HOST/CC AND CC/CC ASYNCHRONOUS CONTROL LINE DRIVER IN THE MIMICS NETWORK

by

ERWIN LYNN BEHME

B.S., Kansas State University, 1975

-------------------

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977

Approved by:

[Signature]
Major Professor
# TABLE OF CONTENTS

## CHAPTER 1

**Introduction** ....................................................... 1
  1.1 Structure of Report ........................................... 1
  1.2 MIMICS Network Architecture .............................. 2

## CHAPTER 2

**Asynchronous Lines** ............................................. 11
  2.1 Motivation for use of Asynchronous Lines .............. 11
  2.2 Characteristics of Asynchronous Lines ................. 12
  2.3 Interdata PASLA ............................................. 17
  2.4 Example of a CRT Driver Using a PASLA ................. 23

## CHAPTER 3

**Asynchronous Control Line Driver** –
  an Assembler Version ............................................ 27
  3.1 Interrupt Structure .......................................... 28
  3.2 Relationship to Asynchronous Processes ............... 29
  3.3 Module Layout ................................................ 32
  3.4 ACLDR Entry Points .......................................... 34
  3.5 Interrupt Service Routines ................................. 39
  3.6 Notes on ACLDR Operation ................................... 41

## CHAPTER 4

**Asynchronous Control Line Driver** –
  a Concurrent PASCAL Version .................................... 42
  4.1 IO_MACHINE .................................................. 42
  4.2 Example of a CRT Driver Using a PASLA ............... 46
  4.3 Relationship to Asynchronous Processes ............... 49
  4.4 ACLDR Entry Points .......................................... 51

## CHAPTER 5

**Summary** ........................................................ 52
  5.1 Comparison of Assembler and PASCAL Versions ....... 52
  5.2 Worked Completed ........................................... 54
  5.3 Extensions ................................................... 54

**BIBLIOGRAPHY** .................................................. 55

**APPENDICES**

APPENDIX A - IO_MACHINE Instructions and Errors ........... 56
APPENDIX B - ACLDR Assembler Code .............................. 63
APPENDIX C - ACLDR Concurrent PASCAL Code .................. 77
THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE. THIS IS AS RECEIVED FROM CUSTOMER.
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>General Configuration of MIMICS</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Possible Machine Configuration in a Cluster</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Message Data Flow in MIMICS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. User Tasks with a Common CC</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>b. User Tasks in Same Cluster, but Different CC's</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>c. User Tasks in Different Clusters</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Transmission of the Character &quot;S&quot; on an Asynchronous Line</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Terminal/Computer Connection Using Modems</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Connection of Two PASLAs for MIMICS</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>STATUS BYTE</td>
<td>19</td>
</tr>
<tr>
<td>2.5</td>
<td>COMMAND BYTES</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Format of an SCB Being Transmitted</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Transfer of Control Caused by an Interrupt From Device X'11'</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Relationship of ACLDR to Asynchronous Processes</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Modules of the ACLDR</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>IO_MACHINE STATUS WORD</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Relationship of ACLDR to Asynchronous Processes</td>
<td>50</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to thank Dr. Virgil Wallentine, my major professor, for suggesting this project and for his help and guidance. I would like to thank Drs. William Hankley and Myron Calhoun for their suggestions concerning the paper. Also, I would especially like to thank Dave Neal, Jim Ratliff, Gary Anderson, and Earl Harris for all the help they have given me. Finally, I would like to thank the Computer Science Department and the Computing Center for the use of their equipment in running the programs and printing the report.
CHAPTER 1
Introduction

The Mini-Micro Computer System (MIMICS) is being developed at Kansas State University as a network of mini and micro-computers for a distributed data base system. This network is designed to use mini and micro-computers in order to provide the maximum amount of computing power at a minimum cost. MIMICS functions can be at geographically dispersed locations or in clustered activities and only the speed at which these functions are accomplished is affected. The computers in the network are not limited to one manufacturer or type, therefore the links between them must be universal. One type of link used between the computers is an asynchronous line. This paper contains a description of the design and implementation of an Asynchronous Control Line Driver (ACLDR) in the MIMICS network for these lines. The driver handles the functions necessary for sending and receiving of control information between computers within a cluster of the network.

1.1 Structure of the Paper

The remainder of this chapter contains an overview of the MIMICS network architecture based on the description in reference WHA76. It also describes the function of an Asynchronous Control Line Driver (ACLDR) in this network. Chapter 2 presents the reasons for using asynchronous lines in MIMICS. In it are described the asynchronous lines in
general, and then specifically, Interdata's Programmable Asynchronous Single Line Adapter which is used to link two Interdata machines together for this report. Chapter 3 gives the description of the Asynchronous Control Line Driver as implemented using Interdata's Common Assembler Language. Chapter 4 presents the Concurrent PASCAL version of this driver which runs under a KERNEL on the Interdata 16 bit machines. It also describes a special entry point in that Concurrent PASCAL KERNEL which the driver uses. Chapter 5 gives a comparison of the two versions of the driver plus a summary of the work completed and some extensions to conclude the report.

1.2 MIMICS Network Architecture

This section presents an overview of the MIMICS network architecture. It is not intended to give a detailed description of the entire network, but just the necessary elements to give the reader an understanding of how the Asynchronous Control Line Driver functions in MIMICS. This overview is based on reference WHA76, which gives a much more complete look at the network.

The general configuration of the MIMICS network is shown in Figure 1.1. It consists of nodes, which are known as "clusters", connected by low speed synchronous communication paths. It is not necessary that the clusters be geographically remote from each other but they probably will be. Each cluster contains one or more central processing units (CPU's), all located "close" to each other
Figure 1.1

General Configuration of MIMICS
(on the order of tens to hundreds of feet). Localizing of these CPU's in a cluster provides for interconnection with high speed synchronous data paths (perhaps 5-10 Megabytes per second transfer rate).

This is the view of the network that the average user might get. The actual makeup of MIMICS is much more complicated. Figure 1.2 shows a possible machine configuration in a cluster. It contains three host CPU's and two Communication Controller (CC) CPU's. The double lines represent high speed data paths and the single lines represent low speed control lines. The circles on the single lines are the hardware interfaces which operate the lines. Most of the burden of network communication is given to the CC's to lessen the load on the host CPU's. Each host is connected to a CC via a KSUBUS. This bus provides high speed data transfers from the memory of one CPU to that of another on the same KSUBUS under control of local Data Movers (DM). It also allows this fast transfer of data from one KSUBUS to another KSUBUS in the same cluster through remote Data Movers. The job of a CC is to communicate with the hosts on its own KSUBUS and with the other CC's in the cluster to set up the Data Movers to accomplish this movement of data. One CC in each cluster is designated as the enter/exit CC and its function is to handle communication from its cluster to remote clusters across synchronous lines. Asynchronous lines carry control information between the different CPU's within the cluster.

When a CPU requests some data through its operating
system, the control computer associated with that host is contacted via the asynchronous line link. The request is then passed to the network message system which takes the necessary steps to find the requested data. There are many distributions of data and many ways of supplying that data to the requesting tasks.

1 - If the requested data exists in the host's memory or the memory of another host on the same KSUBUS, then the CC sets up a high speed copy of the data into the memory area of the requesting user task.

2 - If the data resides in the CC's memory then a similar high speed copy is executed.

3 - If the data is in the memory of a host in the cluster but not on the same KSUBUS, then the CC's associated with the hosts arrange for the transfer of the data using the Remote Data Movers that connect the two KSUBUS's.

4 - If the requested data exists in a remote cluster, it is transferred first to the enter/exit CC of the cluster where the data resides, then across a synchronous line to the enter/exit CC of the cluster of the requesting host, and finally to that host across the KSUBUS.

Of course, these are only a few of the possibilities, but they should give the reader a feel for the types of transfers that may be needed. These examples are also illustrated in Figure 1.3, which was reprinted, with permission, from reference WHA76.

There are two types of computer-to-computer connections using asynchronous lines in the MIMICS network. The first
type is the link between a host computer and its communication controller (CC). In Figure 1.3, the data requests are sent across these asynchronous control lines. The second type, which also uses asynchronous lines, is the connection between communication controllers in a cluster of the network. This type is shown in Figure 1.3b.

The requests for data that are sent across the control lines are set up in the Asynchronous Communication Processes. These processes then call a driver for the asynchronous lines to transfer the requests. The design and implementation of this Asynchronous Control Line Driver is presented in this paper.
Figure 1.3a

Message Data Flow in MIMICS:
User Tasks with a Common CC,
(Either same host or two hosts on same KSUBUS)
Figure 1.3b

Message Data Flow in MIMICS:
User Tasks in the same Cluster,
but not the same CC.
Figure 1.3c

Message Flow in MIMICS:
User Tasks in Different Clusters

-10-
CHAPTER 2

Asynchronous Lines

One major problem in designing a network is how to connect the computers together without having to build special interfaces between the machines. One method chosen to transmit control messages for the MIMICS network is to handle communication across asynchronous lines. This chapter contains the basis for this decision, the characteristics of asynchronous lines, and how they are used on the Interdata 16-bit machines.

2.1 Motivation for use of Asynchronous Lines

In an effort to minimize the amount of money and time spent to develop the MIMICS network, asynchronous lines were chosen to carry control messages between computers. This choice was made because "off-the-shelf hardware" for these lines can be used, they are basically universal, and they are relatively straight-forward to operate.

Most manufacturers of computers have designed and built asynchronous line interface hardware that is compatible with their machines, thus eliminating the need for the network developers to design, build, and debug them. This greatly reduces the time and money spent for linking the computers together.

RS-232C is the Electronic Industries Association interface standard for Asynchronous Lines. This is very important because a network can be made up of many different
kinds of computers and if the connections between them are similar, then the interface drivers which control these connections can be similar also. Because of the RS-232C standard, greater portability of the control line driver between machines may be obtained.

All asynchronous interfaces which are RS-232C compatible have common output signals of Data Terminal Ready, Request to Send, and one output data line, plus common input signals of Carrier, Data Set Ready, Clear to Send, Ring and one input data line. These wires are connected to different data sets according to the needs of that data set. The use of these signals for this report will be discussed in the following sections.

Since the interface hardware handles all functions of the actual transmitting and receiving of data across the wire, the device driver is only concerned with transferring data to and from the interface and handling interrupts from that interface. The handling of interrupts differs depending on the machine used, but the different hardware interfaces usually interrupt in like manner, i.e. an interrupt is generated when a character enters the interface and one is generated when a character leaves the interface.

2.2 Characteristics of Asynchronous Lines

Asynchronous lines are characterized by data transmission with an unknown length of time between characters. The data flow across the line is independent of the operation of the central processor and the timing needed
for the hardware interface is part of each character being transmitted. This timing is in the form of a start bit at the beginning of each character and one or two stop bits at the end of the character. These bits are attached to and deleted from the character by the hardware.

