

The Ethernet hardware consists of a FIFO buffer, an output shift register and phase encoder, a clock recovery circuit, an input shift register, a CRC register, and one microcode task. The hardware is shown in block diagram form in Figure 8. Packets on the Ether are phase encoded and transmitter synchronous: it is the responsibility of the receiver to decide where a packet begins (and thus establish the phase of the data clock), separate the clock from the data, and deserialize the incoming bit stream. The purpose of the write register is to synchronize data transfers between the input shift register whose clock is derived from the incoming data, and the FIFO which is synchronous to the processor system clock. The large FIFO is necessary because the Ethernet task has relatively low priority, and the worst case latency from request to task wakeup is on the order of 20 microseconds. The phase encoder uses the system clock (one Ethernet bit time is two clock periods).

Included in the clock recovery section is a one-shot which is retriggered by each level transition of a passing packet. This detects the envelope of a packet and is called its 'carrier'. Ethernet phase encoders mark the beginning of a packet by prefixing a single 1 bit, called the *sync* bit, to the front of all transmissions. The leading edge of the sync bit of a packet will trigger the carrier one-shot of a listening receiver and establish the receiver clock phase. The sync bit is clocked into the input shift register and recirculated every 16 bit times thereafter to mark the presence of a complete word in the register. If carrier drops without the sync bit at the end of the register, the transmission was incomplete, and is flagged in the hardware status bits. When the shift register is full, the word is transferred to the write register where it sits until the FIFO control has synchronized its presence and there is room to accept it. If the shift register fills up again before the word has been transferred from the write register to the FIFO, data has been lost and the input data late flip flop is set.

Ethernet transmitters accumulate a 16 bit cyclic redundancy checksum on the data as it is serialized, and append it to an outgoing packet after the last data word. As a receiver deserializes an incoming packet it recomputes the checksum over the data plus the appended CRC word. If the resulting receiver checksum is non-zero, the received packet is assumed to be in error, and the condition is flagged in the hardware status byte. Since the CRC is of no interest to the emulator program, a wakeup request to empty data from the FIFO is only made when it contains two or more words. This reduces the effective size of the FIFO by one word, but insures that the CRC will be left behind at the end of a packet.

The phase encoder is started when the microcode has decremented the countdown to zero, there is no carrier present, and either the FIFO is full, or if the message is less than 16 words long, all of it has been transferred to the FIFO. The phase encoder will *not* start up while there is carrier present. This means that collisions can only happen because of delay in sensing carrier between widely spaced transmitters. Collisions are detected at the transceiver by comparing the data the interface is supplying to the data being received off the Ether. If the two are not identical, a signal is returned to the interface which sets the collision flip flop causing a wakeup request to the microcode which resets the interface. Countdowns are accomplished by setting a flip flop from the microcode which will cause a wakeup request on the next occurrence of SWAKMRT. This makes the grain size of countdowns about 38 microseconds.

The interface and the transceiver are connected together by three twisted pairs for signals plus two supply voltages and ground supplied from the interface. The signals are (1) transmitted data to the transceiver, (2) received data from the transceiver, and (3) the collision signal from the transceiver indicating interference.

7.3 Ethernet Microcode

The Ethernet microcode uses a single task and 2 registers in R:

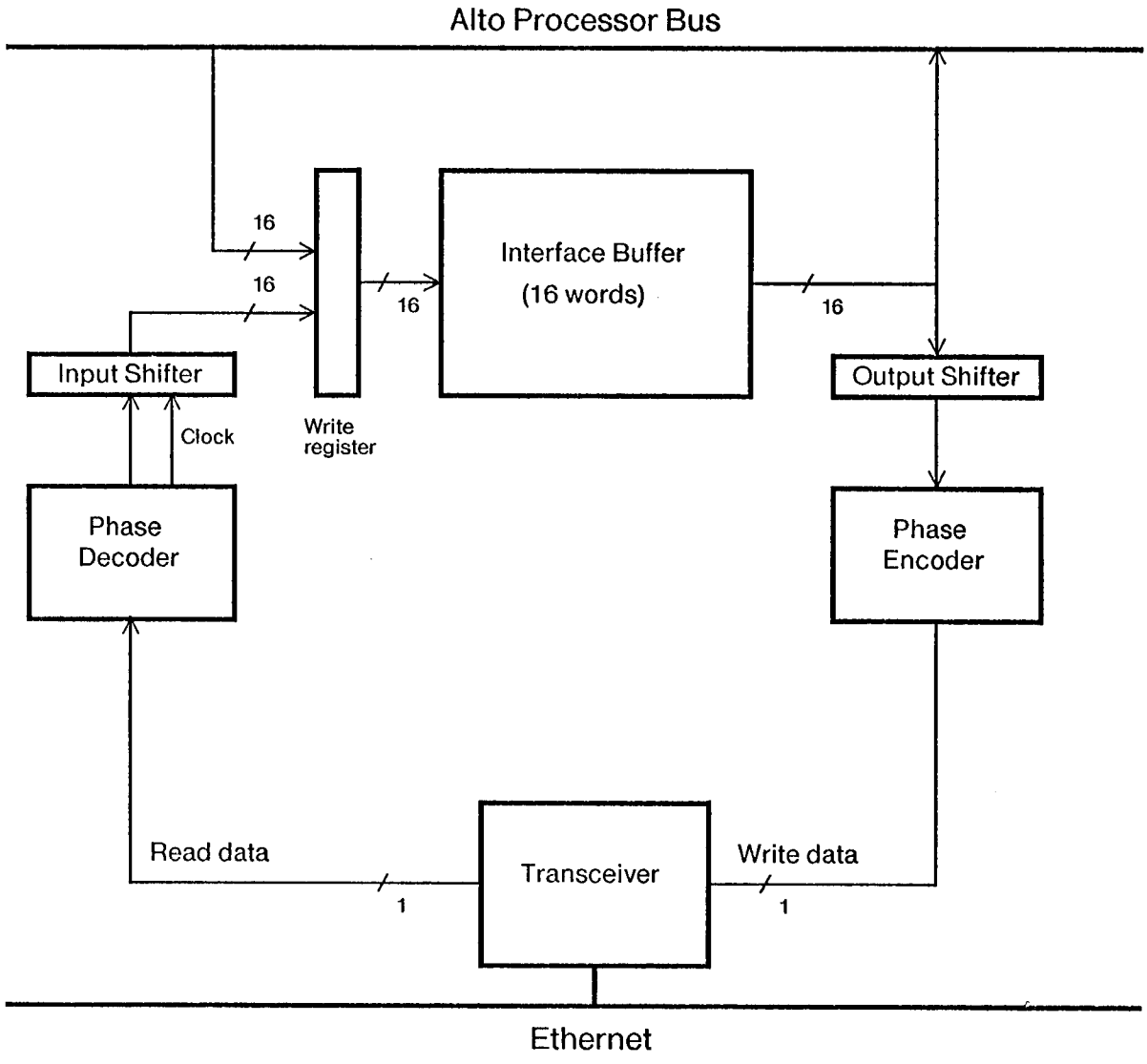


Figure 9 -- Ethernet Control

0: Normal input completion
1: Normal output completion
2: Input buffer overrun
3: Load overflow
4: Zero length buffer
5: Reset by software
6: Impossible microcode condition
7-377b: Reserved

ECNTR: The number of words remaining in the buffer.
EPNTR: Points at the word prior to that next to be processed.

