Inter-Office Memorandum

To	Alto Gateway Project	Date	February 10, 1980
From	Ed Taft	Location	Palo Alto
Subject	Alto Gateway Microcode Package	Organization	PARC/CSL

XEROX

Filed on: [Ivy]<Portola>GatewayMicrocode.press

The Gateway Microcode package provides facilities by means of which Mesa or Bcpl programs may conveniently and efficiently control either of two communication interfaces: the XEOS Alto EIA board or the ASD Communication Processor. It also includes various other pieces of microcode useful in Alto Gateways.

The microcode package has the following properties.

It permits operating communication lines at relatively high speeds: many lines at 9.6 KB, or a few (probably not more than two) lines at 56 KB, full-duplex.

It provides a framework making it possible to control individual lines independently of each other. That is, the microcode may provide independent *software interfaces* to each line.

The present microcode implements the complete finite state automata for reception and transmission of data frames (packets) using a subset of the BiSync protocol. The software specifies packet reception and transmission by means of queues of Line Command Blocks. The software is notified (by an interrupt) when reception or transmission of a packet has been completed.

Other software interfaces presently implemented are character-at-a-time reception and transmission via ring buffers and uninterpreted block-at-a-time transmission.

Separate versions of the microcode exist for the EIA board and the CommProc. While the two pieces of hardware are quite different, the software interfaces and overall organization of the driver are virtually identical. The only intentional differences have to do with initialization, exception handling, and interpretation of control and status words. Note, however, that simultaneous operation of an EIA board and a CommProc in the same Alto is *not* supported.

The microcode is usable by both Mesa and Bcpl programs. Slightly different versions of the microcode are required for the two languages.

This document describes the software interface to the Gateway microcode package. Information available from the hardware documentation for the EIA board and the CommProc is not duplicated here. Reference is made to the one-page command and status word format summary provided by XEOS, and to the *CommProc Functional Specification (hardware)* provided by ASD.

All information applies to both interfaces and to both Bcpl and Mesa programming unless otherwise specified. All numbers are decimal unless suffixed with 'B', in which case they are octal.

Hardware requirements

The microcode runs only on an Alto-II with the Extended Memory option (though there is no necessity for more than 64K of memory actually to be installed).

The microcode is usable by Mesa programs only if the Alto has a 2K Control board, with the Mesa microcode blown into ROM1.

EIA board

One or more EIA boards should be installed as described in the documentation accompanying them. The XEOS wire-wrap interface requires several modifications before it may be used with the EIA microcode. These modifications will be detailed in a separate memo. If the IC in location D2 is a 74LS174, the modifications have already been made; if it is a 74LS175, they have not. The ASD printed-circuit interface is expected to incorporate these changes.

A modification is required to make the Memory Refresh Task be RAM-related. This involves adding a single wire to the backplane: MRTACTIVE' (slot 11, pin 109) to either TASKA' (slot 10, pin 13) or TASKB' (pin 14), whichever isn't being used by other devices. If both an EIA board and a CommProc are installed in the same Alto, TASKB' must be used, because the CommProc uses TASKA'.

CommProc

The CommProc installation instructions should be followed. No other modifications are required.

Exception: if two additional Ethernet interfaces are also installed in the same Alto, the CommProc interval timer must be disabled because the task which it uses (task 1) is needed for the second extra Ethernet. To do this, while installing the CommProc *omit* the connection of WAKE1' to pin 114 and 1ACT' to pin 119 of slot 16.

Organization and data structures

The EIA microcode runs as part of the Memory Refresh Task. Every 38 microseconds, it checks to see whether any EIA line is requesting service, and if so performs all processing required. The CommProc microcode runs as a dedicated task that wakes up only when service is required for data transfers.

In either case, the microcode requires the presence of several data structures in main memory. These data structures are used both for maintaining line state and for communication with software.

Line table

All line state is maintained in a table in main memory, pointed to by an S-register called LINTAB. Five other S-registers are used for scratch; all six must be dedicated to the Gateway microcode. The S-registers presently assigned are 71 through 75 and 77, which have been chosen not to conflict with the Mesa emulator.

Associated with each line is a six-word Line Table Entry (LTE), whose first word lies at LINTAB + 6**line*, where *line* is the communication line number (in the range 0 to 7 for EIA boards, 0 to 15 for CommProc). LTEs must lie on even-word boundaries. The structure of an LTE is:

Word 0	Unused.
Word 1	*EIA board only:* Hardware status is stored here every time the EIA line requests service, for whatever reason. See the "Status Word Format" in the EIA interface documentation. The microcode interprets the INTP flag as a request for service.
Word 2	Pointer to current input Line Command Block (LCB), or zero if no LCB is

set up.

Word 3 Current input state (described below).

Word 4 Pointer to current output LCB, or zero if no LCB is set up.

Word 5 Current output state.

Additionally, a few words below the beginning of the line table are reserved for various purposes, as will be described later. Here is a summary:

Word -4 *CommProc only:* Interval Timer interrupt bit mask.

Word -3 *CommProc only:* Interval Timer interval.

Word -2 *CommProc only:* Interval Timer counter.

Word -1 Pointer to CRC table.

Line states

The state word is a small integer representing the state of the microcode with respect to a given line and direction. When a line requests service, the microcode uses the state to dispatch to the appropriate stretch of microcode for handling the line at that point. The microcode updates the state when appropriate.

Associated with the finite state automaton for each software interface is a set of states. Each set is closed; i.e., the microcode will transition only to other states in the same set. The software is expected to initialize the input and output state words of each line to the *initial* states of the software interfaces being used. Thereafter, the microcode maintains the states automatically. The initial states presently defined are:

0 Throwaway mode. Input data is thrown away and output requests are ignored *(EIA board)* or cause Send Enable to be reset *(CommProc)*. Unused lines should have their states initialized to this value.