In Figure 2.1 we show how the character "S" is transmitted on an asynchronous line. The start bit informs the hardware interface of an incoming character. The 7-bit character is next. The 8th bit of this character is used for parity, followed by either one or two stop bits. In some interfaces, such as Interdata's PASLA [INT01], the value of this bit is determined and checked by the hardware for possible errors in transmission, but in others it must be software controlled. Certain interfaces, like the PASLA, also allow the character length to be as little as 5 bits long. The length of the character is either hardwired on the interface board or under program control.

Since asynchronous lines are predominantly used with modems or terminals, special signals known as Ring, Data Terminal Ready(DTR), Data Set Ready(DSR), Carrier(CARR), Request to Send(RTS), and Clear to Send(CTS) are used along with Transmit Data(TDATA) and Receive Data(RDATA). With a hook up between a terminal and a computer using modems and a telephone line similar to Figure 2.2, the connection and transfer of data from computer to the terminal would be obtained as described in INT05:

1. The operator dials the computer using the terminal's telephone.
Figure 2.1
Transmission of the Character "S" on an Asynchronous Line
Figure 2.2
Terminal/Computer Connection Using Modems
2. A ringing signal goes through the answering modem to the computer.

3. The computer returns a DATA TERMINAL READY (DTR) signal to the answering modem.

4. The answering modem sends a tone signal to the originating modem. The operator hears the tone and presses the data button of the originating modem.

5. The originating modem sends a DATA SET READY (DSR) signal to the terminal.

6. The answering modem sends a DATA SET READY (DSR) signal to the computer.

7. The modems are now in the data mode.

8. The computer can raise REQUEST TO SEND (RTS) which informs the answering modem it wants to transmit data.

9. The answering modem then responds with CLEAR TO SEND (CTS) and begins transmitting a carrier signal.

10. The originating modem detects CARRIER ON (CO) and informs the terminal that the computer wishes to transmit.

11. On detection of CTS, the computer can start sending data to the terminal.

12. The terminal receives the data as transmitted.

13. When the computer has finished sending all data, it drops REQUEST TO SEND (RTS).

14. The answering modem then stops sending carrier.

Steps 8 through 14 can be repeated (possibly with roles reversed) until either end terminates transmission.

For this report, the number of wires running between machines was kept to a minimum, therefore, several of these signals were not needed. The only ones used were Data Terminal Ready (DTR), Carrier (CARR), Transmit Data (TDATA),
and Receive Data (RDATA). In Figure 2.3 we show that DTR was used to activate the CARR signal of the other machine, and as with normal connections, TDATA was tied to RDATA of each machine. A common ground wire was also run between the machines.

![Diagram of PASLA connection](image)

**Figure 2.3**

Connection of two PASLAs for MIMICS

2.3 Interdata PASLA

For implementation on the Interdata machines the asynchronous lines are controlled by Interdata's Programmable Asynchronous Single Line Adapter (PASLA) [INT01]. The PASLA is a hardware device that is connected to the Multiplexor Bus and provides an interface between the Interdata computer and a number of data sets such as a CRT or a Modem. Depending on the type of data set, the PASLA can be wired for Half-duplex or Full-duplex operation. It can also be wired for two different baud rates, ranging from 47 Baud up to 16K Baud (the value of K is 1024). The choice of which rate to use plus several other options is determined by the programmer communicating with the PASLA.
He accomplishes this by using the machine language instructions Sense Status, Output Command, Write Data, and Read Data [INT02].

The Sense Status (SS or SSR) instruction is used to interrogate the status of the line and to test whether a character transfer was complete and correct. When an SS instruction is used, the device to which it is directed returns an 8-bit status byte. The format of the Sense Status instruction is:

\[ \text{SSR} \ R1,A2(X2) \quad \text{Sense Status} \]
\[ \text{SS} \quad R1,R2 \quad \text{Sense Status Register} \]

R1 contains an 8-bit device address and the returned status is placed in the second operand. The format of the status byte of the PASLA is shown in Figure 2.4 and explained below.
Figure 2.8

STATUS BYTE

<table>
<thead>
<tr>
<th>bits</th>
<th>OV</th>
<th>PF or CL2S</th>
<th>ERR</th>
<th>RCR</th>
<th>BSY</th>
<th>EX</th>
<th>CARR</th>
<th>OFF</th>
<th>RING</th>
</tr>
</thead>
</table>

**OV**
The Overflow bit is 1 when a character that was received by the PASLA was not read before another one was received. The last character to be received is the one that exists in the PASLA.

**PF**
The Parity Flag bit is 1 in read mode if received parity does not agree with the programmed parity. It is 1 in transmit mode if the Clear to Send (CL2S) signal is not being sent from the data set.

**ERR**
The Framing Error bit is 1 if an incoming character has no stop bit.

**RCR**
The Reverse Channel Receive bit is an option with some half-duplex data sets and is used to indicate the state (whether transmitting or receiving) of the data set.

**BSY**
The busy bit is 1 whenever a character is being transmitted or received and indicates that the processor cannot transfer data to or from the PASLA without mutilating the character being transmitted or received.

**EX**
The Examine bit is disabled on the transmit side of the PASLA in full-duplex mode and is set to one on the receive side if any one of OV, PF,
or FR ERR bits are set.

CARR_OFF The Carrier Off bit is one if the Carrier Signal is no longer being received.

RING The Ring bit indicates that a ring signal is coming from the data set.

In order to set up the PASLA, the programmer needs some way of communicating with it and this is done with the Output Command instruction. The format of the Output Command instruction is:

```
OC  R1,A2(X2)  Output Command
OCR R1,R2     Output Command Register
```

R1 contains an 8-bit device address and the second operand contains the command byte that is sent to the device. This command byte is of two forms which are shown in Figure 2.5 and explained below:

```
<table>
<thead>
<tr>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIS</td>
<td>EN</td>
<td>DTR</td>
<td>ECHO</td>
<td>PLEX</td>
<td>RCT</td>
<td>or</td>
<td>DTE</td>
</tr>
<tr>
<td>CMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTE 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>CLK</td>
<td>BIT SEL</td>
<td>STOP</td>
<td>BIT</td>
<td>PARITY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BYTE 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 2.5**

**COMMAND BYTES**

Bit 7 tells the PASLA which command is being sent to it and Bit 0 of COMMAND 2 is unused.

**PASLA COMMAND 1:**

-20-
DIS/EN: These two bits are separate for Transmit and Receive. They control the interrupts of the PASLA in the following manner:

- 0 0: No Change
- 0 1: Enable
- 1 0: Disable (interrupt queued)
- 1 1: Complement (Change State)

DTR: When the Data Terminal Ready bit is 1 then the DTR signal going out of the PASLA is made active.

ECHO-PLEX: When this bit is 1, any incoming character is transmitted back out.

RCT/DTB: Reverse Channel Transmit and Data Terminal Busy are options of certain data sets and are unused in this implementation.

TRANS LB: When on, the PASLA transmits a continuous space. It is unused used in this implementation.

WRT/RD: In order to get an interrupt when a character has been transmitted, this bit must be active otherwise the hardware holds the busy signal high until a character is received.

PASLA COMMAND 2:

CLK: The PASLA board can be wired for two different baud rates and if this bit is 0 then the lower of the two is chosen, otherwise the higher one is used.

BIT SEL: These two bits select how many data bits there are per character, not including parity.

<table>
<thead>
<tr>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Number of data bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

STOP BIT: When this bit is 0 then only one stop bit is transmitted, otherwise two stop bits are sent.

PARITY: Parity is set according to the following table:
Since the only control signal being sent out of the PASLA is Data Terminal Ready and the only one received is the Carrier signal, bits 3 through 6 of COMMAND 1 are not needed and are set to zero. After COMMAND 2 is issued to the PASLA to set the baud rate, the number of data bits, the number of stop bits, and the type of parity to match the two machines which are connected, COMMAND 1 is issued to turn DTR on and set the interrupt conditions for the PASLA. COMMAND 1 must be issued once for the receive side and once for the transmit side if both are to be used. If a character is to be sent to the PASLA then a Write Data instruction is issued. The format of the Write Data instruction is:

\[
\begin{align*}
\text{WDR} & \quad R1, A2(x2) \quad \text{Write Data} \\
\text{WD} & \quad R1, R2 \quad \text{Write Data Register}
\end{align*}
\]

where \( R1 \) contains an 8-bit device address and the second operand contains the 8-bit byte to be transferred to the PASLA. When the byte is transmitted out of the PASLA, an interrupt is generated by the device if enabled. If interrupts are not desired then the WD instruction can be issued and status of the PASLA can be sensed until the busy flag goes to zero before sending another character. This is known as a "busy wait loop". The disadvantage of using "busy wait loop" is that it ties up the processor until the character has been transmitted. A "busy wait loop" can also be used to wait on a character to be received by the PASLA.
or an interrupt can indicate to the processor that a character is available. When a character is in the PASLA, the program can get that character by issuing a Read Data instruction. The format of the Read Data instruction is:

\[
\begin{align*}
RD & \quad R1,A2(X2) \quad \text{Read Data} \\
RDR & \quad R1,R2 \quad \text{Read Data Register}
\end{align*}
\]

where \( R1 \) contains an 8-bit device address and the character is put into the second operand.

2.4 Example of a CRT Driver Using a PASLA

To give the reader a better feel for the operation of a PASLA, an example is given next. It shows the types of output commands that are issued to control the PASLA for a CRT in full duplex mode.

This PASLA is wired for full-duplex mode and a baud rate to match the speed of the CRT. For this example, the format of the characters transferred to and from the CRT will have one start bit, seven bits of data, odd parity, and two stop bits. Because of this configuration, the Output Command issued to set up the PASLA would use a COMMAND 2 byte of the value, binary '01101100'. This value can also be represented as hexadecimal '6C' (X'6C'). The next two Output Commands would be issued to set up the receive and the transmit sides of the PASLA to enable interrupts and activate Data Terminal Ready. The COMMAND 2 byte can be sent to either side but the COMMAND 1 byte has to be sent to both, if both are to be used. The receive side command byte would be a X'61' and the transmit side would be a X'63'.

-23-
After the PASLA has been initialized and a character is received from the CRT, an interrupt is generated if they are enabled. When the processor recognizes the interrupt, it sends control to a routine which usually reads the character out of the PASLA. When a character is to be transmitted, it is written out to the PASLA. The PASLA then sends it out on the asynchronous line. When the character leaves the PASLA, an interrupt is generated, if enabled, to inform a routine that another character can be sent.

The following program is a simple example which reads in ten characters from a CRT and then echoes all ten back out to the CRT, followed by a carriage return and a line feed. The main routine sets up the PASLA, waits for the message to be read in, starts the transmission of the message by an Output Command to enable interrupts, and then halts. The RCV routine is entered each time there is an interrupt on the receive side of the PASLA. It reads the character and puts it into a buffer. The XMT routine is entered whenever an interrupt occurs on the transmit side of the PASLA. The first interrupt is caused by Data Terminal Ready becoming active from the Output Command. Subsequent interrupts are caused by a character leaving the PASLA, indicating another one can be sent. When the buffer is empty, the Write Data instruction is skipped, terminating interrupts on the transmit side. The code for this example follows:
CRT DRIVER USING A PASLA

PROG= *NONE*

ASSEMBLED BY CAL/16 00-00

PAGE 1 00156139 00/00/00