The task and R registers are shared by input and output so that at any time they are (1) unused, (2) transmitting a packet, or (3) receiving a packet. When an Ethernet SIO is issued while the Ethernet microcode is reset, the code dispatches on whether it is an input, output, or reset command.

Each Ethernet SIO has a result which is posted when the command completes. The state of the microcode and hardware at the time of the post is deposited in EPLOC, the contents of ECNTR is deposited in EELOC, and the contents of EBLOC is ORed into NWW. Note that resetting the interface with EBLOC non-zero will result in an interrupt.

An input command (SIO with AC0[14:15] = 2) causes the microcode to start the input hardware searching for the start of a packet and then block. When a packet begins to arrive, the hardware wakes up the microcode which compares the packet's address against the filtering instructions left in EHLOC by the emulator program. The packet will be accepted if any of three conditions is true: (1) If EHLOC is zero, the receiver is said to be *promiscuous* - all packets are accepted; (2) if the destination address (left byte of the first word) of the packet is zero, the packet is *a broadcast* packet - all receivers accept broadcast packets; or (3) if the destination byte matches the right byte of EHLOC - the packet was sent to that specific host. If none of these conditions is met, the packet is rejected by restarting the receiver, which causes it to ignore the current packet and to hunt for the beginning of the next packet. If the packet is accepted, the microcode enters the input main loop.

The input main loop first loads ECNTR and EPNTR from EICLOC and EIPLOC. Note that EICLOC and EIPLOC are not read until the receiver is committed to transferring data to memory, which may be long after the receiver was started; therefore, these locations should not be disturbed while the receiver is on. The main loop repeatedly counts down the buffer size in ECNTR and advances the buffer pointer in EPNTR depositing packet words until either the hardware says that the packet has ended or the buffer overflows; in either case, the input operation terminates and posts.

An output command (SIO with AC0[14-15] = 1) causes the microcode to compute a random retransmission interval, wait that long, and then start transmitting the packet described by EOCLOC and EOPLOC. The retransmission interval is computed by ANDing the contents of ELLOC with the contents of R37, the low part of the real time clock (ELLOC is not modified). Then a one bit is left shifted into ELLOC and the high order bit of the result is tested. If the high order bit is on, the transmission attempt is aborted with a 'load overflow' microcode status. The above process is repeated each time the transmitter detects a collision while transmitting the packet. If ELLOC started out zero, each collision will double the value of ELLOC, thus doubling the mean of the random number generated by ANDing ELLOC with the real time clock. If 16 consecutive collisions occur without successfully transmitting the packet, the attempt is aborted.

The retransmission interval is decremented every 38.08 microseconds (the memory refresh task wakeup signal is used for this) until it reaches zero, at which time ECNTR and EPNTR are loaded from EOCLOC and EOPLOC and the transmitter part of the interface is started. This may occur long after the emulator program issued the output command, so EOCLOC and EOPLOC should not be changed while the transmitter is on. Note that the mean of the first retransmission interval will be zero, so the first transmission attempt will begin immediately. Actual transmission of the packet does not begin until the FIFO has been filled by the output main loop (or if the packet is smaller than the FIFO, until all of the packet is in the FIFO) and there is silence on the Ether. If EICLOC is non zero while the transmitter is counting down a retransmission interval, the receiver is turned on and if a packet arrives with an acceptable address, the transmission attempt is forgotten and the microcode enters the input main loop as if an input command had been issued.

The output main loop repeatedly counts down the packet length in ECNTR and advances the address in EPNTR taking words from the output buffer and putting them in the FIFO until either the main memory buffer is emptied or a hardware condition aborts the operation. The output main loop is awakened for a

data word once every 5.44 microseconds on the average. The microcode signals the hardware when the main memory buffer is empty and waits for the hardware to terminate; it then posts status.

A reset command (SIO with AC0[14-15] = 3) will *always* bring the interface back to a reset state. If the receiver was on, it is stopped even if a packet was pouring into memory. If the transmitter was on, it is stopped, even if it was in the middle of transmitting a packet (the result to the receiver of the interrupted packet will almost certainly be an incomplete transmission and incorrect CRC). Status will immediately be posted in EPLOC: the microcode will post the reset status (5) in the microcode status byte, and the hardware will post the conditions at the time of the reset in the hardware status byte. The contents of the ECNTR R register will be deposited in EELOC, and the contents of EBLOC will be ORed into NWW, possibly causing interrupts. After doing this, the interface and microcode are reset and ready for another command.

The task specific microcode functions for the Ethernet interface are summarized below.

EIDFCT *	BS=4	Input Data Function. Gates the contents of the FIFO to BUS[0-15], and increments the read pointer at the end of the cycle.
EILFCT *	F1=13B	Input Look Function. Gates the contents of the FIFO to BUS[0-15] but does not increment the read pointer.
EPFCT	F1=14B	Post Function. Gates interface status to BUS[8-15]. Resets the interface at the end of the cycle.
EWFCT	F1=15B	Countdown Wakeup Function. Sets a flip flop in the interface that will cause a wakeup to the Ether task on the next tick of SWAKMRT. This function must be issued in the instruction after a TASK. The resulting wakeup is cleared when the Ether task next runs.
EODFCT	F2=10B	Output Data Function. Loads the FIFO from BUS[0-15], then increments the write pointer at the end of the cycle.
EOSFCT	F2=11B	Output Start Function. Sets the OBusy flip flop in the interface, starting data wakeups to fill the FIFO for output. When the FIFO is full, or EEFct has been issued, the interface will wait for silence on the Ether and begin transmitting.
ERBFCT	F2=12B	Reset Branch Function. This command dispatch function merges the ICMD and OCMD flip flops, into NEXT[6-7]. These flip flops are the means of communication between the emulator task and the Ethernet task. The emulator task sets them from BUS[14-15] with the STARTF function, causing the Ethernet task to wakeup, dispatch on them and then reset them with EPFCT.
EEFCT	F2=13B	End of transmission Function. This function is issued when all of the main memory output buffer has been transferred to the FIFO. EEFCT disables further data wakeups.
EBFCT	F2=14B	Branch Function. ORs a one into NEXT[7] if an input data late is detected, or an SIO with AC0[14:15] non-zero is issued, or if the transmitter or receiver goes done. ORs a one into NEXT[6] if a collision is detected.
ECBFCT	F2=15B	Countdown Branch Function. ORs a one into NEXT[7] if the FIFO is not empty.
EISFCT	F2=16B	Input Start Function. Sets the IBusy flip flop in the interface, causing it to hunt for the beginning of a packet: silence on the Ether followed by a transition. When the interface has collected two words, it will begin generating data wakeups to the microcode.