1 BiSync data frame input.

2 BiSync data frame output.

3 Character-at-a-time input via a ring buffer.

4 Character-at-a-time output via a ring buffer.

5 Uninterpreted block output.

When additional software interfaces are implemented, their initial states will be added to this list.

Line Command Blocks

Input and output data transfers are specified by separate chains of Line Command Blocks (LCBs). The LTE contains pointers to the first LCB in each chain. When a complete packet transfer has taken place, the microcode stores ending status in the LCB, issues an interrupt (if the software so desires), and begins processing the next LCB in the same chain. Exception: the Character-at-a-time software interfaces use only a single LCB, which describes a ring buffer that is used indefinitely.

An LCB must be aligned on an even-word boundary. The first four words of every LCB have a standard format, regardless of the software interface; additional words are interface-dependent. Words marked '*' are modified by the microcode and must therefore be reinitialized before every use of the LCB; other words are never stored into by the microcode.

Word 0	Link to next LCB, or zero if none.
Word 1	Bit mask indicating channel(s) on which an interrupt is to be initiated when processing of the LCB has been completed.
Word 2*	Ending microcode status (see below). The software must initialize this to zero before appending the LCB to a chain.
Word 3*	Ending hardware status.

The microcode indicates that it has completed processing an LCB by setting bit 0 of the microcode status word (word 2), storing an error code in bits 1-15 of the same word, and storing hardware status in word 3. The microcode may store an error code and hardware status prior to the end of the transfer, so the software should look only at bit 0 of the microcode status to determine whether LCB processing has completed.

The following error codes are independent of the software interface in use on the line. Other codes are defined by specific software interfaces.

0 No error.

1 A fatal hardware status was reported. The hardware status word stored in the LCB will show what the error was. The microcode will post this code in preference to a software interface-dependent error if both occur during processing of the same LCB.

During input, the status bits treated as fatal errors are the following:

RPE (Receive Parity Error): Can occur only if parity checking has been enabled.

ROR (Receive Overflow Error): Input data late, characters lost.

RFE (Receive Framing Error): Can occur only when the line is in asynchronous mode.

During output, the fatal error status bits are:

FCT (Fill Character Transmitted): *EIA board only.* Can occur only when the line is in synchronous mode. Indicates that the USRT transmitted a fill character because a new data character was not supplied within one character time after a request. Note that this is a perfectly normal condition between frames, when there is no data to be sent. It is an error only if it occurs within a frame. The microcode for block-mode transmission is careful to ignore FCT status at the beginning of a frame. However, during character-at-a-time transmission, the software must make this distinction.

Transmit Overrun: *CommProc only*.

Emulator-level subroutine calls

The Mesa or Bcpl software communicates with the microcode by three means: manipulating the shared data structures, directly manipulating the hardware interface registers, and calling Ram subroutines that have been included in the microcode. Ram subroutine calls are accomplished somewhat differently in Mesa and Bcpl.

The following examples show how to call the SilentBoot subroutine, which takes a single argument and sets the Boot Locus Vector to it, then boots the Alto. Other Ram subroutines are declared and called similarly.

In the Mesa version of the microcode, subroutine entry points are at fixed locations in the Ram and are intended to be called directly by means of the JRAM Mesa opcode, with argument and result values passed through the Mesa stack in the usual fashion. The entry point to the SilentBoot subroutine is Ram location 1400B. The subroutine is declared and called as follows. This is valid only in Mesa versions 4.0 and later.

```
silentBootAddr: CARDINAL = 1400B; -- Ram entry point for SilentBoot
SilentBoot: PROCEDURE[bootLocusVector: WORD] =
    MACHINE CODE BEGIN
    Mopcodes.zLIW, silentBootAddr/256, silentBootAddr MOD 256;
    Mopcodes.zJRAM;
    END;

SilentBoot[177376B]; -- the call --
```

Ram subroutine calls must be complete Mesa statements, not nested expressions. When a Ram subroutine returns a value, a call to it must be a statement of the form "simpleVariable _ Routine[arguments]".

In the Bcpl version of the microcode, subroutines are called by executing undefined opcodes that trap into the Ram. For example, the opcode for SilentBoot is 63000B. Arguments are passed in accumulators 0, 1, and 3, and the result (if any) is returned in AC0.

A Bcpl-callable procedure is available to do this. It is contained in the file CallRam.asm, and is called as follows:

```
result = CallRam(opcode, ac0, ac1, ac3)
```

For example:

```
CallRam(63000B, 177376B)  // equivalent to Mesa's SilentBoot[177376B]
```

The following table summarizes the Ram subroutine names and the corresponding Mesa Ram entry points and Bcpl trap opcodes for all the routines presently defined in the microcode.

Name	*Mesa entry point*	*Bcpl opcode*
SilentBoot	1400B	63000B
EnableEIA *(EIA board)*	1401B	63400B
SetLineTab *(CommProc)*	1401B	63400B
PupChecksum	1402B	64000B
ProcessEIA *(EIA emulator version only)*	-----	65400B
ChangeControlReg *(CommProc)*	1403B	65400B
SetConPtr *(1822 version only)*	1420B	-----
PREncrypt *(1822 version only)*	1421B	-----

Initialization

Prior to operating the communication hardware or Gateway microcode, the software must perform a great deal of initialization.