```assembly
0001  2  TAKST  16
0002  3  RDEV  EQU  1
0003  4  XDEV  EQU  2
0004  5  FLAG  EQU  3
0005  6  BPTR  EQU  4
0006  7  CHAR  EQU  5
0007  8  CHK  EQU  6
0008  9  CPTR  EQU  7
0009 10  RDEV, RADDR
0010 11  LH  RDEV, RADDR
0011 12  XDEV, XADDR
0012 13  DC  RDEV, SETUP
0013 14  RDEV, ENRCV
0014 15  RDEV, CHAR
0015 16  DUMMY READ TO INSURE BUSY IS SET
0016 17  LHI  BPTR, BUFFER
0017 18  BHI  BPORT, BUFFND1
0018 19  LOOP  BPORT
0019 20  XDEV, ENRCV
0020 21  STOP  B  STOP
0021 22  LOOP
0022 23  #### RECV INTERRUPT SERVICE ROUTINE
0023 24  RCV
0024 25  DC  0,0.X'3000'
0025 26  OLD PSW AND NEW STATUS
0026 27  RDEV, (BPTR)
0027 28  STORE CHARACTER IN BUFFER
0028 29  BPORT+1
0029 30  INCREMENT BUFFER POINTER
0030 31  LPSW  RCV
0031 32  RETURN
0032 33  LOOP
0033 34  END OF MESSAGE?
0034 35  CLHI  BPORT, BUFFND2
0035 36  SEND OUT CHARACTER
0036 37  XDEV, (BPTR)
0037 38  INCREMENT BUFFER POINTER
0038 39  BPORT+1
0039 40  RECEIVE DEVICE ADDRESS
0040 41  XDEV, 0X'16'
0041 42  PASLA SET UP COMMAND
0042 43  RDEV, XADDR
0043 44  ENRCV  DB  X'61'
0044 45  ENABLE INTS, DTR, READ, COMMAND1
0045 46  XDEV, 0X'63'
0046 47  ENABLE INTS, DTR, READ, COMMAND1
0047 48  BPORT
0048 49  BUFFER = 10 CHARACTERS LONG
0049 50  RX
0050 51  END
0051
```
<table>
<thead>
<tr>
<th>Command</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTOP</td>
<td>0000</td>
</tr>
<tr>
<td>ADC</td>
<td>0002</td>
</tr>
<tr>
<td>BPTR</td>
<td>0004</td>
</tr>
<tr>
<td>BUFFER</td>
<td>005FR</td>
</tr>
<tr>
<td>BUFFRD1</td>
<td>0069R</td>
</tr>
<tr>
<td>BUFFRD2</td>
<td>0069R</td>
</tr>
<tr>
<td>CHAR</td>
<td>0005</td>
</tr>
<tr>
<td>ENCRCV</td>
<td>0030R</td>
</tr>
<tr>
<td>EXXNT</td>
<td>005ER</td>
</tr>
<tr>
<td>FLAG</td>
<td>0003</td>
</tr>
<tr>
<td>IMPTOP</td>
<td>006CR</td>
</tr>
<tr>
<td>LADC</td>
<td>0001</td>
</tr>
<tr>
<td>LOOP</td>
<td>003CR</td>
</tr>
<tr>
<td>RADOR</td>
<td>0058R</td>
</tr>
<tr>
<td>RCV</td>
<td>0030R</td>
</tr>
<tr>
<td>RDEV</td>
<td>0001</td>
</tr>
<tr>
<td>SETUP</td>
<td>005CR</td>
</tr>
<tr>
<td>STOP</td>
<td>002CR</td>
</tr>
<tr>
<td>XADOR</td>
<td>005AR</td>
</tr>
<tr>
<td>XDEV</td>
<td>0002</td>
</tr>
<tr>
<td>XDONE</td>
<td>0054R</td>
</tr>
<tr>
<td>XMT</td>
<td>0048R</td>
</tr>
</tbody>
</table>
CHAPTER 3
Asynchronous Control Line Driver -
an Assembler Version

The decision was made to use asynchronous lines to carry control supervisor information between machine message systems within a cluster of the MIMICS network. Therefore, a driver was needed for these lines and it is called the Asynchronous Control Line Driver (ACLDR). The ACLDR acts as an interface between the Asynchronous Communication Processes and the PASLA hardware.

The ACLDR is designed to set up the PASLA and transfer the control information across asynchronous lines in the form of System Control Blocks (SCB). Figure 3.1 shows the format of an SCB that is being transmitted.

<table>
<thead>
<tr>
<th>CHECKSUM</th>
<th>S C B</th>
<th>LENGTH</th>
</tr>
</thead>
</table>

**Flow of Data**

**Figure 3.1**
Format of SCB Being Transmitted
The length, which is transmitted first, is one byte long and is equal to the number of bytes in the SCB excluding the length and checksum bytes. Each byte of the SCB is sent
next followed by a checksum. This checksum is the Modulo 256 sum of the length and all the bytes of the SCB. This chapter describes the ACLDR and how it was implemented on the Interdata 16 bit machines, using Interdata's Common Assembler Language (CAL) [INT02], to transfer SCB's across asynchronous lines.

3.1 Interrupt Structure

Before getting into the details of the ACLDR, one must understand how the Interdata machines handle interrupts. The Interdata has several types of interrupts, such as External, Machine Malfunction, and Fixed Point Fault. These are explained in reference INT02. This section is concerned only with Automatic I/O interrupts which are used for the ACLDR.

When the processor detects an interrupt, it saves the current state of the machine and transfers control to a software routine to handle the interrupt. Automatic I/O interrupts can be enabled or disabled by setting or resetting bits in the Program Status Word. The Program Status Word (PSW) [INT02] is a register in the machine which defines the state of the processor at any given time. If both bits 1 and 4 in the PSW are set then Automatic I/O interrupts are enabled. If bit 1 or both bits 1 and 4 are reset then they are disabled.

When an interrupt occurs and both bits 1 and 4 of the PSW are set, the microcode of the machine takes the interrupting device address, multiplies it by two, and uses
this value as a displacement into the Interrupt Service Table (IST). The IST starts at memory location \texttt{X'D0'} and contains the addresses of the Interrupt Service Routines (ISR) for 255 devices that could be in the system. Referring to Figure 3.2, if device \texttt{X'11'} gave an interrupt the microcode would add \(2 \times X'11'\) to \texttt{X'D0'} to obtain \texttt{X'F2'} which would be the memory location that contains the address of the ISR for device \texttt{X'11'}. When the ISR is entered, the PSW at the time of the interrupt is stored in the first two halfwords of the ISR and the new status is put into the PSW from the third halfword. The new instruction counter in the PSW points to the fourth halfword and execution of the ISR begins there. The Load PSW instruction loads a new state of the processor from a location in memory into the PSW register and when the ISR is to be exited, this instruction is executed with the OLD PSW that was stored in the first two halfwords of the ISR. In doing so the routine that was interrupted regains control at the point where the interrupt occurred.

3.2 Relationship to Asynchronous Processes

As shown in Figure 3.3, the Asynchronous Communication Processes call the ACLDR via one of its three entry points, ACL.SETUP, ACL.RCV, or ACL.XMT. These entry points communicate with the PASLA and the interrupt service routines to set up the PASLA and transfer bytes of data to and from it.
Figure 3.2

Transfer of control caused by an interrupt from device X'11'
Figure 3.3

Relationship of ACLDR to Asynchronous Processes
3.3 Module Layout

In Figure 3.4 the modules of the ACLDR are presented. The ACLDR is reentrant so a block of memory is allocated for all the variables used for each PASLA (Receive/Transmit pair). This block of memory is called the Device Variable Table (DVT). The address of the DVT for a particular device is found in the Device Variable Table Map (DVTMAP). Each entry point in the ACLDR uses the physical line address passed to it as a displacement from the beginning of the DVTMAP to obtain the address of the DVT for that particular device.

When an interrupt occurs on a PASLA, the Device Map (DEVMAP), corresponding to the device which caused that interrupt, is entered. The DEVMAP is similar to Interdata's DCB [IDAT03]. It saves the current state of the machine, sets up the new state for handling the interrupt, and then branches to the Interrupt Service Routine. In the new machine state, interrupts to the processor are disabled and one register contains the address of the DVT. All the variables used by the ISR are obtained by a displacement from this address. The ISR is exited by a branch back to the DEVMAP where the previous state of the machine is restored and the processor resumes execution of the code which was interrupted.
Solid Line — transfer of control
Dotted Line —— pointer to data or addresses

Figure 3.4

Modules of the ACLDR
3.4 ACLDR entry Points

The Asynchronous Communication Processes can call any one of the three entry points in the ACLDR, ACL.SETUP, ACL.RCV, or ACL.XMT. Before doing so, the Asynchronous Process must store the parameters or addresses of the parameters to be passed in an array of contiguous halfwords (two bytes) in memory. The address of this array is stored in General Purpose Register 1 and the return address from the ACLDR is in General Purpose Register 15. Each entry point's functional specifications, the parameters passed to them, and high level algorithms of their code are given below.
ACL_SETUP(line_addr, return_code)

Functional Specifications:

ACL_SETUP is called by the Asynchronous Processes to set up the PASLA corresponding to the line_address passed as a parameter to allow transmission of an SCB.

Parameters:

line_addr - entry parameter - Physical line address of the PASLA being initialized.

return_code - return parameter - The return code to the calling routine. There are two conditions that can be returned:
0 - OK
1 - Instruction timeout

Algorithm:

ACL_SETUP(line_addr, return_code)
DISABLE INTERRUPTS
SET UP THE PASLA
IF "INSTRUCTION TIMEOUT" THEN
    RETURN_CODE = "INSTRUCTION TIMEOUT"
ELSE RETURN_CODE = "OK"
ENABLE INTERRUPTS
RETURN.
ACL_RCV(line_addr, in_buff_addr, length, max_size, return_code)

Functional Specifications:

ACL_RCV is called by the Asynchronous Processes when an SCB is expected to be received. The incoming SCB is placed in the input buffer. If an SCB was not received after a certain length of time, then the return code indicates this condition and control returns to the calling routine.

Parameters:

line_addr - entry parameter - Physical line address of the PASLA used to receive the SCB.

in_buff_addr - entry parameter - The address of the buffer where the SCB to be received will be stored.

length - return parameter - The length of the SCB that was received.

max_size - entry parameter - The maximum size of the input buffer.

return_code - return parameter - The return code to the calling routine. There are five conditions that can be returned:
0 - Reception OK
1 - Loss of Carrier indicating bad line condition
2 - Bad Checksum; error in transmission
3 - Timeout; SCB was not received in allotted amount of time
4 - Length too large; length received was larger than buffer length
Algorithm:

ACL_RCV(line_addr, in_buff_addr, length, maxsize, return_code)
SET RECEIVE_MODE TO ALLOW RECEIVING OF CHARACTERS
DO WHILE TIMEOUT HAS NOT OCCURRED
IF CARRIER WAS LOST THEN
    SET RETURN_CODE = "LOSS OF CARRIER"
    CLEAR RECEIVE_MODE
    RETURN
IF RECEPTION OF SCB IS DONE THEN
    IF CHECKSUM IS BAD THEN
        SET RETURN_CODE = "BAD CHECKSUM"
    ELSE SET RETURN_CODE = "OK"
    CLEAR RECEIVE_MODE
    RETURN
END WHILE
CLEAR RECEIVE_MODE
SET RETURN_CODE = "TIMEOUT"
RETURN.
ACL_XMT(line_addr, out_buff_addr, length, return_code)

Functional Specifications:

ACL_XMT is called by the Asynchronous Processes for transmission of an SCB from the output buffer. The length of this SCB is transmitted just before, and a checksum right after, the SCB itself. If the loss of carrier is detected during transmission then the return code indicates this fact, otherwise OK is returned.

Parameters:

- **line_addr** - entry parameter - The physical line address of the PASLA used to transmit the SCB
- **out_buff_addr** - entry parameter - The address of the SCB to be transmitted
- **length** - entry parameter - The length of the SCB being transmitted. This length is attached to the beginning of the SCB
- **return_code** - return parameter - The return code to the calling routine. There are two conditions that can be returned:
  0 - OK
  1 - Loss of carrier

Algorithm:

ACL_XMT(line_addr, out_buff_addr, length, return_code)
SIMULATE AN INTERRUPT TO START TRANSMISSION
DO WHILE TRANSMISSION TAKING PLACE
IF LOSS OF CARRIER THEN
   SET RETURN_CODE = "LOSS OF CARRIER"
   RETURN
END WHILE
SET RETURN_CODE = "OK"
RETURN.
3.5 Interrupt Service Routine:

RCV

Functional Specifications:

RCV is a reentrant procedure used to service interrupts coming from the receive side of a PASLA. The RCV routine ignores all interrupts until ACL_RCV sets the Receive Mode flag on. It then receives the length of the incoming SCB, the SCB itself, and a checksum. If the length is longer than the buffer size, then only enough characters to fill up the buffer will be read in. The excess will be ignored, resulting in a checksum error. If the length is correct but the checksum is wrong then the return code from ACL_RCV will indicate so.

Algorithm:

IF LOSS OF CARRIER IS DETECTED THEN
    SET CARRIER FLAG
    RETURN

IF THE RECEIVE MODE IS NOT ACTIVE THEN RETURN

READ THE CHARACTER

IF IT IS THE FIRST CHARACTER THEN
    SET THE BUFFER POINTER
    SET LENGTH = VALUE OF THE CHARACTER
    RETURN

IF THE BUFFER IS FULL THEN
    IF THE CHECKSUM = THE CHARACTER
        THEN CHECKSUMFLAG = "OK"
        ELSE CHECKSUMFLAG = "BAD CHECKSUM"
    SET THE DONE FLAG
    DISABLE RECEIVE MODE
    RETURN

ELSE
    STORE THE CHARACTER IN THE BUFFER
    ADD IT TO THE CHECKSUM
    INCREMENT THE BUFFER POINTER
    RETURN

-39-
XMT

Functional Specifications

XMT is a reentrant procedure used to service interrupts coming from the transmit side of a PASLA. When a process wishes to send a message across an asynchronous line it calls ACL_XMT. ACL_XMT sets up the necessary variables to transmit a buffer and then simulates an interrupt to send control to XMT. XMT then transmits a character from the buffer. Each time a character leaves the PASLA another interrupt is generated which informs XMT that it may send another character. XMT first transmits the length of the buffer, then each character in the buffer, and then a checksum.

Algorithm:

IF DATA TERMINAL READY INTERRUPT THEN RETURN
IF LOSS OF CARRIER DETECTED THEN
  SET CARRIER FLAG
  RETURN
IF BEGINNING OF BUFFER THEN
  TRANSMIT LENGTH
  RETURN
IF BUFFER EMPTY THEN
  IF CHECKSUM HAS BEEN SENT THEN
    RETURN
  ELSE TRANSMIT CHECKSUM
  SET THE DONE FLAG
  RETURN
ELSE
  ADD CHARACTER TO CHECKSUM
  TRANSMIT THE CHARACTER
  INCREMENT THE BUFFER POINTER
  RETURN.
3.6 Notes on ACLDR Operation

The user of the ACLDR must know about certain assertions and conditions necessary for its correct operation. The ACLDR assumes that any parameter that is sent to it is a legitimate variable, i.e. the line_addresses are actually in the system, the buffer_addresses do not exceed the top of memory, etc. Also, when loss of carrier occurs because of a line break, hardware malfunction, etc., this condition is hidden from the Asynchronous Processes until the ACLDR is called to transmit or receive an SCB.
CHAPTER 4

Asynchronous Control Line Driver -

a Concurrent PASCAL Version

Because of the difficulty in programming and debugging in assembler language, the ACLDR was also written in Concurrent PASCAL. This high level language was designed and implemented by Per Brinch Hansen [BRH75a] on a PDP 11/45. It has also been "ported" to an Interdata 8/32 by KSU personnel [MAR76]. The NAVY has developed a KERNEL [COZ76] which runs on Interdata 16-bit machines and Concurrent PASCAL programs compiled on the 8/32 run under this KERNEL. The remainder of this chapter first describes the input/output machine of the NAVY's KERNEL and then the implementation of the ACLDR in Concurrent PASCAL.

It is assumed that the reader is familiar with Concurrent PASCAL and the compilation of programs under the SOLO operating system [BRH76][MAR76]. Since the 16-bit Interdata machines' amount of memory is too small to run a PASCAL compiler, the KERNEL and any Concurrent PASCAL programs running under that KERNEL must be compiled on the 8/32 using the SOLO system and transferred via mag tape, disk, etc. to the 16-bit machines.

4.1 IO_MACHINE

The KERNEL which runs on the 16-bit machines contains an entry point called the IO_MACHINE [COZ76]. This entry
point was designed to allow the user to control the operation of a device, independent of a particular machine or device. This eliminates the need for the user to handle the interrupts at the hardware level and also hides from him the actual method of transferring data to and from the device. This however, does not relinquish his control of the device. The IO_MACHINE instructions handle the usual I/O functions of issuing commands, sensing status, and reading or writing of data for each device. Although a lot of the work is done by the IO_MACHINE, the user must still understand the functional operation of the device in order to know which output commands control it and to understand the status that is returned. A device, in this context, is the actual hardware interface which handles the transfer of data to and from a terminal, disk, etc. or in the case of this report, to and from another hardware interface, i.e. a PASLA.