* These functions have a peculiar timing restriction associated with them. The microinstruction that executes one of them must stop the clock for one cycle. On Alto I, the microprogrammer must do this using memory timing (i.e., by referencing MD in the same microinstruction, during the third or fourth cycle of a memory reference). On Alto II, the hardware automatically stops the clock for one cycle when necessary; however, due to a design error, the instruction *following* the one specifying EIDFCT or EILFCT is occasionally stopped instead. Consequently, the programmer must not permit a task switch to occur between these two microinstructions, nor start a memory reference in the following microinstruction.

8.0 CONTROL RAM, ROM, AND S REGISTERS

In addition to the 1K microinstruction ROM containing the standard emulator and I/O microcode, an Alto may contain additional microinstruction memory in the form of either ROM or RAM; these are accompanied by additional registers, called S registers, whose purpose and operation are similar to the standard R registers.

Several different configurations exist, depending on the Alto vintage:

1K RAM — All Altos have at least 1K of read/write microinstruction memory and one bank of 31 S registers. (At one time these were optional on Alto I, but they are now considered standard.)

2K ROM — Certain Alto IIs have 2K of read-only microinstruction memory rather than 1K. The first 1K contain the standard emulator and I/O microcode, and the second 1K may be programmed with additional microcode. This configuration includes the 1K RAM and 31 S registers described previously.

3K RAM — Certain other Alto IIs have 3K of read/write microinstruction memory and 8 banks of 31 S registers.

8.1 RAM-Related Tasks

The control RAM and S registers perform data manipulation (as distinct from microcode fetching) functions in response to certain values of the F1 and BS fields of the microinstruction. Not all tasks are likely to be interested in these functions. Moreover, not all tasks will have the appropriate values of the F1 and BS fields uncommitted. A RAM-related task is defined as one during whose execution the control RAM card will respond to F1 and BS fields of microinstructions. The standard Alto is wired so that the emulator task is the only RAM-related task. At most two other tasks can be made RAM-related by a simple backpanel wiring change.

8.2 Processor Bus and ALU Interface

The Alto's ALU output and processor bus are each 16 bits wide and its microinstruction bus is 32 bits wide, so loading the control RAM from the ALU output and reading the control RAM (or ROM) onto the processor bus is slightly clumsy. It is done by using the RAM-related F1's WRTRAM and RDRAM (see Appendix A).

For both reading and writing, the control RAM address is specified by the control RAM address register (see Figure 2), which is loaded from the ALU output whenever T is loaded from its source. This load may take place as late as the microinstruction in which WRTRAM or RDRAM is asserted. The bits of the ALU output have the following significance as a control RAM address:

BIT	USE
0-1	Ignored (should be zero).
2-3	BANKSEL - Selects RAM bank in 3K RAM configuration; ignored when operating on ROM. 0 RAM0 1 RAM1 2 RAM2 3 Undefined

4 RAM/ROM
0 Means operate on the control RAM.
1 Means operate on the control ROM. (This doesn't quite work the way you might think. See section 8.8 for details.)

5 HALFSEL - Ignored when writing
0 Means read out the low-order 16-bits of the addressed word.
1 Means read out the high-order 16-bits of the addressed word.

6-15 Word address (0-1023).

Since it is expected that reading the control RAM will be a relatively infrequent operation, a single assertion of RDRAM reads out only one half of a 32-bit control RAM (or ROM) word onto the processor bus. To read out both halves, the control RAM address register must be loaded twice and RDRAM invoked twice. Data resulting from RDRAM is AND'ed onto the processor bus during the microinstruction following that in which the RDRAM was asserted.

In contrast, it is expected that writing into the control RAM will occur frequently. Therefore a single application of WRTRAM writes both halves of a control RAM word at once. The M register contents (see section 8.7) after the microinstruction containing the WRTRAM will be written into the high-order half of the addressed control RAM word. The ALU output during the microinstruction following the WRTRAM will be written into the low-order half. This protocol mates well with doubleword main memory reads.

8.3 Microinstruction Bus Interface

The correspondence of ALU output bits with microinstruction fields appears in the following table:

High/Low Order Halfword	Bit of ALU Output	Meaning	Value in Example
H	0-4	R Register Select	0
H	5-8	ALU Function Select	0
H	9-11	Bus Data Source	5
H	12-15*	Function 1	2
L	0-3*	Function 2	0
L	4	Load T	0
L	5*	Load L	1
L	6-15	Next micro address	325B

Fields denoted by * are represented with their high-order bit inverted; this is an artifact of hardware microinstruction decoding.

As an example, consider the representation of the microinstruction

L←MD, TASK, :LOCA;

where LOCA is 325B. The values for the various microinstruction fields are listed in the table above. After complementing the appropriate high-order bits and concatenating, we see that the microinstruction above would be represented as 132B in its high-order halfword and 100325B in its low-order halfword.

8.4 Microinstruction Memory Banks

An alert reader will by now have noticed that the NEXT field of each microinstruction provides a 10-bit address, and that more bits are required to fully address the microinstruction memory. The MI memory is divided into up to four banks of 1024 instructions each:

NAME	WHAT
MI ROM0	The standard microcode ROM.
MI ROM1	Second bank of ROM in the 2K ROM configuration.
MI RAM0	The standard microcode writeable RAM.
MI RAM1	Second bank of RAM in the 3K RAM configuration.
MI RAM2	Third bank of RAM in the 3K RAM configuration.

Switching among banks is controlled in two ways: (1) a *RAM related* task already running may "switch" banks, and (2) it possible to initiate a task in either ROM0 or RAM0.

Bank switching is accomplished with a special transfer mechanism, available only to the emulator task, in the form of SWMODE, a RAM-related F1. SWMODE will switch the bank of the running task, taking effect after the microinstruction following that in which the SWMODE appears. In other words, the emulator task SWMODE behaves much like an address modifier. Tasks other than the emulator cannot switch banks. The effect of SWMODE depends on the ROM/RAM configuration, the bank in which the task is currently executing, and the value of NEXT in the instruction following the one that asserts SWMODE.

In the 1K RAM configuration (neither the 2K ROM nor the 3K RAM option installed):

If currently executing in	go to NEXT in
ROM0	RAM0
RAM0	ROM0

In the 2K ROM configuration (which includes 1K of RAM):

If currently executing in	and NEXT[1]=0 then go to NEXT in	else go to NEXT in
ROM0	RAM0	ROM1
ROM1	ROM0	RAM0
RAM0	ROM0	ROM1

In the 3K RAM configuration:

If currently executing in	NEXT[1]=0		NEXT[1]=1	
	NEXT[2]=0	NEXT[2]=1	NEXT[2]=0	NEXT[2]=1
ROM0	RAM0	RAM2	RAM1	RAM0
RAM0	ROM0	RAM2	RAM1	RAM1
RAM1	ROM0	RAM2	RAM0	RAM0
RAM2	ROM0	RAM1	RAM0	RAM0

If the table above determines that control is to be transferred to the RAM, and the RAM is not installed, control remains in the bank in which the task is currently executing.