Microcode initialization

First, the microcode must be loaded. There are a large number of possible microcode configurations, depending on whether Mesa or Bcpl is being used, what devices are to be controlled, and how they are to be operated. Packed Ram images for seven standard configurations are available in the following files, to be found on [Ivy]<Portola>. Other configurations can easily be constructed; see the "Packaging and other facilities" section for information.

MesaGateEIA.br	Mesa, EIA board.
MesaGateEIAChain.br	Mesa, EIA board, and up to 3 Ethernet board to be operated with chained input buffers.
BcplGateEIA.br	Bcpl, EIA board.

BcplEmuEIA.br	Bcpl, EIA board, emulator-level processing. (See the "Timing information" section for an explanation of the circumstances under which this version is required.)
MesaGateCP.br	Mesa, CommProc
MesaGateCPChain1822.br	Mesa, EIA board, up to 3 Ethernets with input chaining, and an Alto-1822 interface.
BcplGateCP.br	Bcpl, CommProc

The Boot Locus Vector should be set to a value that will cause the appropriate tasks to be run in the Ram, and the Alto should be "silently" booted. This may be accomplished using the SilentBoot Ram subroutine, presented previously. The correct Boot Locus Vector value for each Ram image is contained in word zero of that Ram image; see the "Packed Ram" and "LoadRam" package documentation for further details on the format of Ram images. The Bcpl LoadRam procedure obtains the Boot Locus Vector from the Ram image automatically (if the *boot* argument is true).

Caution: the silent boot causes an extended memory Alto's bank registers to be reset to zero. Therefore, the code that executes the silent boot had better be running in bank zero, or else it will perform an unexpected non-local goto. If any other tasks (e.g., the display) are running in extended memory, their bank registers will need to be reset also. XMesa programmers beware.

Determining which lines exist

To determine whether any EIA boards exist, simply store 140000B into location 177700B and then read location 177704B. If the result is nonzero, at least one EIA board exists. After reading 177704B, you should read 177703B once; otherwise, you may leave the EIA board in a bad state. You must do all this *before* enabling the EIA microcode. Note that the EIA board for line zero must be present in order for any other line to work.

In order to determine which EIA lines are present, it is necessary to complete the initialization of the Line Table (below) and enable the EIA microcode. Then zero the hardware status word in each LTE and poke each line by storing 14*x*000B into location 177700B, for *x*=0 to 7. Within a maximum of 608 (=2*8*38) microseconds, the status word for every line that exists will have become nonzero.

To determine which CommProc lines exist, store some nonzero value into location 177300B + 4**line*, for *line*=0 to 15. If the same nonzero value can be read back, the line exists. The nonzero value chosen should not cause Receive Enable (bit 8) or Send Enable (bit 9) to be set, since that would cause the CommProc microcode to be awakened.

Data structure initialization

Next, the main memory data structures must be initialized. LINTAB should be set up as documented previously, with initial states appropriate to the software interfaces to be used on each line.

The word at LINTAB-1 must contain a pointer to a 256-word table, CRCTAB, that assists in CRC computation. The software must initialize this table using the algorithm presented by the following Bcpl program:

```
for i = 0 to 255 do
  [
  let crc = 0
  let val = i
  for power = 0 to 7 do
    test (val & 1) eq 0
      ifso
        val = val rshift 1
      ifnot
        [
        crc = crc xor (120001B rshift (7 - power))
        val = (val rshift 1) xor 120001B
        ]
  CRCTAB ! i = crc
  ]
```

This table is usable by software as well as microcode. The following procedure, given a partial CRC and a new data character, returns the updated CRC:

```
let UpdateCRC(oldCRC, data) =
  (oldCRC rshift 8) xor CRCTAB ! ((oldCRC xor data) & 377B)
```

Hardware initialization

Next, the hardware for each line must be put into the correct initial state. For the EIA board, this is accomplished by storing a sequence of command words into main memory location 177700B, as described under "Control Word Formats" in the EIA interface documentation. For the CommProc, the control words at addresses 177300B + 4**line* + *n* (*n*=0, 1, 2) must be initialized as appropriate. The CommProc Receive Enable and Send Enable flags must not be set until after SetLineTab has been called (below).

The exact command words required depend on how the line is to be used and what software interface is to be employed. Typical command sequences are suggested later in the descriptions of specific software interfaces.

Enabling and disabling the microcode

EIA board: The microcode is controlled by bit 15 of the R-register R37: if this bit is set, the EIA microcode is permitted to run.

The microcode is enabled and disabled by a Ram subroutine, EnableEIA[LINTAB]. If LINTAB is nonzero, the microcode is enabled and LINTAB establishes the address of the Line Table. If LINTAB is zero, the microcode is disabled.

The software must turn the EIA off when quitting and when entering the debugger, and back on again when leaving the debugger. When quitting, it is necessary to execute SilentBoot[177776B] in order to get MRT out of the Ram.

CommProc: The Line Table address is established by calling the Ram subroutine SetLineTab[LINTAB], which has the same entry point as EnableEIA but does *not* perform the additional function of enabling or disabling the microcode.

Unfortunately, the CommProc provides no way to enable and disable the microcode as a whole in a manner that doesn't disturb the state of the hardware. Probably the best method of shutting off wakeups when entering the debugger is to execute the following for each line:

Read and save the contents of control word 0 (by fetching 177300B + 4**line*).

Turn off Send Enable and Receive Enable by executing ChangeControlReg[4**line*, 300B].

When leaving the debugger, restore the saved value of control word 0 by storing it directly into

177300B + 4**line*.

Of course, executing SilentBoot[177776B] still suffices to reset everything when quitting.

Additional initialization for Emulator-level EIA processing

There is a special version of the Bcpl EIA microcode that is functionally equivalent to the standard one but performs EIA processing in the Emulator task rather than the Memory Refresh Task. The justification for this variant is described later in the "Timing information" section. Here we describe the additional initialization that this microcode requires.