There are 21 instructions available for the user of the IO_MACHINE. These instructions can be divided into three categories. First, the device control instructions consist of CHANNEL, COMMAND, SENSE, WAIT_INTERRUPT, PREEMPT, and FREE. Second, the flow of execution is controlled by the COMPARE, JUMP, and RETURN instructions. The final category, data transfer instructions, are DATA, START_INPUT, and START_OUTPUT. The explanation of each individual instruction can be found in Appendix A. This explanation is contained in reference CO276; it was included because it was not a permanent record at KSU as of April 1977, and was
found to be concise. A description of the instructions by this author would be redundant so the remainder of this section is dedicated to an explanation of how the IO_MACHINE is used followed by a simple example.

When the IO_MACHINE is called, the parameter passed to it is a record in the form shown below:

\[
\text{io_record} = \text{RECORD}
\begin{align*}
\text{instruction_counter: integer;} \\
\text{status: integer;} \\
\text{bytes_transferred: integer;} \\
\text{internal_use: ARRAY [0..12] OF integer;} \\
\text{inst: ARRAY [0..max_instruction] OF integer}
\end{align*}
\text{END "record";}
\]

The instruction_counter is the index of which instruction will be executed next in the instruction array. The status contains the device status in the lower byte of the integer and IO_MACHINE status in the upper byte upon exit from the IO_MACHINE. The number of bytes transferred is given next. The internal_use array is for the IO_MACHINE to make the code reentrant and is of no use to the user. The "inst" variable is an array of integers specifying the IO_MACHINE instructions and must have the starting index at 0. These integers are the instructions interpreted by the IO_MACHINE. Each instruction will be referenced in this report by name (e.g. IO_COMMAND or IO_RETURN) and not by its integer value.

The instruction array can be set up at either initialization time or at run time. Particular instructions can also be added or changed at run time as long as the instruction array index does not exceed the maximum number
of instructions. If any instruction has an argument, this argument will be in the next successive element or elements of the instruction array. In the case of an immediate instruction, the data is the argument itself; otherwise the argument is the address of the data.

The instruction_counter in the io_record points to the next instruction to be executed. When the IO_MACHINE is entered, the first instruction must be an IO_CHANNEL or an IO_CHANNEL_IMMEDIATE instruction. These instructions indicate to the IO_MACHINE which device to use. Each instruction is executed in order unless a jump is encountered. A normal exit from the IO_MACHINE occurs when an IO_RETURN is executed. Abnormal exits are caused by errors which occur during the execution of the IO_MACHINE instructions. These errors are also explained in Appendix A. The error conditions are returned in the STATUS word of the io_record, as shown in Figure 4.1, with the value of the device status in the lower byte and the value of the IO_MACHINE status in the upper byte. This feature gives the user the ability to look at the condition of the device and take appropriate action to control the device in the manner which best fits his needs.
4.2 Example of a CRT Driver Using a PASLA

The use of the IO_MACHINE is best described by a short example. Section 2.4 gave the assembler version for a CRT driver which reads in 10 characters and then echoes them back to the CRT. The following example does the same function, but it is written in Concurrent PASCAL and uses the IO_MACHINE. The Command words for the Output Commands are the same in both examples except that the assembler version uses a hexadecimal value and the Concurrent PASCAL version uses decimal integers. In other words, the COMMAND 2 byte for the example in Section 2.4 was hexadecimal '6C' and in this it is decimal '108'. The Concurrent PASCAL code for this example follows:
CONST
CRT = 91 "CRT - PASLA X*12"

TYPE EXAMPLE_CLASS = CLASS

CONST
MAX_INSTRUCTION = 29
BUSY_STAT = 81
"COMMAND 1 BITS"
TEABLE = 44 "INTERRUPTS ON"
TDR = 321 "DATA TERMINAL READY"
TMD = 81 "WHITE MODE"
TRD = 41 "READ MODE"
TCHND1 = 11 "COMMAND 1"
"COMMAND 2 BITS"
TIGH = 41 "HIGH BAUD RATE"
T7BITS = 81 "7 BITS OF DATA"
T2STOPs = 81 "2 STOP BITS"
T00D = 41 "ODD PARITY"

TYPE
IO_TYPE = RECORD
   IC, "IO INSTRUCTION COUNTER"
   STATUS, "IO MACHINE STATUS ON RETURN"
   NCXFER: INTEGER
   INTERNAL: ARRAY [0..12] OF INTEGER
   INST: ARRAY [0..MAX_INSTRUCTION] OF INTEGER
END "IO_TYPE"

TYPE LIN12 = ARRAY [1..12] OF CHAR;
VAR
IO_RECORD: IO_TYPE;
BUFFER: LIN12;

PROCEDURE ENTRY EXAMPLE;
BEGIN
WITH IO_RECORD DO
BEGIN
   IC := 01; STATUS := 01;
   IO(IO_RECORD): "READ IN 10 CHARACTERS"
   BUFFER[12] := 111111111; "LINE FEED"
   IC := 13; STATUS := 01;
   IO(IO_RECORD): "WRITE OUT THE BUFFER"
END;
END;

BEGIN "INIT"
WITH IO_RECORD DO
BEGIN
   INST[0] := IO_CHANNEL_IMMEDIATE; INST[1] := CRT
   INST[9] := IO_WAIT_INTERRUPT
   INST[12] := IO_RETURN;
INST[15] := IO_COMMAND_IMMEDIATE;
INST[16] := ENABLE+TDR+TMR+TCMD1;
INST[20] := IO_SENSE;
INST[25] := IO_WAIT_INTERRUPT;
INST[26] := IO_START_OUTPUT; INST[27] := NO_COMMAND;
INST[28] := IO_WAIT_INTERRUPT;
INST[29] := IO_RETURN;
END;
END "EXAMPLE CLASS";
END.
4.3 Relationship to Asynchronous Processes

The Asynchronous Control Line Driver is functionally the same for both the assembler version and the Concurrent PASCAL version. The Asynchronous Communication Processes call the three entry points of the ACLDR to set up the PASLA (CALL ACL_SETUP), receive an SCB (CALL ACL_RCV), or transmit an SCB (CALL ACL_XMT). Each entry point then communicates with the PASLA to perform the desired operations. In the assembler version, it is the job of the programmer to write the code which communicates with the PASLA and handles the interrupts coming from that PASLA. The PASCAL process (or a class being executed by a process) needs only to call the IO_MACHINE in the KERNEL, with a set of instructions as parameters, and the IO_MACHINE does the rest.

In Figure 3.3 we show several modules needed for the implementation of the ACLDR in assembler language. The use of Concurrent PASCAL and the IO_MACHINE eliminates the need for the user to program these modules. Concurrent PASCAL allows the entry points, defined to be classes, to be reentrant so the user does not have to be concerned with special variable tables for each device, nor does he need the mapping of addresses to gain access to these tables. Because of the features of the IO_MACHINE, Figure 3.3 for the assembler version has been reduced to Figure 4.2 for the Concurrent PASCAL version. The ACLDR now contains only three classes, each being an entry point called by the Asynchronous Processes.
Figure 4.2 Relationship of ACLDR to Asynchronous Processes
4.4 Entry Points

The purpose of each entry point call and the parameters passed with the call are identical for both the assembler version and the Concurrent PASCAL version except for one slight change. Section 3.4 presented the functional specifications, the parameters passed, and high level algorithms for each entry point. To review, the calls and the parameters passed were as follows:

ACL_SETUP(line_addr, return_code)
ACL_RCV(line_addr, in_buff_addr, length, max_size, return_code)
ACL_XMT(line_addr, out_buff_addr, length, return_code)

The only difference between the two versions is that the line_addr parameter is the actual hardware device address in the assembler version where as it is a device index in the Concurrent PASCAL version. The KERNEL uses this index as a displacement into a table of the devices in the system.
CHAPTER 5

Summary

In this paper we presented the design and implementation of an Asynchronous Control Line Driver for the MIMICS network. We have shown how the ACLDR is used in the network and why asynchronous lines were chosen to carry control information in MIMICS. Implementation was accomplished on an Interdata 7/16 and an Interdata 85 using PASLA's to connect the two machines. Two different languages were used for this implementation. One version of the ACLDR is written in Assembler language and the other in Concurrent PASCAL. This summary discusses the advantages and disadvantages of using assembler language versus Concurrent PASCAL in the implementation of the ACLDR, the work completed, and a few possible extensions of the ACLDR to other machines.

5.1 Comparison of Assembler and PASCAL versions

The ACLDR was first written in assembler language on the Interdata 85. Since the operation of the PASLA for this project is somewhat different than most Input/Output devices, assembler language provided a low enough level of programming to control the hardware in the manner needed. It also allowed the programmer to minimize the amount of memory used by the ACLDR which is necessary because the ACLDR will eventually be used in micro-computers where memory use is critical.

-52-
Programming in assembler language has its advantages but it is very time consuming and often difficult to debug. After the assembler version of the ACLDR was written, Concurrent PASCAL became available for the Interdata 16-bit machines. This allowed the programming of the driver with a high level language. Concurrent PASCAL eliminates the need for the programmer to manipulate addresses or registers or to make the code reentrant. Along with these, the major advantage of using Concurrent PASCAL and the IO_MACHINE is that interrupts are handled by the IO_MACHINE. The time involved in writing and debugging the ACLDR in assembler language was approximately three times as long as for the Concurrent PASCAL version.

The only disadvantage in using Concurrent PASCAL is the amount of memory used. The assembler version of the ACLDR used approximately 1K bytes of memory with one asynchronous line. Each additional asynchronous line adds 134 bytes for the DEVMAP and the DVT of that line. The IO_MACHINE, on the other hand, eliminates the need for the Interrupt Service Routines but it also uses approximately 6K bytes of memory. This code however, is utilized by all peripheral devices. The three classes that make up the ACLDR take nearly 2K bytes of memory. On this basis, if the amount of memory used by the ACLDR is not as critical as the time and effort of implementation, then development using Concurrent PASCAL is worth the use of more memory.
5.2 Worked completed

Both the Assembler version and the Concurrent PASCAL version of the ACLDR run on either the Interdata 85 or the 7/16. The Concurrent PASCAL KERNEL is different for the two machines but this has no effect on the ACLDR running under the KERNEL.

5.3 Extensions

One of the major reasons for the use of Asynchronous lines to carry control information between computers in MIMICS is that they are fairly standard throughout the computer industry. This allows the connection of any two machines that support asynchronous lines. This project involved linking two Interdata 16-bit machines together. Some other possible connections would be to an Interdata 8/32, an ALTAIR 8800, or a Data General NOVA. The 8/32 would use the PASLA just like the Interdata 16-bit machines. The ALTAIR is a microcomputer which uses a Motorola Asynchronous Communication Interface Adapter (ACIA). The ACLDR would then have to be written in Intel 8080 code which is the assembler code for the ALTAIR 8800. If the NOVA were to be used the ACLDR would have to be converted to NOVA code and hardware interface would be NOVA's 4029 modem interface. The control signals used for the link between these other machines would be the same for the links to the two Interdata machines.
BIBLIOGRAPHY


Channel _Immediate (device index)

On every entry to the IO_MACHINE, the device index must be registered before any other instructions are executed. This allows the IO_MACHINE to make sure that the device is known to the system, that there are no errors associated with the device, and that the device has not already been preempted by another process. The device index is an integer established to reference the peripheral without needing to know its hardware address.

Channel (address of device index)

This is the same as 'Channel _Immediate' except for the argument. For this instruction, the next location in the instruction array contains the address where the device.index may be found.

Command _Immediate (command data)

This instruction sends 'command data' to the currently addressed device as a command word. Since the Interdata commands are bytes, only the eight least significant bits of the argument are interpreted as a command. The rest of the integer is ignored. The status of the device is checked here. If the device does not respond (timeout status), then execution in the IO_MACHINE is terminated.

Command (address of command data)

This is similar to 'Command _Immediate' above.

Sense

This instruction simply senses the status of the presently addressed device.

Sense _External (address of status location)

The device status is read and stored into the location provided by the argument. The status word in the io record is also updated.
Compare_IMMEDIATE (mask)

The contents of the status word in the io record is logically AND'ed with 'mask'. If the result is zero, an internal flag is set to FALSE, otherwise it is set to TRUE. This flag is used by the conditional jump and return instructions.

Compare (address of mask)

This is similar to 'Compare_IMMEDIATE', of course.

Jump (instruction index)

Set the instruction counter to the value of the argument. This instruction causes a jump to continue execution at another part of the instruction array.

Jump_TRUE (instruction index)

If the result of the last comparison instruction was TRUE, then set the instruction index to the argument value. Otherwise, execution continues with the following instruction.

Jump_FALSE (instruction index)

If the result of the last comparison was FALSE, then jump to the instruction specified by the argument to continue execution. Otherwise, execution continues at the following instruction.

Return

Execution of instructions in the IO_MACHINE is terminated, and control of the program returns to the calling process. The instruction counter is left positioned at the next instruction so that another call to the IO_MACHINE with this io record would resume execution of the io instructions following the 'Return'.

Return_TRUE

If the result of the last comparison was TRUE, then control returns to the calling process as in the
'Return' instruction. Otherwise, execution continues with the following instruction.

**Return_False**

If the result of the last comparison was FALSE, then the IO_MACHINE returns to the calling process. Otherwise, execution continues with the following instruction.

**Data_Immediate** (n, byte 1, byte 2, ..., byte n)

The purpose of this instruction is to establish a data buffer for a subsequent transfer to the addressed device. In the case of the 'Immediate' version, the data buffer is actually in the instruction array following the 'Data_Immediate' instruction. The first argument gives the number of bytes to be transferred; the remaining arguments are the data. The form of data transfer depends upon the device - see the description of the 'Start' instruction for more details. Note that no direct buffer linking is possible with the IO_MACHINE. This is a very insecure method of handling the devices and was avoided. If a 'Data' instruction is executed for some device before the previous buffer was used, the description of the previous buffer will be lost.