Many Alto IIs have the 2K ROM capability but contain nothing in ROM1. In these Altos, the SWMODE operation is normally configured so that it behaves as if ROM1 didn't exist (i.e., according to the first table rather than the second). This is determined by the chip in position 51 on the control board. If it is labelled SW2K then ROM1 exists, but if SW1K then it does not. The alternate chip is kept in unused socket 76.

SWMODE is actually defined in all RAM-related tasks, not just the emulator; however, it does not work correctly in tasks other than the emulator in Altos with the 2K ROM or 3K RAM configuration.

Each of the 16 micro-tasks may be started either in ROM0 or in RAM0 when a hardware reset ("bootstrap") operation is performed, *regardless of whether the task is RAM-related.* A 16-bit "reset mode

register" is used to determine which tasks will start in ROM0 and which will start in RAM0. The emulator F1 RMR← causes the reset mode register to be loaded from the processor bus. The 16 bits of the processor bus correspond to the 16 Alto tasks in the following way: the low order bit of the processor bus specifies the initial mode of task 0, the lowest priority task (emulator), and the high-order bit of the bus specifies the initial mode of task 15, the highest priority task (recall that task *i* starts at location *i*; the reset mode register determines only which microinstruction bank will be used at the outset). A task will commence in ROM0 if its associated bit in the reset mode register contains the value 1; otherwise it will start in RAM0. Upon initial power-up of the Alto, and after each reset operation, the reset mode register is automatically set to all ones, corresponding to starting all tasks in ROM0.

8.5 Standard Emulator Access

The standard emulator includes three instructions allowing basic access to the control RAM. More sophisticated access may be implemented by using the basic access primitives to write other access microcode into the control RAM and then transferring control to that microcode.

RDRAM (61011B) Read from Control RAM:

Reads the control RAM (or ROM) halfword addressed by AC1 into AC0. The microcode is:

```
T←AC1, RDRAM;
L←ALLONES;      (AND'ed with control RAM data)
AC0←L, :START;
```

Note: In Alto IIs running microcode version 2, this instruction does not work reliably if the Ethernet interface is running.

WRTRAM (61012B) Write into Control RAM:

Writes AC0 into the high-order half and AC3 into the low-order half of the control RAM word addressed by AC1. The microcode is:

```
T←AC1;
L←AC0, WRTRAM;      (This loads the M register)
L←AC3;
:START;
```

JMPRAM (61010B) Jump to Control RAM:

This emulator instruction provides a software interface to the SWMODE instruction so that the emulator task may enter another bank in RAM or ROM. The next emulator microinstruction will be determined from the value in AC1 (mod 1024) -- see the discussion of bank switching in section 8.4. *Note that the instruction name (jump to RAM) is misleading, as SWMODE may jump to other places as well.* The microcode for JMPRAM is:

```
T←AC1, BUS, SWMODE;
:NOVEM;     (NOVEM = 0)
```

This operation is fraught with peril. If done in error it is the one of the few emulator instructions which can cause the machine to plunge completely off the deep end. Although clever coders can use JMPRAM to determine whether or not a control RAM is installed, they are better advised to make this determination using WRTRAM and RDRAM (see section 9.2.4).

8.6 Interpretation of Emulator Traps

All unused opcodes except 77400B-77777B (which is used by Swat, the Alto debugger) and 61*xxx*B, where *xxx* is between 0 and 377B, transfer to microlocation RAMTRAP with the instruction in L, the instruction cycled by 8 bits in the R-register XREG, and the emulator's R-register PC counted one beyond the trapping instruction:

```
RAMTRAP:  SWMODE, :TRAP;
...
...
TRAP:  ..., :TRAP1;
```

The result of this is that if your machine has a control RAM, these instructions will cause control to enter it at a location which is equal to TRAP1 in the ROM microcode. If no RAM is present, the unimplemented opcode will be handled as described in Section 3.3.

8.7 M and S Registers

The control RAM card also includes an M register and 31 S registers. If the 3K RAM option is installed, there are 8 banks of 31 S registers (see below). The M register is the analog of the basic Alto's L register. It provides data for the S registers, which are analogous to the basic Alto's R registers. These additional registers are provided to ease the tight constraint on R register availability which might limit the utility of the control RAM.

The similarities between the M and L registers and between the R and S registers are striking. Both M and L are loaded from the output of the ALU, and only when the Load L bit of the microinstruction is active. R registers are loaded from L, and S registers are loaded from M. Both R and S registers output data onto the processor bus. Both R and S registers are addressed by the RSELECT field of the microinstruction. (Thus the same caveats which apply to the use of R37 apply to S37 (see section 2.3 f).) Loading and reading of both R and S registers are controlled by the BS field of the microinstruction.

Nevertheless there are considerable differences. To begin with, the M and S registers are active only when a RAM-related task is executing. This means, for example, that in the highest-priority RAM-related task it is not necessary to save the value of M across a TASK, since no higher-priority task can change the value of M. (It is perilous to take advantage of this "feature", however, since several non-standard Alto peripherals make use of RAM-related tasks.)

Unlike the data path from the L register to the R registers, the data path from the M register to the S registers contains no shifter. When an S register is being loaded from M, the processor bus receives an undefined value rather than being set to zero. The emulator-specific functions ACSOURCE and ACDEST have no effect on S register addressing. And finally, when reading data from the S registers onto the processor bus, the RSELECT value 0 causes the current value of the M register to appear on the bus. (This explains why there are only 31 useful S registers.)

For the purposes of writing microcode, the S registers are assigned numbers 40B through 77B, and appear to the microassembler as if they simply extended the R register address space. Hence, for example, the M register is defined as R40.

In the 3K RAM configuration, there are 8 banks of 31 S registers rather than only a single one. Each RAM-related task has associated with it a 3-bit *register bank number* that determines which bank is referenced when a microinstriction specifies that an S register be read or loaded. There is an emulator F1 called ESRB← and a RAM-related F1 called SRB← that sets the register bank number for the currently-executing task from BUS[12-14]. It is illegal to execute ESRB← or SRB← in the last cycle before a task switch, i.e., in the microinstruction after a TASK is executed.

Note that the function code is different for emulator and non-emulator tasks: ESRB← is F1=15 and is defined only in the emulator task, while SRB← is F1=13 and is defined in all RAM-related tasks besides the emulator. (F1=13 corresponds to RMR← in the emulator. In Altos without the 3K RAM option, F1=13 performs RMR← in all RAM-related tasks, including the emulator.)

The register bank numbers are all reset to zero by a reset (bootstrap) operation, thereby causing the Alto to behave the same as a standard Alto with a single bank of S registers shared among all RAM-related tasks.