When the Memory Refresh Task discovers that EIA service is required, it initiates an interrupt on channel 16B, which has highest priority except for the parity error channel. The interrupt routine for this channel (pointed to directly by the channel's dispatch word, which is location 502B) is expected to consist entirely of the following two instructions:

```
ProcessEIA                  ; opcode 65400B
bri                         ; 61002B
```

The ProcessEIA instruction calls a Ram subroutine that performs the necessary EIA processing. ProcessEIA takes no arguments and returns no results.

Before enabling the EIA microcode, it is necessary to set up this interrupt routine and to turn on channel 16B (by "or"ing 2 into Active = 453B). This may *not* be done using the Bcpl Interrupt package, since that package performs a context switch and a Bcpl procedure call on every interrupt, which is definitely not what you want. Also, the software must reserve channel 16B early enough during initialization so that it will not be usurped by other uses of the Interrupt package.

General operation

EIA board

The EIA interface requests service whenever the INTP condition is true (see "Status Word Format" in the EIA documentation). The microcode examines the status word of the requesting line to determine what to do. If the RDA (Receive Data Available) flag is set, it performs input processing on the line (i.e., it dispatches on the line's input state). Similarly, if TBMT (Transmit Buffer Empty) is set, it does output processing.

Regardless of whether or not RDA or TBMT is set, the microcode stores the hardware status in word 1 of the Line Table Entry. This permits the software to monitor the status of an otherwise idle line (e.g., to notice Send Indicator). Note, however, that not all changes in status cause the EIA interface to request service. In particular, the dropping of SI or CD causes a request but the raising of those signals does not. Therefore, to guarantee that the status is up-to-date, the software should poke the interface using the command that forces a line to request a wakeup, then wait at least 38 microseconds before reading the status word. Poking the interface is accomplished by storing the command 14*x*000B (where *x* is the line number) into location 177700B.

CommProc

Service requests in the CommProc cause a dedicated task to be awakened. Requests are generated only for transmission or reception of data bytes, not by changes of control or status levels. The CommProc task dispatches on line number, I/O direction, and line state in a manner similar to the EIA microcode.

Unlike the EIA microcode, the CommProc microcode does *not* store hardware status into word 1 of the Line Table Entry, since the hardware status register may be interrogated directly by software. At any time, the status word for *line* may be read from memory location 177300B + 4**line* + 3.

A Ram subroutine is provided by means of which the software may change selected bits in a control register without disturbing the other bits. This operation is atomic with respect to CommProc task execution.

```
ChangeControlReg: PROCEDURE[lineTimes4: CARDINAL, changeMask: WORD]
```

LineTimes4 should be four times the line number to be affected. The low-order two bits identify the control word to be manipulated. Control word zero is the only one for which this operation is likely to be useful. The control register bits corresponding to ones in bits 1 through 15 of changeMask are set to the value in bit 0 of changeMask. The remaining bits are not disturbed. Note that bit 0 is not used in any control register.

General

In general, the microcode performs only data transfer operations between a line and memory, and does not touch the interface control registers. This means that it is the software's responsibility to initialize the interface appropriately (synchronous or asynchronous, full- or half-duplex, baud rate, etc.) and to monitor status bits as required.

In the descriptions of the block-mode software interfaces, references are made to the operation of appending an LCB to a chain. Care must be taken to do this in a race-free fashion. The following Bcpl procedure accepts *head*, a pointer to the head of an LCB chain (within a Line Table Entry), and *lcb*, a pointer to an LCB to be appended to the chain.

```
let AppendLCB(head, lcb) be
  [
  lcb ! 0 = 0
  let p = head
  while p ! 0 ne 0 do p = p ! 0
  p ! 0 = lcb
  if head ! 0 eq 0 then if lcb ! 2 ge 0 then head ! 0 = lcb
  ]
```

Note that it is the software's responsibility to keep track of LCBs it has put on queues, since the microcode does not hand back pointers to the LCBs that it is finished with.

To reset a line that has timed out or is otherwise in need of reinitialization (e.g., when switching to a different software interface), the following procedure is offered:

Set the line state to zero.

Remove the LCB, set the LCB pointer to zero, and reset control registers as required.

Set the line state to the initial state of the appropriate software interface.

Restart the hardware.

BiSync software interface

The microcode implements a subset of the standard BiSync protocol, namely the part dealing with transparent transmission of data frames. Such frames are used to encapsulate Pups transmitted over synchronous lines among Nova and Alto Gateways. The microcode does not implement control messages, ACKs, or other aspects of full BiSync. Indeed, it is not presently usable as part of a software implementation of full BiSync, though it could be made so with some effort.

LCB format

The standard 4-word LCB is extended to contain the following additional information:

Word 4*	Byte count and half-word selection. Bit 0 = 0 if the next byte to be transferred is the left byte in a word, 1 if the right byte. Bits 1-15 contain the number of bytes remaining.
Word 5*	Address of word containing next byte to be transferred.
Word 6*	Partial CRC. The software must initialize this to zero before appending the LCB to a chain.

Data bytes are packed two per word, left-to-right, consistent with Mesa, Bcpl, and Pup conventions.

Hardware initialization: EIA board

The following control sequences are suggested to initialize the EIA line appropriately for synchronous communication. These are precisely the settings required for the Pup Gateway application.