**Data** (n, address of buffer)

This is similar to 'Data_Immediate' above, but the address of the data buffer is the second argument.

**Wait_Interrupt**

On executing this instruction, the calling process is suspended from execution on an interrupt queue associated with the currently addressed device. When an interrupt from that device is detected, execution of the process resumes immediately inside the IO_MACHINE. Remember that the interrupts are always off while the executing IO_MACHINE instructions. The user must insure that only one process is allowed to wait for an interrupt for any given device at a time. Clearly, this is an opportunity for messing up the io, since a process could wait for an interrupt without having previously caused the device to generate one. It could be a long wait!
Start_Input (command)

Data is transferred from the currently addressed device to the data buffer which was established earlier with a 'Data' or 'Data_Immediate' instruction. When execution of io instructions resumes, the entire data buffer has been transferred. The status word in the io record is set to indicate the final status of the device. The number of bytes actually transferred is returned in the io record, so in case of an incomplete operation, the calling process can determine the state of things.

The argument (command) is an output command that is sent immediately to the device and is included mainly for disk operations.

In the IO_MACHINE, devices are configured to transfer data in one of four distinct ways. The configuration of the current devices is given in the device table below. For general reference, the four types of transfer are:

Direct IO: The first and slowest of these methods uses direct io instructions to transfer data to or from the device. This may be by single byte or by halfword, and may be done with or without interrupts. If interrupts are not used, then the device must send or accept data as fast as the IO_MACHINE can transfer it. If the device transfer is to use interrupts, then if more than one io instruction must be executed, the IO_MACHINE waits for an interrupt from the device and checks that the status is zero before proceeding. If interrupts are being used, the user will probably want to use an explicit 'Wait_Interrupt' instruction before starting a read and after finishing a write.

Autochannel: Interdata provides an 'automatic' method for sending one byte per interrupt called the Automatic IO Channel. In this scheme, the entire block is transferred using interrupts without the user having to handle them explicitly with IO_MACHINE instructions.

Selector Channel: This is essentially a single channel DMA in series with the device. The Selector Channel hardware handles the interrupts and transfers the data to memory without disturbing the CPU.

Direct Memory Access: Our DMA will transfer data
directly to or from the device and memory. There are eight channels available.

Start_Output (command)

This is similar to 'Start_Input'. For some devices, the handling of interrupts is different for output than for input so the user must be careful.

Preempt

If the calling process wants to be sure that no other device will interfere with its device operation, it may execute the 'Preempt' instruction. This sets the state of the currently addressed device so that no other process may address the device with the 'Channel' instructions until the preempting process decides to release it. If a process is already waiting for an interrupt, it is returned to execution with the status indicating that it was preempted. Note that the preempting process cannot preempt an already preempted device, because the initial 'Channel' instruction would have failed. It is also important for the user to be sure that preempting a device doesn't leave it in some sort of state where errors could be produced.

Free

When a process which has preempted a device is through with it, this instruction is executed so that other processes may have the use of the device. A preempted device stays preempted through multiple calls to the IO_MACHINE - only execution of the 'free' will release it.
IO_MACHINE ERRORS

When the IO_MACHINE detects an error condition, it sets the status word in the io record to indicate the problem. In the In the Interdata, the status of a device is represented by one byte. The IO_MACHINE returns device status in the least significant byte, internal status in the next byte. Since any internal error causes the IO_MACHINE to terminate execution of the instruction, take care to leave the device in a proper state. In particular, an error may cause a process to be removed from the IO_MACHINE before a preempted device can be freed. The various error states are mutually exclusive and are represented by different integers. The values of the IO_MACHINE errors and their explanation follow. Some have been described earlier in the discussion of the IO_MACHINE instructions.

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Instruction Counter Range - The instruction counter has been incremented past the end of the instruction array.</td>
</tr>
<tr>
<td>2</td>
<td>Device Index Range - The device index as given by the argument to the 'channel' instruction is out of range.</td>
</tr>
<tr>
<td>3</td>
<td>Device Interrupt - An interrupt occurred for this device while there was no process waiting to handle it. This status may be set upon execution of the 'channel' instruction.</td>
</tr>
<tr>
<td>4</td>
<td>Instruction Range - The integer indicated by the instruction counter is not a valid IO_MACHINE instruction.</td>
</tr>
<tr>
<td>5</td>
<td>Command Timeout - Sending the command to the addressed device gave a timeout status. That status will be in the lower byte of the io record status word.</td>
</tr>
<tr>
<td>6</td>
<td>Channel Timeout - Block transfer attempted on a channel which gives a timeout status.</td>
</tr>
<tr>
<td>7</td>
<td>Autochannel Timeout - A device using the autochannel returned a timeout status.</td>
</tr>
<tr>
<td>8</td>
<td>Selch Timeout - A selector channel returned a timeout status.</td>
</tr>
<tr>
<td>9</td>
<td>DMA Timeout - A DMA channel returned a timeout status.</td>
</tr>
<tr>
<td>10</td>
<td>Preempted - The device referenced by a 'channel'</td>
</tr>
</tbody>
</table>
instruction is already preempted, or else this process was preempted from a device interrupt queue.

11 Device Setup - The first instruction is not either a 'channel' or 'channel immediate' instruction.
APPENDIX B

ACLDR Assembler Code

The following listing is the Assembler version code for the ACLDR.
ASYNCHRONOUS CONTROL LINE DRIVER

THE ASYNCHRONOUS CONTROL LINE DRIVER CONSISTS OF
THREE ENTRY POINTS: ACL, SET, ACL, RCV, ACL, XMT.
PLUS TWO INTERRUPT SERVICE ROUTINES: RCV AND XMT.
EACH ENTRY POINT CAN BE CALLED BY THE ACLDR
TO SET UP A PASLA AND TRANSMIT AND RECEIVE
CONTROL MESSAGES THROUGH THAT PASLA

REGISTER EQUATES

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>R0 EQU 0</td>
</tr>
<tr>
<td>0001</td>
<td>R1 EQU 1</td>
</tr>
<tr>
<td>0002</td>
<td>R2 EQU 2</td>
</tr>
<tr>
<td>0003</td>
<td>R3 EQU 3</td>
</tr>
<tr>
<td>0004</td>
<td>R4 EQU 4</td>
</tr>
<tr>
<td>0005</td>
<td>R5 EQU 5</td>
</tr>
<tr>
<td>0006</td>
<td>R6 EQU 6</td>
</tr>
<tr>
<td>0007</td>
<td>R7 EQU 7</td>
</tr>
<tr>
<td>0008</td>
<td>R8 EQU 8</td>
</tr>
<tr>
<td>0009</td>
<td>R9 EQU 9</td>
</tr>
<tr>
<td>000A</td>
<td>R10 EQU 10</td>
</tr>
<tr>
<td>000B</td>
<td>R11 EQU 11</td>
</tr>
<tr>
<td>000C</td>
<td>R12 EQU 12</td>
</tr>
<tr>
<td>000D</td>
<td>R13 EQU 13</td>
</tr>
<tr>
<td>000E</td>
<td>R14 EQU 14</td>
</tr>
<tr>
<td>000F</td>
<td>R15 EQU 15</td>
</tr>
</tbody>
</table>
ASYNCHRONOUS CONTROL LINE DRIVER

DEVICE VARIABLE TABLE DISPLACEMENTS

42 *
43 *
44 *
45 *
46 *
47 *
48 *
49 *
50 *

0000    RCVADR EQU 0
0002    XMTADR EQU 2
0004    INPTR EQU 4
0006    INSCB EQU 6
0008    ISCHND EQU 8
000A    OUTPTR EQU 10
000C    OUTSCB EQU 12
000E    OUTCHND EQU 14
0010    MAXSIZE EQU 16
0012    LENGTH EQU 18
0014    MLNFLG EQU 20
0016    CNFIFLG EQU 22
0018    CNFIFLG EQU 24
001A    CKSFLG EQU 26
001C    DONEFLG EQU 28
001E    INITFLG EQU 30
0020    MODE EQU 32
0022    CHECKSUM EQU 34
0024    PASLA EQU 36
0026    CASRCV EQU 37
002A    ENKFT EQU 38
002C    REGSAVE EQU 40

42 * 43 * 44 * 45 * 46 * 47 * 48 * 49 * 50 *

PASLA ADDRESSES

0014  XADDR EQU X'14'
0015  XADDR EQU X'15'
ASYMMRONOUS CONTROL LINE DRIVER

DEVICE MAP FOR RECEIVE SIDE OF PASLA

0000H 0000 94  DEVMAP14 DC 000X'3000'
0002H 0000
0004H 3000
0006H 0000 003CR
0008H 0100 001AR
96  STM 0, IORSAVE  SAVE REGISTERS
97  LM 13, DBV14  REG 13 = ADDR OF SERVICE ROUTINE
98  *  REG 14 = DEVICE VARIABLE TABLE
99  *  AND REG 15 = RETURN ADDRESS
000AH 01FD
0019H 0100 003CR
100  BALR 15, 13  GO TO ISR
101  LM 0, IORSAVE  RESTORE REGISTERS
102  LPSW DEVMAP14  RETURN
103  DBV14 DC Z(RCV)
104  DC Z(GVT14)
105  DC 0
106  *
107  *
108  *
109  *
110  *
111  *
112  *
113  *
114  *
115  DEVMAP15 DC 000X'3000'

DEVICE MAP FOR TRANSMIT SIDE OF PASLA

0010H 0000
0020H 0000
0022H 3000
0024H 0000 003CR
0028H 0100 003AR
116  STM 0, IORSAVE  SAVE REGISTERS
117  LM 13, DBV15  REG 13 = ADDR OF SERVICE ROUTINE
118  *  REG 14 = DEVICE VARIABLE TABLE
119  *  AND REG 15 = RETURN ADDRESS
002AH 01FD
0029H 0100 003CR
120  BALR 15, 13  GO TO ISR
121  LM 0, IORSAVE  RESTORE Registers
122  LPSW DEVMAP15  RETURN
123  DBV15 DC Z(XMT)
124  DC Z(DVT14)
125  DC 0
126  *
127  *
128  *
129  IORSAVE DSH 16  SAVE AREA
### Asynchronous Control Line Driver

**Device Variable Table for PASLA X'14'**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>005CR</td>
<td>0014</td>
<td>DC</td>
</tr>
<tr>
<td>005ER</td>
<td>0015</td>
<td>DC</td>
</tr>
<tr>
<td>006OR</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>006DR</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>006HR</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>006AR</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>006CR</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>007OR</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>007ER</td>
<td>0000</td>
<td>DC 0</td>
</tr>
<tr>
<td>008OR</td>
<td>76</td>
<td>DB X'76'</td>
</tr>
<tr>
<td>0081R</td>
<td>61</td>
<td>DB X'61'</td>
</tr>
<tr>
<td>0082R</td>
<td>63</td>
<td>DB X'63'</td>
</tr>
<tr>
<td>0083R</td>
<td>00</td>
<td>DB 0</td>
</tr>
<tr>
<td>0084R</td>
<td>00</td>
<td>DB 0</td>
</tr>
<tr>
<td>0085R</td>
<td>00</td>
<td>DB 0</td>
</tr>
</tbody>
</table>

**Register Description**

- **RCVADDR**: Receive Address
- **XMTADDR**: Transmit Address
- **INPTR**: Input Pointer
- **INSCI**: Input Count
- **ISCBD**: Input Count Buffer
- **OUTPR**: Output Pointer
- **OUTSCB**: Output Count Buffer
- **MAXSIZE**: MAX Size
- **MLNFLG**: MLN Flag
- **CARRFLG**: Carry Flag
- **LENGFLG**: Length Flag
- **CKSMFLG**: Checksum Flag
- **DONEFLG**: Done Flag
- **INITFLG**: Initialize Flag
- **MODE**: Mode
- **CHECKSUM**: Checksum
- **PASLA SET UP**: PASLA Setup
- **ENRCV**: Enable Receiver
- **ENMAT**: Enable Matrix
- **REGSAVE**: Register Save
<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>177</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>ACL_SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>181</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>182</td>
<td>ACL_SET IS USED TO SET UP A PASLA AND ENABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>183</td>
<td>INTERRUPTS FOR THE RECEIVE AND TRANSMIT SIDES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>184</td>
<td>OF THE PASLA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>186</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>189</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>LH R14,0(R1)</td>
<td>GET LINE NUMBER</td>
<td></td>
</tr>
<tr>
<td>191</td>
<td>LHR R13,R14</td>
<td>SAVE RECEIVE LINE NUMBER</td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>LHR R12,R13</td>
<td>SAVE TRANSMIT LINE NUMBER</td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>AIS R12,1</td>
<td>XMT LINE NUMBER</td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>NHI R14,X'FFFE'</td>
<td>SUBTRACT X'10' AND MAKE IT EVEN</td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>LHI R11,UVMAP</td>
<td>GET DEVICE VARIABLE MAP</td>
<td></td>
</tr>
<tr>
<td>196</td>
<td>AHR R14,R11</td>
<td>ADD DISPLACEMENT</td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>LH R14,0(R14)</td>
<td>GET ADDR OF DEVICE VARIABLE TABLE</td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>STM 0,REGSAVE(R14)</td>
<td>SAVE REGISTERS</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>LPSW DISABLE</td>
<td>DISABLE INTERRUPTS</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>DC R15,PASLA(R14)</td>
<td>TURN PASLA ON</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>BFC 4,13</td>
<td>NO BRANCH IF INST TIME-OUT</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>LIS R10,1</td>
<td>SET RCODE</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>TH R10,2(R1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>LPSW SPSW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>I3 DC R13,CHRCV(R14)</td>
<td>ENABLE RECEIVE INTERRUPTS</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>UC R12,ENXMT(R14)</td>
<td>ENABLE TRANSMIT INTERRUPTS</td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>XHR R10,R10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>STH R10,INITFLG(R14)</td>
<td>CLEAR INITIALIZATION FLAG</td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>RDA R13,9</td>
<td>INSURE THAT BUSY IS SET</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>LPSW ENABLE</td>
<td>ENABLE INTERRUPTS</td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>I4 STH R10,2(R1)</td>
<td>SET RCODE</td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>LIS R6,1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>STH R6,CARFLG(R14)</td>
<td>SET CARRIER FLAG</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>SRTN LA 0,REGSAVE(R14)</td>
<td>RESTORE REGISTERS</td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>B 0(R10)</td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>217</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>DISABLE DC X'3000'+Z(I2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>ENABLE DC X'7000'+Z(I4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>SPSW DC X'7000'+Z(SRTN)</td>
<td>RETURN</td>
<td></td>
</tr>
</tbody>
</table>
ASYNCHRONOUS CONTROL LINE DRIVER

ACL.RCV

ACL.RCV IS USED TO SET THE ACLDR IN THE RECEIVE MODE AND WAIT FOR A CONTROL MESSAGE TO BE RECEIVED OR A TIMEOUT TO OCCUR.

222
223
224
225
226
227
228
229
230
231
232
233
234

235 ACL.RCV EQU *
236 LH R14,0(R1)
237 LH R12,K14
238 LH R12,K14
239 LH R13,DVTHAP
240 LH R14,K13
241 LHI R14,0(R14)
242 STM 0,R14A(R14)
243 LH R0,2(R1)
244 LH R0,1(R14)
245 LH R0,6(R1)
246 LH R0,MAYSIZE(R14)
247 LH R0,1
248 LH R0,MODE(R14)
249 XHR R0,R0
250 STM R0,LENFLG(R14)
251 STM R0,LENFLG(R14)
252 STM R0,LENGHT(R14)
253 LH R12,RCVADDR(R14)
254 STM R0,7(R1)
255 STM R0,7(R1)
256 STM 7(R1)
257 AR0 015AR EQU *
258 AR0 015AR EQU *
259 AR0 015AR EQU *
260 AR0 015AR EQU *
261 AR0 015AR EQU *
262 AR0 015AR EQU *
263 AR0 015AR EQU *
264 AR0 015AR EQU *
265 AR0 015AR EQU *
266 AR0 015AR EQU *
267 AR0 015AR EQU *
268 AR0 015AR EQU *
269 AR0 015AR EQU *
270 AR0 015AR EQU *
271 AR0 015AR EQU *
272 AR0 015AR EQU *
273 AR0 015AR EQU *
274 AR0 015AR EQU *
275 AR0 015AR EQU *
ASYMMONOUS CONTROL LINE DRIVER

EQU *
GET MAX LENGTH FLAG
LH R8,MLENFLG(R14)
LENGTH TO LARGE?
CLHI R6,1
BNE ARb
BRANCH IF NOT

DELAY LOOP
LIS R6,1
FOR 5 SECOND
AD
TIMEOUT
SIS R6,1
DECIMENT TIME COUNTER
DIP AR6
NO BRANCH IF TIMEOUT

RCODE = TIMEOUT

PUT RCODE IN PARM LIST

CLEAR RECEIVE MODE

RESTORE REGISTERS

RETURN
ACLXMT is used to start the transmission of a control message and see to its completion.