8.8 Restrictions and Caveats

1. Both RDRAM and WRTRAM cause the microprocessor's system clock to stop for one cycle. This may yield unspecified results if the system clock is also stopped for some other reason (e.g., waiting for memory data). As a general rule, the system clock should run without hesitation during the microinstruction following a RDRAM or WRTRAM, except for the effect of the RDRAM or WRTRAM itself. On Alto I, there is an additional timing problem which manifests itself in some machines, for example, in the following microcode sequence:

```
MAR←FOO;          Starts memory reference
T←FIE;            Loads the control RAM address register
L←MD, WRTRAM;     Save away the high-order word in M
L←MD;             Completes the write into the RAM
```

What happens is that the last instruction suspends the system clock for one microinstruction, and some Alto I memories cannot keep the memory data good for two microinstruction times, so a parity error may occur. The data is actually stored in the RAM at the end of the first microinstruction time, so there is probably no error in the data even if a parity interrupt subsequently occurs. This "phantom" parity error may be averted by the following code, which takes three more microinstruction times, but does not invoke the horrendous microcode overhead of parity error recording:

```
MAR←FOO;          Starts memory reference
NOP;              Required for memory timing
L←MD;             Save away the low-order word
T←MD;             Save away the high-order word
TEMP←L, L←T;
T←FIE, WRTRAM;    Loads the address register, starts the write.
L←TEMP;           Complete the write into the RAM
```

2. Unlike the control RAM, which can be addressed from 2 places, the control ROM gets its address only from the MPC RAM. Consequently, to read ROM location x, the instruction following the one with F1=12B (RDRAM) must reside at location (x mod 1024). Therefore, you'll probably want to put the "reading" code in the RAM:

```
    T←AC1, RDRAM, :X;     Only AC1[4-5] are relevant
X:  L←ALLONES;            Here the read takes place
    AC0←L, ...
```

Note also that only ROM0 can be read by these means. There is no known way to read ROM1.

3. Some Alto Is have been observed not to evaluate the BUS=0 function correctly when reading an S-register during the first microinstruction after a task switch. The same operation in other than the first microinstruction causes no difficulty.

9.0 NUTS AND BOLTS FOR THE MICROCODER

9.1 Standard Microcode Conventions

The microassembler which assembles microcode for the Alto is called Mu. . By convention, microcode source files have the extension .MU, and binary files have the extension .MB. Standard Alto I ROM microcode versions will be called AltoCode*x*.MU; those for Alto II will be called AltoIICode*x*.MU. A microcode source file can be divided into three largely separable pieces: the language definitions, which tell Mu what names will be used for what octal values of what microcode fields; the constant definitions, which declare all constants that may later be referenced, and which cause the constant memory to be laid out; and the register declarations, microinstruction label declarations, and microinstructions.

In order for microprograms written to execute in the RAM to be compatible with those in the ROM, at a minimum the constants assumed by the RAM microcode must be a subset of those declared by the ROM microcode, and the subset must reside in the same addresses. As a practical matter, one should preface one's RAM microcode by the same constant definitions which were used in the assembly of one's ROM microcode. In order to facilitate and encourage this compatibility, the file AltoConsts*x*.MU will be maintained (the *x* corresponding to the latest AltoCode*x*) containing definitions and constants for both Alto I and Alto II. These can be logically incorporated into other microcode assemblies via the "include" feature of Mu (#AltoConsts*x*.MU;).

If one or more microcode tasks pass control back and forth between ROM and RAM, it becomes necessary to associate addresses with microinstruction labels. It is possible to do this completely generally, based on the microcode version number. A more limited solution is simply to fix the addresses of certain useful labels. The following addresses are guaranteed in all standard Alto I microcode versions after 20, and all standard Alto II microcode versions (and are included in AltoConsts*x*.MU):

ADDRESS	LABEL	SEMANTICS
20B	START	Beginning of emulator's main loop; starts a new emulated instruction.
37B	TRAP1	RAM location to which unfamiliar traps are sent; ROM location which implements trap sequence.
22B	RAMCYCX	Fast cyclic shift subroutine.
105B	BLT	Block transfer subroutine.
106B	BLKS	Block store subroutine.
120B	MUL	Multiply subroutine.
121B	DIV	Divide subroutine.
124B	BITBLT	BITBLT subroutine.
160B	L0	Cyclic shift dispatch table.
777B	SWRET	In ROM1 only -- see below

A standard convention requires that location SWRET in ROM1 have the following microcode:

```
SWRET:  SWMODE;
        :START;
```

This sequence enables a program to discover whether ROM1 exists, i.e., whether the Alto has the 2K PROM option (see section 9.2.4).

9.2 Microcode Techniques Which Need Not Be Rediscovered

For the most part, since the Alto is such a simple machine, writing Alto microcode is a straightforward exercise in rule-following. However, during the course of writing the few-odd thousand microinstructions which have ever been written by anybody for the Alto, a few microcoding techniques have emerged as particularly ingenious or useful or both. They are recorded here for posterity.

The beginning microcoder is advised to acquire a copy of the standard microcode (AltoCode*x*.MU), and to study it carefully in conjunction with this manual. The knack comes easily.

9.2.1 Microcode Subroutines

You have probably already noticed that that the Alto hardware does not provide an easy way of doing microcode-level subroutine calls and returns. Several subroutine-call techniques have evolved. Two of these are used for RAM-to-ROM subroutine calls, and these will be presented first.

PC CALL (used with BLT, BLKS, MUL, DIV, BITBLT)

This call takes advantage of the assumption that nobody in his right mind would want the emulator to execute in the non-memory I/O area from 177000B to 177777B. Therefore when one of these ROM subroutines terminates, the R-register PC is examined. If it is outside the range 177000B-177777B, then control is passed to the beginning of the emulator's main loop in the ROM. Otherwise, control is passed to location PC AND 777B in RAM or ROM1. The bank dispatched to is determined by the SWMODE rules described in section 8.4.

Warning: Some of these ROM subroutines modify PC during execution. If BLT or BLKS or BITBLT is terminated by an interrupt condition, PC is decremented by 1 so that the instruction can be resumed later. If a DIV is successful, PC is incremented by 1 to cause a skip.

REGISTER CALL (used with RAMCYCX)

This call uses an R-register, in this case CYRET (R-register 5), to dispatch into a table of successor instructions. The cyclic shift subroutine, for example, is called from six places in the ROM. Each of these places sets CYRET to the index of its successor instruction in the return dispatch table [0-5], and then dispatches into the cycle table beginning at L0. The successor corresponding to RAMCYCX dispatches into RAM or ROM1 using the low-order 10 bits of the PC register, according to the SWMODE rules described in section 8.4.

IR CALLS

These calls use the emulator's IR register in various ways: some straightforward and some devious. The main advantages of IR calls are that

1) several levels of return can be encoded into a single number, because it is fairly easy to dispatch on various parts of IR, and

2) unlike R-registers, IR can be loaded in one microinstruction.