10*x*400B	(where *x* is the line number) INITIF with DS=1 (communicating with Data Set), HD=0 (full-duplex), STD=0 (no secondary channel). The Baud rate is normally irrelevant when operating in synchronous mode because the clock is provided by the modem. If it is desired to connect two Altos with a direct synchronous line (no modems), exactly one of the Altos should set DS=0 (communicating with Data Terminal) and select an appropriate Baud rate. When operating in half-duplex mode, it is the software's responsibility to manage the S (sending) bit appropriately.
13*x*200B	RESET with RRT=1 (reset the USRT chip).
11*x*700B	INITRT with NDB=3 (8 data bits per character), NPB=1 (no parity bit), A/S=0 (synchronous). POE and NSB are irrelevant.
12*x*026B	INITRG with F/S=0, Data=26B (SYN byte for receiver sync character register).
12*x*777B	INITRG with F/S=1, Data=377B (all-ones byte for transmitter fill character register).

After the hardware and data structures are initialized and the EIA microcode enabled, synchronous input should be initiated (just once) by issuing a "Restart Receiver" command (store 13*x*100B in location 177700B). Thereafter, the EIA microcode tracks the incoming data stream, regardless of whether or not an LCB has been provided by the software.

Hardware initialization: CommProc

CommProc control word *n* is initialized by storing into location 177300B + 4**line* + *n*, for *n*=0, 1, 2.

word 0 _ 60B	Data Terminal Ready, Request to Send. For test purposes, setting Wrap Enable (bit 13) may be useful.

word 1 _ 302B — Word Select 3 (8-bit bytes), Clock Select 0, Mode Select 2 (synchronous). Clock Select is relevant only when running in asynchronous mode or when Wrap Enable is true. The baud rate selected by any of the four values of Clock Select is determined by jumpers on the backplane.

word 2 _ 26B — SYN byte for receiver sync character recognition.

After the hardware and data structures have been completely initialized, input should be initiated (just once) by setting the Receive Enable bit in control word 0. This is most easily accomplished by ChangeControlReg[4**line*, 100200B]. Thereafter, the CommProc microcode tracks the incoming data stream, regardless of whether or not an LCB has been provided by the software.

Input

To enable reception of a packet, the software should set up an LCB as described previously. Word 5 should be initialized with a pointer to a buffer and word 4 with its length in bytes. Note that words 2 (ending microcode status) and 6 (partial CRC) must be initialized to zero. The LCB should then be appended to the line's input LCB chain.

When a packet has been received, the microcode stores status as described previously, requests interrupts as specified by word 1 of the LCB, and removes the LCB from the chain. Bits 1-15 of word 4 (byte count) indicate the number of unused bytes remaining in the input buffer.

Microcode status codes peculiar to BiSync input are:

3 — Input line control error (byte after first DLE wasn't STX).

4 — Input buffer overflowed. Assuming the buffer really was large enough for the largest expected packet, an overflow most likely means that the receiver lost character sync with the incoming data. Therefore, the microcode drops sync and restarts the receiver when this happens.

5 — Input line control error (in body of packet, byte after DLE wasn't DLE, SYN, or ETX).

6 — Input CRC error.

The receiver is then restarted (searching for character sync), regardless of whether or not there is another LCB on the chain. If a packet starts to arrive and no LCB is set up, the microcode throws the data bytes away but still maintains packet frame sync. If a new LCB is provided by the software while the microcode is in the middle of receiving a packet, the remainder of the packet is stored in the buffer. The ending status will most likely indicate a CRC error in this case, since the CRC is computed only over the data actually stored in the buffer.

To assure correct reception of every incoming packet, therefore, it is advisable that the software attempt to maintain at least two LCBs on the input chain at all times.

Output

To transmit a packet, the software should set up an LCB as described previously and append it to the line's output LCB chain. It should then initiate transmission in one of the following ways:

EIA board: Poke the interface by storing 14*x*000B into location 177700B.

CommProc: Set Send Enable by calling ChangeControlReg[4**line*, 100100B].

This ensures that the microcode will wake up and notice that a new LCB has been set up. Note that it is OK to poke the interface in this way even when it is already running; ongoing data transfers are not disturbed.

When the packet has been transmitted, the microcode stores status and requests interrupts as described previously and removes the LCB from the chain. If there is another LCB on the chain, transmission of that packet begins immediately. Otherwise, the transmitter becomes idle (the EIA

board transmits continuous fill characters; the CommProc's Send Enable flag is reset).

BiSync data frame format

A frame consists of the following sequence of bytes:

SYN SYN ... SYN DLE STX ... transparent data ... DLE ETX CRC CRC

The frame begins with at least two SYN bytes, which the receiver searches for in order to establish byte synchronization. The sequence DLE STX signals the start of the data portion of the frame.

Within the data portion, all bytes are treated as literal data except DLE, which is an escape character. A literal data byte whose code corresponds to DLE is transmitted by doubling it, i.e., by sending DLE DLE. If the sequence DLE SYN appears, both bytes are ignored. Some synchronous interfaces, though not the Alto EIA, automatically transmit DLE SYN when a transmit data-late condition occurs.

The end of the data packet is indicated by the sequence DLE ETX. Following this are two bytes containing the 16-bit CRC, transmitted low-order byte first. The algorithm used is the industry-standard CRC-16, computed over the transparent data bytes and the ETX. When the sequence DLE DLE appears, only one of the DLEs contributes to the CRC. When DLE SYN appears, neither byte is included.

While no frame is in progress, continuous fill characters (all ones) are transmitted.

Uninterpreted block transmission interface

This software interface permits transmission of arbitrary blocks of characters, with the microcode performing no interpretation whatever. It is useful primarily in conjunction with software implementation of more elaborate synchronous protocols, such as full BiSync.