```
O16AR 48E1 0000
016AR 069E
016CR 49E0 FFEE
0160R C060 00A0
0164A 0AEC
016EA 48EE 0000
016AR D0A8 002A
016CR 0700
016R 400E 001C
016R 400E 001A
016R 400E 0001
0160R 480E 0001
0160R 40AE 000C
0160R 0000
016AR 40AE 0010
016AR E20C 0000
016AR 400E 0016
016AR 48A0 0001
016A 40AE 0016
016R C500 0000
022R 4330 0234R
022R 46AE 001C
022AR 05A0 0001
022AR 4250 0212R
022R 07AA
022R 494A 0006
022AR D0A0 022A
023F 430F 0000
```

ACLXMT EQU *

LH R14,0(R1)
GET LINE NUMBER

LHR R14,R14
SAVE LINE NUMBER

NHX R14,FFEE
SUBTRACT X*10' AND MAKE IT EVEN

LHL R13,0DETMAP
GET DEVICE VARIABLE MAP

AHR R13,R13
ADD DISPLACEMENT

LH R14,0(R14)
GET ADDRESS OF DVT

STM R10,REGSAVE(R14)
SAVE REGISTERS

XMR R13,R13
CLEAR DONE FLAG

STM R13,CKSUMFLG(R14)
CLEAR CHECKSUM FLAG

STM R13,DONEFLG(R14)
CLEAR DONE FLAG

STM R11,LENGTH(R14)
STORE IT

LH R11,LENGTH(R14)
GET LENGTH OF SCB

STM R11,LENGTH(R14)
STORE IT

STM R10,OUTSCB(R14)
ADD LENGTH

STM R11,0
ADD LENGTH

STM R11,0SCBND(R14)
STORE ENDING ADDRESS OF SCB

SINT 0(R12)
START TRANSMISSION OF SCB

SINT 0(R12)
START TRANSMISSION OF SCB

LH R13,CARRFLG(R14)
GET CARRIER FLAG

LH R10,1
SET CARRIER FLAG

LH R10,1
SET CARRIER FLAG

CLHI R13,0
LOSED CARRIER

AX2 LH R10,0DETFLG(R14)
GET DONE FLAG

AX2 LH R10,0DETFLG(R14)
GET DONE FLAG

CLHI R10,1
CON #

BNE AX1
BRANCH IF NOT

XMR R10,R10
RCODE = OK

AXRTN STH R10,0(R1)
PUT RCODE IN PARM LIST

AXRTN STH R10,0(R1)
PUT RCODE IN PARM LIST

LM R10,REGSAVE(R14)
RESTORE REGISTERS

B 0(R15)
RETURN
RECEIVE INTERRUPT SERVICE ROUTINE

RCV HANDLES ALL INTERRUPTS COMING FROM THE RECEIVE SIDE OF THE PASLA. IF THE ACLDR IS IN THE RECEIVE MODE THEN THE CHARACTERS ARE READ IN OTHERWISE THE INTERRUPTS ARE IGNORED.

ACLDR85
ASYNCHRONOUS CONTROL LINE DRIVER

0240R 480E 0000
0244R 9004
0248R 4220 0254H
024CR 0744
0250R A4F 0000
0254R 43F4 0000
0258R 404E 001A
025CR 45E2 0020
0260R 9003
0264R 9003
0268R 503E
026CR 485E 0012
0270R 0714
0274R 40AC 0004
0278R 0535
027AR 4200 02BC
027CR 436C 0012
0280R 2551
0284R 605C 0014
0288R 450F 0000
028CR 403C 0012
0290R 403C 0022
0294R 4A90
0298R 40AC 000A
029CR 43F0 0000
029DR 406C 0004
029FR 406C 0008
02A0R 4210 0070CR
02A2R 487C 0022
02A4R C700 0FF
02A8R 0537
02AAR C377
02ADR 4230 02C2R
02B0R 0777
02B2R 607C 001A
02B4R 4000 02C6R
02B6R 407C 001A
02B8R C670 0001
02B9R C670 0001

337
338
339
340
341
342
343
344
345
346
347
348
349
350

351 RCV LH R13,RCVADDR(R14) GET RECEIVE ADDRESS
352 RC1 SSR R13,R4 GET STATUS
353 DFC 2*X2 BRANCH IF CARRIER IS ON
354 XHR R4,R4
355 STH R4,CARRFLG(R14) CLEAR CARRIER FLAG
356 B 0(R15) RETURN
357 RC2 XHR R4,R4
358 CLH R4,MODE(R14) IN RECEIVE MODE?
359 BE 0(R15) IGNORE INTERRUPT IF NOT
360 ADR R13,R3 READ CHARACTER
361 LH R5,LENGTH(R14) LENGTH = 0 IF FIRST CHAR
362 CLIH R5,0
363 BNE RC4 BRANCH IF NOT FIRST CHAR
364 LH R5,MAXSIZE(R14) GET MAXIMUM SIZE OF BUFFER
365 LH R10,INSG(R14) GET SCS ADDRESS
366 STH R10,INPTR(R14) SET INPUT POINTER
367 CLMR R3,R5 LENGTH = MAXSIZE
368 BNP RC3 BRANCH IF LENGTH <= MAXSIZE
369 STH R3,LENGTH(R14) STORE LENGTH
370 LIS R5,1
371 B R5,LENFLG(R14) GET MAX LENGTH FLAG
372 RETURN
373 RC3 STH R3,LENGTH(R14) LENGTH = CHARACTER CHECKSUM = LENGTH
374 STH R3,CHECKSUM(R14) ADD SCS ADDRESS AND LENGTH
375 AHR R10,R3 STORE SCS ENDING ADDRESS
376 B 0(R15) RETURN
377 RC4 LH R6,INPTR(R14) GET SCS POINTER
378 CLH R6,INSG(R14) BUFFER FULL?
379 B R7 RC7
380 LH R7,CHECKSUM(R14) GET CHECKSUM
381 AHI R7,X’0FF’+ CHECKSUM OK?
382 CLMR R3,R7 CHECKSUM FLAG = OK
383 BNE RC5 BRANCH IF NOT
384 XHR R7,R7
385 STH R7,CHECKFLG(R14) CHECKSUM FLAG = ERROR
386 LH R7,R1 CHECKSUM OK?
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>02CEC</td>
<td>407E</td>
<td>001C</td>
<td></td>
</tr>
<tr>
<td>02D2H</td>
<td>0777</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02D4H</td>
<td>407E</td>
<td>0020</td>
<td></td>
</tr>
<tr>
<td>02D8H</td>
<td>430F</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>02CAH</td>
<td>0236</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>02E8H</td>
<td>613E</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>02EAH</td>
<td>406E</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>02E9H</td>
<td>430F</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>391</td>
<td></td>
<td>STH</td>
<td>R7.DOONEFLG(R14)</td>
</tr>
<tr>
<td>392</td>
<td></td>
<td>XHR</td>
<td>R7+R7</td>
</tr>
<tr>
<td>393</td>
<td></td>
<td>STH</td>
<td>R7+MODE(R14)</td>
</tr>
<tr>
<td>394</td>
<td></td>
<td>B</td>
<td>0(R15)</td>
</tr>
<tr>
<td>395</td>
<td></td>
<td>STR</td>
<td>R3+O(R6)</td>
</tr>
<tr>
<td>396</td>
<td></td>
<td>ADD</td>
<td>R3+CHECKSUM(R14)</td>
</tr>
<tr>
<td>397</td>
<td></td>
<td>AIS</td>
<td>R6+1</td>
</tr>
<tr>
<td>398</td>
<td></td>
<td>STH</td>
<td>R6+INPTR(R14)</td>
</tr>
<tr>
<td>399</td>
<td></td>
<td>RRATN</td>
<td>0(R15)</td>
</tr>
</tbody>
</table>

RETURN
ASYNCHRONOUS CONTROL LINE DRIVER

**TRANSMIT INTERRUPT SERVICE ROUTINE**

- **XNT** handles all interrupts coming from the transmit
- **SIDA** of the PABLA. The initial interrupt caused by **DATA TERMINAL READY** becomes active is ignored and the rest are used to transmit the characters.