The most straightforward use of IR is dispatching on its low-order 8 bits using the DISP bus source. Since DISP is a bus source >3, a constant may be "and-ed" onto the bus with DISP, allowing one to dispatch on sub-fields of DISP.

The most devious use of IR involves a group of constants labeled sr0 to sr12, sr14 to sr17, and sr20 to sr37 (as you might suspect, the numbers on these constant names are octal). If the constant sr*i* has been loaded into IR, then the following code will cause control to transfer to location FOO OR *i*:

```
IDISP;          (see section 3.5)
:FOO;
```

The statement above is only true if *i* is less than 20B; otherwise an additional dispatch on the DISP field of IR is required to get the desired effect:

```
FOO13:      SINK←DISP, BUS;
            :FOO20;
```

(This explains why there is no sr13. Any of sr20-sr37 will carry control to the 13Bth entry in FOO's dispatch table, where an additional level of dispatch can be used to differentiate among them if necessary. You may be wondering what is special about 13B. You are in good company.)

9.2.2 The Silent Boot

Many of the effects of a hardware "reset" operation (invoked by the boot button, or BUS[0]=1 in conjunction with the emulator-specific F1 STARTF (17B)) can be faithfully simulated by emulated software. At least two important ones cannot. A reset operation is the only way of moving non-RAM-related tasks back and forth between ROM0 and RAM0, and the only way of guaranteeing that all tasks are initialized. However, the time required for a reset operation is not necessarily longer than a few microseconds. On both Alto Is and Alto IIs a reset operation does not alter the contents of the Alto's R or S registers, its microinstruction RAM, or its main memory. Therefore if these memories contain appropriate contents it is not really necessary to go through the full disk or Ethernet bootstrap load sequence, since the major purpose of those sequences is to initialize these memories with desired contents.

The "silent boot" consists first of getting the desired contents into the RAM and main memory. RAM0 should contain an emulator task (beginning with address 0) which, for example, simply jumps into the main loop of the ROM emulator code, skipping all the bootstrap code. For example:

```
NOVEM:  SWMODE;     (RAM0 location 0, task 0's reset location,)
        :START;     (to ROM0 location 20B)
```

Second, the reset mode register should be set so that the reset operation will begin execution of the emulator task in RAM0, and the other tasks wherever they are desired. Finally, the reset operation is initiated, the emulator hiccoughs momentarily into RAM0, and then proceeds in ROM0 as if nothing had happened.

9.2.3 Debugging the Emulator

As someday it may happen that a bug must be found in a new version of the emulator, microcodes should be aware of a nice trick. Suppose you have an Alto with a working emulator in its ROM, and load the suspect emulator into the RAM. Your courage leads you to execute a JMPRAM with AC1=20B (START), and hope that the new emulator behaves. But alas, the machine dives into oblivion. Now the trick applies: before jumping into the RAM version, plant a JMPRAM (with AC1=20B) somewhere in the Nova code that you know will be executed. Now go to the RAM with the horrid JMPRAM. If the suspect emulator has not died by the time it executes the JMPRAM you planted, control will return to the benign ROM. This method, together with the obvious search technique, may locate an offending emulator instruction.

9.2.4 How to tell if extended ROM or RAM exists

A standard convention assures that location 777B in ROM1, if it exists, contains the code:

```
SWRET:  SWMODE;
        :START;
```

First, we store the following snatch of code in RAM0, with INRAM located at location 777B:

```
INRAM:  L←AC0+1, SWMODE;
        AC0←L, :START;
```

Now we store 0 in AC0, and use the JMPRAM emulator instruction to branch to location 777B. This will cause either the SWRET or INRAM code to be executed; in any case, the emulator instruction following the JMPRAM will eventually be executed. If AC0 has been set to 1, ROM1 does not exist; otherwise ROM1 does exist.

To determine whether the 3K RAM option is present, use WRTRAM to write different values into corresponding locations in two different RAM banks, then use RDRAM to read back the first location written. If the 3K RAM option is present, the location will still contain the value written into it; if the option is absent, it will have been clobbered by the value intended for the second RAM bank.

9.2.5 RAM Utility Area

It sometimes happens that a small piece of microcode must be loaded into the RAM so that the emulator can execute it by doing a JMPRAM to it; it will then return to the emulator. For example, such a piece of code is required in order to set the reset mode register. By convention, we reserve a *utility area* of RAM0 for this purpose. The normal procedure is to save the contents of this area (using RDRAM), store the piece of code that is to be executed (using WRTRAM), execute the code (using JMPRAM), and then restore the original contents. Writers of microcode should avoid placing code in the utility area that is not part of the emulator task, as it may be temporarily altered for these utility operations.

The normal utility area is 774B through 1003B inclusive. The alert reader will recognize that JMPRAM can successfully transfer into this area in RAM0 when coming from ROM0 (locations 1000B-1003B are accessible) or from ROM1 (locations 774B-777B are accessible). A program will therefore need to know where it is executing (ROM0 or ROM1) and use an appropriate entry point to the utility area.

9.2.6 Other Information

Correct operation of most Alto peripherals depends vitally on their tasks receiving adequate service. This in turn depends on two things:

1. A task must have sufficient priority to gain however many cycles it needs for service, at the expense of lower-priority tasks. The choice of priority must be made carefully when the interface is designed.

2. Other tasks at the same and lower priorities must be well-behaved. In particular, they must perform task switches no further apart than the maximum latency permitted for the task in question.

It is believed that the standard Alto peripheral most sensitive to task latency is the Diablo disk controller when connected to a Model 44 disk drive. This is due to the fact that the data rate is relatively high and the controller has only 16 bits of buffering.

It has been determined empirically that task latency greater than 20 microinstruction times causes Diablo Model 44 disks to encounter data-late errors. Therefore, when writing microprograms, it is essential that you issue a TASK at least once every 20 microinstructions (preferably once every 15). When counting microinstruction times, do not forget to include the cycles during which the processor is suspended due to memory references.

APPENDIX A - MICROINSTRUCTION SUMMARY

FIELDS:

0-4	RSELECT
5-8	ALUF
9-11	BS
12-15	F1*
16-19	F2*
20	LOAD T
21	LOAD L & M*
22-31	NEXT

*High-order bit complemented by RDRAM and WRTRAM.

All subsequent numbers on this page are in octal.