Note that there is no corresponding interface for reception of uninterpreted blocks, since there is no way for the microcode to determine when a frame ends without interpreting the protocol. For input, the character-at-a-time (ring buffer) interface is probably the most appropriate.

This software interface is not included in the MesaGateCPChain1822 and MesaGateEIAChain configurations.

LCB format

Three words are added to the basic 4-word LCB:

Word 4*	Byte count and half-word selection. Bit 0 = 0 if the next byte to be transferred is the left byte in a word, 1 if the right byte. Bits 1-15 contain the number of bytes remaining.
Word 5*	Address of word containing next byte to be transmitted.
Word 6	Control command to be issued after transmission of block has been completed.

The block pointed to by the LCB is expected to contain two data bytes per word, packed left-to-right.

Operation

To transmit a packet, the software should set up an LCB and append it to the line's output LCB chain. It should then poke the interface in the manner described above under "BiSync Software Interface: Output".

When the packet has been transmitted, the microcode first issues the control command in word 6 of the LCB by storing it in location 177700B *(EIA board)* or 177300 + 4**line (CommProc)*. The intended use of this feature is to turn around a half-duplex synchronous line at the end of transmission. This word should be zero if no control command is to be issued.

The microcode then stores status and requests interrupts as described previously and removes the LCB from the chain. If there is another LCB on the chain, transmission of that packet begins immediately; otherwise, the transmitter becomes idle.

Note that the microcode does not add any SYN characters or other framing at either the beginning or the end of the block. If such characters are required, the software must include them in the data block.

Character-at-a-time software interface

This interface implements character-at-a-time reception and transmission via ring buffers. This is the most suitable interface for asynchronous lines. It can also be used for synchronous lines when it is desired to implement the synchronous line protocol in software.

This software interface is not included in the MesaGateCPChain1822 and MesaGateEIAChain configurations.

LCB format

Unlike the block-oriented software interfaces described previously, this interface requires a single LCB that is used indefinitely. The standard 4-word LCB is extended by an additional 4 words that describe a ring buffer. The LCB link (word 0) *must* contain a pointer to itself.

Word 4	Address of first word of ring buffer.
Word 5	Address of last word of ring buffer +1 (i.e., the first address plus the length of the buffer).
Word 6*	Address of next word to be read from the buffer.
Word 7*	Address of next word to be written in the buffer.

Data is stored in the buffer one byte per word, right-justified. The input microcode returns status in the left byte *(EIA only)*, and the output microcode expects the left byte to be zero.

The reader and writer of the ring buffer must obey certain conventions to ensure race-free operation:

> There must be only one reader and one writer, though the reader and writer need not be interlocked in any fashion.
>
> An empty buffer is represented by read pointer equal to write pointer. A full buffer is represented by read pointer equal to write pointer +1 (modulo wrap-around). Note that under these conditions, the word pointed to by the write pointer is not being used.
>
> The reader should read the data from the buffer before incrementing the read pointer.
>
> The writer should write the data into the buffer before incrementing the write pointer.

These conventions are used by the microcode. The ring buffer format and conventions just described are also the ones used by the Bcpl Ring Buffer package (word version).

Input

If an EIA board is operating in synchronous mode, input must first be initiated (just once) by issuing a "Restart Receiver" command (store 13*x*100B in location 177700B). A CommProc line must be started by ChangeControlReg[4**line*, 100200B], whether it is running in synchronous or asynchronous mode.

Whenever a character arrives, the microcode attempts to write the data into the input ring buffer. It then initiates an interrupt as specified by the interrupt mask in the LCB (if it is nonzero). *EIA only:* the microcode stores the rightmost 8 bits of the current hardware status into the left byte of each word at the same time it stores the data into the right byte.

If the ring buffer is full when a new character arrives, the microcode posts an error code of 2 in the LCB and throws the character away. Hardware error conditions will cause an error code of 1 to be posted, as described previously. The microcode never zeroes the microcode status word. It is the software's responsibility to sample the status occasionally and zero it after noticing an error. Note that there is no way to determine which character caused the error.

To read the next character, the software should first wait until the ring buffer is not empty (the RingBufferEmpty procedure in the Bcpl Ring Buffer package is usable for this purpose). It should then read the data byte from the buffer and increment the read pointer. The following Bcpl code illustrates the required operations, which are similar to the ones used by the Bcpl Ring Buffer package.

```
let readPointer = lcb ! 6
while readPointer eq lcb ! 7 do [ ... ]  // buffer empty
let data = readPointer ! 0
readPointer = readPointer+1
if readPointer eq lcb ! 5 then readPointer = lcb ! 4  // wrap around
lcb ! 6 = readPointer
```

Output

To transmit a new character, the software should first wait until the ring buffer is not full (the RingBufferFull procedure in the Bcpl Ring Buffer package is usable for this purpose). It should then store the data byte in the word pointed to by the write pointer, then increment the write pointer. Finally, it should poke the line (to ensure that the microcode wakes up and notices the new character), as described previously under "BiSync Software Interface: Output". It is actually necessary to poke the line only if it was idle at the moment the new character was written into the buffer. This may be determined by checking to see if, after writing the new character, there is exactly one character in the buffer. However, the cost of servicing an unnecessary wakeup is only 4 microseconds, whereas the cost of deciding whether or not to issue the wakeup is undoubtedly much greater, so in practice it is probably better just to poke the interface every time.

Whenever the microcode finishes transmitting a character, it initiates an interrupt as specified by the interrupt mask in the LCB (if it is nonzero).