```assembly
02CEC 40DE 0001E 013 XNT 015, INITFLG(R14) GET INITIALIZATION FLAG
02EFR 0000 0001E 014 LH R13, INITFLG(R14) GET INITIALIZATION FLAG
02FAR 430F 0001E 016 CLHI R13, 1 FIRST INTERRUPT?
0300R 430F 00000 017 BE X00 BRANCH IF NOT
0300R 430F 00000 018 LH R13, 1 RETURN
0300R 421F 00000 019 STH R13, INITFLG(R14) SET INITIALIZATION FLAG
0300R 421F 00000 020 D 0(R15) RETURN
0300R 421F 00000 021 X00 LH R13, HCVADDR(R14) GET PASLA ADDRESS
0300R 421F 00000 022 LH R12, HXTADDR(R14) GET TRANSIT ADDRESS
0300R 421F 00000 023 SRR R13, R2 GET STATUS
0300R 421F 00000 024 DFC 2X0 BRANCH IF CARRIER IS ON
0300R 421F 00000 025 XHR R2, R2 CLEAR CARRIER FLAG
0300R 421F 00000 026 STH R2, CARRFLG(R14) RETURN
0300R 421F 00000 027 B 0(R13)
0300R 421F 00000 028 X0 LH R0, LENGTH(R14) GET LENGTH FLAG
0300R 421F 00000 029 CLHI R0, 1 HAS LENGTH BEEN SENT
0300R 421F 00000 030 BE X1 BRANCH IF YES
0300R 421F 00000 031 LH R0, LENGTH(R14) GET LENGTH
0300R 421F 00000 032 STH R0, CCHKSUM(R14) STORE IN CHECKSUM
0300R 421F 00000 033 WUR R12, R0 TRANSMIT LENGTH
0300R 421F 00000 034 LHI R0, 1 SET LENGTH = 1
0300R 421F 00000 035 LH R10, OUTPTR(R14) GET ADDRESS OF OUTPUT BUFFER
0300R 421F 00000 036 STH R10, OUTPTR(R14) RELOAD OUTPUT POINTER
0300R 421F 00000 037 X1 LH R10, OUTPTR(R14) GET OUTPUT BUFFER POINTER
0300R 421F 00000 038 CLH R10, OUTPTR(R14) BUFFER EMPTY?
0300R 421F 00000 039 BM R3 BRANCH IF NOT
0300R 421F 00000 040 XHR R0, R0
0300R 421F 00000 041 CLH R0, CKSFMLG(R14) HAS CHECKSUM BEEN SENT?
0300R 421F 00000 042 BE X2 BRANCH IF NOT
0300R 421F 00000 043 LHI R0, 8 CLEAR CKSFMLG(R14)
0300R 421F 00000 044 STH R0, CKSFMLG(R14) CLEAR CKSFMLG
0300R 421F 00000 045 STH R0, LENGTH(R14) CLEAR LENGTH
0300R 421F 00000 046 LH R7, 1 SET DOIF FLAG
0300R 421F 00000 047 STH R7, DOIFFLG(R14) SET DOIF FLg
0300R 421F 00000 048 B 0(R15) RETURN
0300R 421F 00000 049 X2 LH R6, CCHKSUM(R14) GET CHECKSUM
0300R 421F 00000 050 WUR R12, R6 TRANSMIT CHECKSUM
0300R 421F 00000 051 LH R6, 1 SET CHECKSUM FLAG
0300R 421F 00000 052 STH R6, CKSFMLG(R14) SET CHECKSUM FLAG
0300R 421F 00000 053 B 0(R15) RETURN
```
ASYNCHRONOUS CONTROL LINE DRIVER

0380R 26S 000A
0390R 6158 0029
0380R 4945
0380R 26A1
0380R 4040 000A
0390R 4300F 0000A
0390R 430F 0000A

455 X3 L8 R5,0(R10)
456 A0H R0, CHECKSUM(R14)
457 WDR R12,R5
458 AIS R10,1
459 STH R10, OUTPTR(R14)
460 XRTW B 0(R15)
461 END

GET CHARACTER
ADD TO CHECKSUM
TRANSMIT CHARACTER
INCREMENT OUTPTR
RETURN
APPENDIX C

ACLDR Concurrent PASCAL Code

The following listing is the Concurrent PASCAL version for the ACLDR
* * * * ASTRONOMICAL CONTROL LINE DRIVER * * * *

This driver consists of three classes, ACL_SETUP, ACL_RCV, and ACL_XMT. Each class can be called by the asynchronous communication processes to set up a PASLA and transmit and receive control messages through that PASLA.

---

IO_MACHINE COMMANDS USED BY THE INTERPRETER

---