ALUF:

0: BUS	4: BUS XOR T	10: BUS-T	14: BUS.T*
1: T	5: BUS+1*	11: BUS-T-1	15: BUS AND NOT T
2: BUS OR T*	6: BUS-1*	12: BUS+T+1*	16: UNDEFINED
3: BUS AND T	7: BUS+T	13: BUS+SKIP*	17: UNDEFINED

*Loads T from ALU output

BUS SOURCE (standard):

0: ←RLOCATION	4: (task-specific)
1: RLOCATION←	5: ←MD
2: None (BUS←-1)	6: ←MOUSE
3: (task-specific)	7: ←DISP

F1 (standard):

0: -	4: ←L LSH 1
1: MAR←	5: ←L RSH 1
2: TASK	6: ←L LCY 8
3: BLOCK	7: ←CONSTANT

F2 (standard):

0: -	4: BUS
1: BUS=0	5: ALUCY
2: SH < 0	6: MD←
3: SH = 0	7: ←CONSTANT

BUS SOURCE (task-specific):

	0 CPU	4,16 KSEC,KWD	7 ETHER	RAM Related
3:	←SLOCATION	←KSTAT	-	←SLOCATION
4:	SLOCATION←	←KDATA	EIDFCT	SLOCATION←

F1 (task-specific):

	0 CPU	4, 16 KSEC,KWD	7 ETHER	11 DWT	12 CURT	13 DHT	14 DVT	RAM Related
10:	SWMODE	-	-	-	-	-	-	(SWMODE)
11:	WRTRAM	STROBE	-	-	-	-	-	WRTRAM
12:	RDRAM	KSTAT←	-	-	-	-	-	RDRAM
13:	RMR←	INCRECNO	ELFCT	-	-	-	-	SRB←
14:	-	CLRSTAT	EPFCT	-	-	-	-	-
15:	ESRB←	KCOMM←	EWFCT	-	-	-	-	-
16:	RSNF	KADR←	-	-	-	-	-	-
17:	STARTF	KDATA←	-	-	-	-	-	-

F2 (task-specific):

	0 CPU	4, 16 KSEC,KWD	7 ETHER	11 DWT	12 CURT	13 DHT	14 DVT	RAM Related
10:	BUSODD	INIT	EODFCT	DDR←	XPREG←	EVENFIELD	EVENFIELD	-
11:	MAGIC	RWC	EOSFCT	-	CSR←	SETMODE	-	-
12:	DNS←	RECNO	ERBFCT	-	-	-	-	-
13:	ACDEST	XFRDAT	EEFCT	-	-	-	-	-
14:	IR←	SWRNRDY	EBFCT	-	-	-	-	-
15:	IDISP	NFER	ECBFCT	-	-	-	-	-
16:	ACSOURCE	STROBON	EISFCT	-	-	-	-	-
17:	-	-	-	-	-	-	-	-

APPENDIX B - STANDARD RESERVED MEMORY LOCATIONS

All numbers are in octal.

Location	Name	Contents
Page 0:		
0-17		Set to 77400B by OS (Swat)
Page 1:		
400-412		Used by standard bootstrap operation
420	DASTART	Display list header (Std. Microcode)
421	-	Display vertical field interrupt bitword (Std. Microcode)
422	ITQUAN	Interval timer stored quantity (Std. Microcode)
423	ITBITS	Interval timer bitword (Std. Microcode)
424	MOUSEX	Mouse X coordinate (Std. Microcode)
425	MOUSEY	Mouse Y coordinate (Std. Microcode)
426	CURSORX	Cursor X coordinate (Std. Microcode)
427	CURSORY	Cursor Y coordinate (Std. Microcode)
430	RTC	Real Time Clock (Std. Microcode)
431-450	CURMAP	Cursor bitmap (Std. Microcode)
452	WW	Interrupt wakeups waiting (Std. Microcode)
453	ACTIVE	Active interrupt bitword (Std. Microcode)
457	-	Zero (Extension of MASKTAB by convention; set by OS)
460-477	MASKTAB	Mask table for convert (Std. Microcode; set by OS)
500	PCLOC	Saved interrupt PC (Std. Microcode)
501-517	INTVEC	Interrupt Transfer Vector (Std. Microcode)
521	KBLK	Disk command block address (Std. Microcode)
522	KSTAT	Disk status at start of current sector (Std. Microcode)
523	KADDR	Disk address of latest disk command (Std. Microcode)
524	-	Sector interrupt bit mask (Std. Microcode)
525	ITTIME	Interval timer time (Std. Microcode)
527	TRAPPC	Trap saved PC (Std. Microcode)
530-567	TRAPVEC	Trap vector (Std. Microcode)
570-577	-	Timer data (OS)
600	EPLOC	Ethernet post location (Std. Microcode)
601	EBLOC	Ethernet interrupt bit mask (Std. Microcode)
602	EELOC	Ethernet ending count (Std. Microcode)
603	ELLOC	Ethernet load location (Std. Microcode)
604	EICLOC	Ethernet input buffer count (Std. Microcode)
605	EIPLOC	Ethernet input buffer pointer (Std. Microcode)
606	EOCLOC	Ethernet output buffer count (Std. Microcode)
607	EOPLOC	Ethernet output buffer pointer (Std. Microcode)
610	EHLOC	Ethernet host address (Std. Microcode)
611-612	-	Reserved for Ethernet expansion (Std. Microcode)
613	-	Alto I/II indication that microcode can interrogate (0=Alto I, -1=Alto II)
614	DCBR	Posted by parity task when a main memory parity error is detected.
615	KNMAR	" (Std. Microcode)
616	DWA	"
617	CBA	"
620	PC	"
621	SAD	"
(Note: Disk and Ethernet bootstrap loaders run in 622-777.)		
700-707	-	Saved registers (Swat)
Page 376B:		
177016-177017	UTILOUT	Printer output (Std. Hardware)
177020-177023	XBUS	Utility input bus (Alto II Std. Hardware)
177024	MEAR	Memory Error Address Register (Alto II Std. Hardware)
177025	MESR	Memory error status register (Alto II Std. Hardware)
177026	MECR	Memory error control register (Alto II Std. Hardware)
177030-177033	UTILIN	Printer status, mouse, keyset (all 4 locations return same thing)
177034-177037	KBDAD	Undecoded keyboard (Std. Hardware)
Page 377B:		
177740-177757	BANKREGS	Extended memory option bank registers -- see section 2.3

APPENDIX C - RESERVED SIO BITS

Bit 0	100000B	Standard Alto: Software boot feature -- See SIO, section 3.3
Bit 14	000002B	Standard Alto: Ethernet
Bit 15	000001B	Standard Alto: Ethernet

APPENDIX D - STANDARD TASKS

Task	Name	Section	Description
0	Emulator	3	Lowest priority. Wakeup always true.
1	-	-	unused
2	-	-	unused
3	-	-	unused
4	KSEC	6	Disk sector task
5	-	-	unused
6	-	-	unused
7	ETHER	7	Ethernet task
10B	MRT	-	Memory refresh task. Wakeup every 38.08 microseconds.
11B	DWT	4	Display word task
12B	CURT	4	Cursor task
13B	DHT	4	Display horizontal task
14B	DVT	4	Display vertical task. Wakeup every 16.666 milliseconds.
15B	PART	5.5	Parity task. Wakeup generated by parity error.
16B	KWD	6	Disk word task
17B	-	-	unused