The following Bcpl code illustrates the operation of writing a new character into the ring buffer:

```
let writePointer = lcb ! 7
writePointer ! 0 = data
writePointer = writePointer+1
if writePointer eq lcb ! 5 then writePointer = lcb ! 4  // wrap around
while writePointer eq lcb ! 6 do [ ... ]  // buffer full
lcb ! 7 = writePointer
```

Note *(CommProc only):* When the microcode discovers the ring buffer to be empty, it resets Send Enable to dismiss the data request. A consequence of this is that there is no way to detect a transmit underflow condition for a line being operated in synchronous mode.

Timing information

The following rough calculations attempt to estimate the fraction of the Alto consumed by the Gateway microcode when driving lines at various data rates.

EIA microcode timing

	BiSync	*Uninterpreted block*	*Character-at-a-time*
Alto cycles/byte	90	65	73
Microseconds/byte	15	11	12
Fraction of Alto cycles consumed running one line, one direction, at			
9.6 KB	1.8%	1.3%	1.4%
19.2 KB	3.6%	2.6%	2.9%
56 KB	10.5%	7.7%	8.4%

Remember, of course, that the percentage figures refer to the raw Alto microinstruction execution rate, with no other overhead. A full-screen display consumes about 60% of the Alto's cycles.

An orthogonal problem is that of MRT task latency and overrun. If the EIA microcode runs for longer than the MRT wakeup interval (38 microseconds), the next wakeup will be missed. The loss of MRT wakeups has several noticeable effects, the most serious of which are that the cursor jiggles vertically and the real-time clock loses time.

The EIA microcode is careful to service at most one character per run of MRT, even if several lines are requesting service simultaneously. The worst-case running time of the EIA microcode is about 20 microseconds. Therefore, in the absence of interference from higher-priority tasks, MRT should never overrun its next wakeup.

Unfortunately, when the display is running, it consumes approximately 60% of the Alto's cycles, at a priority level higher than MRT. The MRT wakeup interval is the same as the display horizontal scan interval. Therefore, if the Alto provides data for the entire scan line, 60% of the machine during that interval is taken up by the display and only 40% is left for MRT and lower-priority tasks. This can cause the EIA microcode's worst-case running time of 20 microseconds to be stretched to nearly 50 microseconds, easily overrunning the MRT wakeup interval.

Therefore, to prevent MRT overruns, it is advisable either to turn off the display entirely or to limit the width of the display bit map. For example, a half-width display will consume only 30% of the Alto during each scan line, probably leaving enough cycles for MRT and the EIA microcode. Note that use of the horizontal tab feature is nearly as costly in Alto cycles as is providing real data for the same portion of the scan line.

The disk word task is also higher-priority than MRT, but it consumes only about 15% of the Alto so is less likely to cause MRT overruns. The Ethernet task is lower-priority than MRT so will not interfere with it at all.

For applications in which it is unacceptable to turn off the display, a special version of the EIA microcode exists in which MRT, rather than doing EIA processing itself, initiates an interrupt on a high-priority channel and lets the Emulator task do the work. This eliminates the MRT overrun problem entirely, but at a cost of nearly 100 additional microinstructions executed per byte. Only a Bcpl version of this special microcode is provided at present. The software must set up an interrupt routine, as described under "Initialization".

CommProc microcode timing

	BiSync	*Uninterpreted block*	*Character-at-a-time*
Alto cycles	72	42	46
Microseconds/byte	12	7	8
Fraction of Alto cycles consumed running one line, one direction, at			
9.6 KB	1.4%	0.8%	1.0%
19.2 KB	2.9%	1.7%	1.9%
56 KB	8.4%	4.9%	5.6%

The CommProc task is lower-priority than MRT and other tasks sensitive to latency, so the latency considerations presented above do not apply.

Packaging and other facilities

The Gateway microcode is packaged to include all the microcode services required by Alto Gateways. However, the microcode source consists of a number of more-or-less independent modules, stored in a single dump-format file [Ivy]<Portola>GatewayMc.dm.

All the standard Ram images are obtained by compiling one of the top-level Mu source files, which inserts (using the "#" facility) various assortments of the other files to produce a given configuration. New configurations can easily be constructed, subject to Ram space limitations and R- and S-register availability. (The MesaCPChain1822 configuration fills the entire Ram and uses all the registers.)

All the Mesa Ram images include the Ram portion of the XMesa (extended memory) microcode. Therefore, if Rom1 contains XMesa microcode also, it is possible to run XMesa-based software. The presence of the XMesa microcode in the Ram does not interfere with operation of standard Mesa software in machines with the standard Mesa microcode in Rom1.

MesaGateEIA.mu	Top-level source for Mesa, EIA board.
MesaGateEIAChain.mu	Top-level source for Mesa, EIA board, and up to 3 Ethernet boards with chained input buffers (see below).
BcplGateEIA.mu	Top-level source for Bcpl, EIA board.
BcplEmuEIA.mu	Top-level source for Bcpl, EIA board, emulator-level processing.
MesaGateCP.mu	Top-level source for Mesa, CommProc.
MesaGateCPChain1822.mu	Top-level source for Mesa, CommProc, up to 3 Ethernet boards with chained input buffers, and an Alto-1822 interface (see below).
BcplGateCP.mu	Top-level source for Bcpl, CommProc.
AltoEIA1.mu	Main EIA board driver, EIA subroutines, and BiSync software interface.
AltoEIA2.mu	Uninterpreted and character-at-a-time EIA software interfaces.
CommProc1.mu	Main CommProc driver, CommProc subroutines, and BiSync software interface.

CommProc2.mu	Uninterpreted and character-at-a-time CommProc software interfaces, and CommProc interval timer.
EIAMRT.mu	Modified version of the Memory Refresh Task, required for the EIA microcode.
EIADispMRT.mu	Version of EIAMRT.mu required when GateDisplay.mu is included (see below).
EmuEIAMRT.mu	Version of EIAMRT.mu required when using BcplEmuEIA.mu.
CPMRT.mu	Modified version of the Memory Refresh Task, to accompany the CommProc microcode. The CPMRT changes are a subset of the EIAMRT changes, including abolition of the software Interval Timer and elimination of the wasteful use of R-register CLOCKTEMP. The CommProc microcode itself does not require MRT to be modified. CPMRT is so named simply to distinguish it from EIAMRT.
CPDispMRT.mu	Version of CPMRT.mu required when GateDisplay.mu is included (see below).
GateDisplay.mu	Microcode for the Alto display and cursor, rewritten to use two fewer R-registers than the standard microcode. This is required when using Mesa and controlling extra devices requiring R-registers, such as additional Ethernet controllers. The registers saved are 21B and 26B. All the display-related tasks (MRT, DWT, CURT, DHT, and DVT, tasks 10B through 14B respectively) must be started in the Ram, and special variants of the MRT microcode (EIADispMRT.mu, CPDispMRT.mu) are required.
ExtraEther1.mu	Microcode for one additional Ethernet interface (see below).
ExtraEther2.mu	Microcode for a second additional Ethernet interface. This is not presently assembled in any of the standard Ram images.
ExtraEther.mu	A top-level source file demonstrating the use of ExtraEther1.mu and ExtraEther2.mu. This file itself is not used in the Gateway microcode package.
ChainEther0.mu	Microcode for controlling the standard Ethernet interface using chained input buffers (see below).
ChainEther1.mu	Chained input microcode for the first additional Ethernet interface.
ChainEther2.mu	Chained input microcode for the second additional Ethernet interface.
Mesa1822.mu	Microcode for controlling an Alto-1822 interface (used for connecting to the Arpanet or the Arpa Packet Radio network). Documentation may be obtained from Larry Stewart (user LStewart).
PREncrypt.mu	Microcode for encrypting data to be sent through the Packet Radio network.

Pup checksum computation

A Ram subroutine is included for efficient computation of the Pup software checksum:

```
PupChecksum:
    PROCEDURE[initialSum: WORD, address: POINTER, count: CARDINAL]
    RETURNS[resultSum: WORD]
```

The address should be a pointer to the first word of a Pup and the count the number of words in the Pup, exclusive of the checksum word (i.e., (Pup.length-1)/2). initialSum must be zero.

Timing for this subroutine is 9 Alto cycles per word, or about 425 microseconds for a maximum-length Pup.

Interval timer (CommProc only)

The CommProc hardware includes a task-driven interval timer. The Gateway microcode package supports this; however, the interval timer is not required by present Pup Gateway applications.

If the interval timer is to be used, the software must reserve and initialize three additional words just below the beginning of the Line Table. These are:

Word -4	Interrupt bit mask. An interrupt is initiated on the designated channel(s) each time the interval timer expires.
Word -3	Timer interval, in units of 625 microseconds.
Word -2	Interval Timer counter. This should be initialized to the timer interval minus one, and is thereafter maintained by the microcode.

The interval timer is enabled by storing 100000B in location 177400B, and disabled by storing 0 into the same word. While the hardware is enabled, the microcode issues one interrupt per timer interval.

If the interval timer is being used, one should take care to disable it before entering the debugger.

Note: the interval timer microcode is not included in the MesaGateCPChain1822 configuration.

Additional Ethernet controllers

The Gateway microcode package includes microcode for controlling up to two additional Ethernet interfaces. For details, consult the memo *How to install extra Ethernet interfaces in an Alto*. The task number, SIO control bit, and control block address assignments are as follows:

Controller	*Task*	*SIO bits*	*Control block*
Standard Ethernet	7	14, 15	600B - 611B
First extra Ethernet	2	12, 13	630B - 641B
Second extra Ethernet	1	10, 11	642B - 653B

Source files ExtraEther1.mu and ExtraEther2.mu contain microcode for controlling the first and second additional Ethernets. This microcode operates identically to the standard Ethernet microcode except for task number, SIO bits, and control block address. All the standard Ram images except MesaGateEIAChain and MesaGateCPChain1822 include ExtraEther1 but not ExtraEther2.

In order to operate a second extra Ethernet interface in conjunction with Mesa (which uses nearly every available R- and S-register), it is necessary to free up two additional R-registers. This may be done by replacing the standard display and cursor microcode by the code contained in GateDisplay.mu, which uses two fewer R-registers.

The second Ethernet interface conflicts with the CommProc interval timer over use of task 1. Since the interval timer is not used in the Pup Gateway application, it suffices to disconnect the interval timer task wakeup and active signals (see "Hardware requirements" at the beginning of this document).

Another version of the Ethernet microcode exists that provides a somewhat different software interface to the Ethernet. The major difference is that input is specified by means of chains of command blocks that are followed automatically by the microcode, and the microcode also handles most of the decisions about switching between input and output. This permits the software to keep the Ethernet receiver turned on a greater fraction of the time than with the standard microcode (which requires the software to restart the interface after every packet sent or received); we expect this to improve the performance of Alto Gateways.

Files ChainEther0.mu, ChainEther1.mu, and ChainEther2.mu contain the microcode for the standard Ethernet interface and the first and second extra interfaces, respectively. The standard configurations MesaGateEIAChain and MesaGateCPChain1822 include all three. Programming information may be obtained from Hal Murray.

Due to Ram space limitations, the MesaGateEIAChain and MesaGateCPChain1822 configurations do not include the uninterpreted block and character-at-a-time software interfaces or the microcode for controlling the CommProc interval timer. These are not required in the Pup Gateway application.