APPENDIX E - S-GROUP INSTRUCTION SUMMARY

Opcode	Trap location	Name
60000-60377	---	CYCLE
60400-60777	531	RAM trap
61000-61377	532	Parameterless opcodes to 61026, ROM trap for rest
61400-61777	533	RAM trap
62000-62377	534	RAM trap
62400-62777	535	RAM trap
63000-63377	536	RAM trap
63400-63777	537	RAM trap
64000-64377	540	RAM trap
64400-64777	---	JSRII
65000-65377	---	JSRIS
65400-65777	543	RAM trap
66000-66377	544	RAM trap
66400-66777	545	RAM trap
67000-67377	---	CONVERT
67400-67777	547	RAM trap
70000-70377	550	RAM trap
70400-70777	551	RAM trap
71000-71377	552	RAM trap
71400-71777	553	RAM trap
72000-72377	554	RAM trap
72400-72777	555	RAM trap
73000-73377	556	RAM trap
73400-73777	557	RAM trap
74000-74377	560	RAM trap
74400-74777	561	RAM trap
75000-75377	562	RAM trap
75400-75777	563	RAM trap
76000-76377	564	RAM trap
76400-76777	565	RAM trap
77000-77377	566	RAM trap
77400-77777	567	ROM trap, reserved for Swat

APPENDIX F - ALTO I / ALTO II DIFFERENCES

The minor differences between Alto I and Alto II are explained in this manual. This appendix serves as an index of those differences:

Memory reference timing (section 2.3)
Certain emulator instructions (RCLK, SIO, SIT, VERS, DREAD, DEXCH, DIAGNOSE1, DIAGNOSE2; section 3.3)
Keyboard layout (section 5.1)
External device connector (section 5.4)
Memory configuration switch (section 5.5)
Memory parity error detection (section 5.5)
2K ROM and 3K RAM options (section 8.4)
Extended memory option (section 2.3)

APPENDIX G - SUMMARY OF KNOWN FEATURES/BUGS IN RELEASED MICROCODE VERSIONS

Alto I version 23:

VERS instruction:	returns engineering number 0, microcode version 1.
BITBLT instruction:	doesn't work reliably if some ram-related task is running (e.g., the Trident disk).

Alto II version 2:

VERS instruction:	returns engineering number 2, microcode version 0.
BITBLT instruction:	doesn't work reliably if some ram-related task is running (e.g., the Trident disk). Expects L to be zeroed by the caller.
RDRAM instruction:	does not work reliably when the Ethernet interface is active.
DEXCH instruction:	does not work at all.
SIT instruction:	TIMEMASK is 7700B but should be 7774B. Fails to store into ITQUAN.
ACSOURCE function:	does not work precisely as documented. Consult McCreight if you really need to know.

Alto I version 24:

No known bugs.

Alto II version 3:

SIT instruction:	Fails to store into ITQUAN.

APPENDIX H - PARC/SDD RESERVED MEMORY LOCATIONS

All numbers are in octal.

Location	Name	Contents
Page 0:		
451	-	Color map pointer
456	-	Mesa disaster flag
526	-	SamllTalk trap exit instruction
622	-	Tape control block list
630-640	-	Second Ethernet control block
631-661	-	Hexadecimal floating-point microcode
640-644	-	Trident disk control block
640-651	-	Third Ethernet control block
720-777	-	SLOT devices
776-777	-	Music
Page 376B:		
177100	-	Summagraphics tablet X
177101	-	Summagraphics tablet Y
177140-177157	-	Organ keyboard
177200-177204	-	PROM programmer
177234-177237	-	Experimental cursor control
177240-177257	-	Alto II debugger
177244-177247	-	Graphics keyboard
Page 377B:		
177400-177405	-	Maxc2 maintenance interface
177400	-	Alto DLS input
177420	-	"
177440	-	"
177460	-	"
177600-177677	-	Alto DLS output
177700	-	EIA interface output bit
177701	EIALOC	EIA interface input bit
177720-177737	-	TV Camera Interface
177764-177773	-	Redactron tape drive
177776	-	Digital-Analog Converter
177776	-	Digital-Analog Converter, Joystick
177777	-	Digital-Analog Converter, Joystick

APPENDIX I - PARC/SDD RESERVED SIO (STARTF) BITS

Bit 1	040000B	Maxc2 Memory Interface
Bit 2	020000B	Maxc2 Memory Interface
Bit 3	010000B	Maxc2 Memory Interface
Bit 4	004000B	Aurora
Bit 5	002000B	Arpanet Interface
Bit 6	001000B	Arpanet Interface
Bit 8	000200B	Tape controller
Bit 9	000100B	available
Bit 10	000040B	Trident disk interface
Bit 11	000020B	Trident disk interface
Bit 12	000010B	available
Bit 13	000004B	Printer interfaces (Orbit, Slot)

Bits 10-11 Second Ethernet interface
Bits 12-13 Third Ethernet interface

APPENDIX J - PARC/SDD TASKS

Devices	Tasks
Trident Disk Controller	3 and 17B
Orbit	1
Slot	1
Tape Controller	5 and 6
Audio	?
Aurora	?
Maxc2 Memory Interface	17B

APPENDIX K - OPTIONAL ALTO PERIPHERALS

This appendix lists hardware items that have been interfaced to the Alto in quantities greater than one. EOD/SPG is the source for information about many of these interfaces and devices, and may be willing to contract to provide necessary hardware. Sources in PARC are not committed to producing any hardware. No software guarantees are made about any of these devices, except as noted.

HyType Printer. A spinning daisy printer can be ordered from Diablo Systems, Inc. Arrangements can be made with SPG to build a cable that will connect the printer to the "printer connector" on the rear of the Alto. No additonal hardware is required, although printers attached to Alto II are required to be self-powered. Software: Bravo prints on the Diablo printer, and a Bcpl subroutine package (DiabloPrinter.Br) is available to drive the interface.

Versatec Printer/Plotter. The Versatec plotters and printer/plotters can be connected to the Alto II without additional hardware. Contact SPG to get a cable (P/N 216540).

Tape Controller. A two-card processor-bus interface to MDS and Kennedy tape drives. It will handle 1600 bpi phase-encoded tapes only. Contact ASD-South.

Trident Disk Interface. An interface to the Trident family of disk drives, manufactured by Calcomp. Alto II owners should contact SPG, Alto I owners contact PARC/CSL. Software: The Trident disks may be accessed in conjunction with Operating-System routines, using the TFS software package (see Alto Subsystems documentation).

Orbit. A piece of hardware which can be used to drive a variety of SLOT printers that obey the "9-wire standard ROS interface." Contact ASD-South.

Extra Ethernets. Up to two extra Ethenets can be installed in an Alto of any vintage. Contact PARC/CSL.

Ethernet Repeaters. Many miles of Ethernet can be hooked together with these. Contact PARC/CSL.

ArpaNet (BBN 1822) Interface. An interface to ArpaNet Imps and Packet Radio Units. Contact PARC/SSL.

EIA Interface. An interface to an AMI S1883 UART and an AMI S2350 USRT. Contact ASD-South.

Communications Processor. Terminates up to 16 lines at many speeds, codes and line control disciplines. Contact ASD-South.