```plaintext
CONST

IO_CHANNEL_IMMEDIATE"(DEVICE_INDEX, VALUE)" = 01
IO_CHANNEL"(DEVICE_INDEX, ADDRESS)" = 11

IO_COMMAND_IMMEDIATE"(COMMAND BYTE, VALUE)" = 21
IO_COMMAND"(COMMAND_BYTE, ADDRESS)" = 31

IO_SENSE "=RETURN_device STATUS BYTE=" = 41
IO_SENSE_EXTERNAL"(STATUS_BYTE, ADDRESS)" = 51

IO_COMPARE_IMMEDIATE"(STATUS_MASK, VALUE)" = 61
IO_COMPARE"(STATUS_MASK, ADDRESS)" = 71

IO_JUMP"(IO_INSTRUCTION_INDEX, VALUE)" = 81
IO_JUMP_TRUE"(IO_INSTRUCTION_INDEX, VALUE)" = 91
IO_JUMP_FALSE"(IO_INSTRUCTION_INDEX, VALUE)" = 101

IO_RETURN "=EXITS IO_INTERPRETER=" = 111
IO_RETURN_TRUE = 121
IO_RETURN_FALSE = 131

IO_DATA_IMMEDIATE"(IN, BYTE, BYTE2, BYTEN)" = 141
IO_DATA"(BUFFER BYTF, LENGTH, BUFFER ADDRESS)" = 151

IO_WAIT_INTERRUPT = 161

IO_START_INPUT "=SETS UP & STARTS AUTOCH, MUX, SELCH=" = 171
IO_START_OUTPUT = 181

IO_PREEMPT "=EMERGENCY DEVICE CONTROL=" = 191
IO_FREE "=EMERGENCY OVER, GIVE DEVICE BACK=" = 201

CONST
ASYNC_LINE1_IN = 71 "X'10' = ASYNCHRONOUS LINE PASLA 10"
ASYNC_LINE1_OUT = 81 "X'11' = ASYNCHRONOUS LINE PASLA 11"
```

TYPE
S_RETURN_CODE = (S0, S1, S2);
TYPE RCV_RETURN_CODE = (R0, R1, R2, R3, R4);
TYPE XMT_RETURN_CODE = (X0, X1);
TYPE CH_WORD = ARRAY [1..2] OF CHAR;

CONST
UNEX_INTERRUPT = 7681;
COMMAND_TIMEOUT = 12001;
CHANNEL_TIMEOUT = 15361;
NO_COMMAND = 2061; "NO OP FOR START I/O"

TYPE STRING76 = ARRAY [1..76] OF CHAR

<<<< COMMENTS >>>>

WHENEVER AN INTERRUPT IS GENERATED BY A DEVICE AND
NO PROCESS IS WAITING FOR THAT INTERRUPT THEN AN
INTERNAL FLAG IS SET. WHEN A PROCESS CALLS THE
IO_MACHINE WITH THAT DEVICE INDEX THE IO_MACHINE
IMMEDIATELY RETURNS CONTROL AFTER THE IO_CHANNEL
INSTRUCTION WITH AN INTERNAL STATUS OF 3 IN THE
STATUS WORD. THESE TYPES OF INTERRUPTS ARE IGNORED
BY THE ACLR SO THE IO_MACHINE IS REENTERED AFTER
DETECTION OF THIS STATUS.
**** ACL_SETU P ****

ACL SETUP IS CALLED TO SET UP A PASLA AND ENABLE
INTERRUPTS FOR THE RECEIVE AND TRANSMIT SIDES OF
THAT PASLA.

TYPE SET_UP_CLASS = CLASS1
      MAX_INSTRUCTION = 71
      "COMMAND 1 BITS"
      TENABLE = 641 "INTERRUPTS ON"
      TCTR = 321 "DATA TERMINAL READY"
      TWRT = 21 "WRITE MODE"
      TRD = 61 "READ MODE"
      TCHND1 = 11 "COMMAND 1"
      "COMMAND 2 BITS"
      THIGH = 641 "HIGH BAUD RATE"
      TBITS = 481 "8 BIT DATA"
      T2STOPS = 641 "2 STOP BITS"

TYPE
IO_TYPE = RECORD
  IC, "IO INSTRUCTION COUNTER"
  STATUS, "IO MACHINE STATUS ON RETURN"
  PERFER, INTEGER
  INTERNAL: ARRAY 1..13 OF INTEGER
  INST: ARRAY 0..MAX_INSTRUCTION) OF INTEGER
END "IO_TYPE"

VAR IO_RECORD: IO_TYPE:
COMMAND_FLAG: BOOLEAN:

PROCEDURE ENTRY ACL_SETUP(LINE_ADDRESS: INTEGER;
VAR RCODE: S.RETURN.CODE);

BEGIN WITH IO_RECORD DO
BEGIN
  COMMAND_FLAG := TRUE;
  REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
  IC := 01; STATUS := 01;
  INST[1] := LINE_ADDRESS;
  INST[2] := TENABLE+TCTR+TWRT+TCHND1;
  IC := IO_INSTRUCTION:
  IO(IO_RECORD) := "TURN ON PASLA"
  IF (STATUS >= COMMAND_TIMEOUT) AND
  (STATUS < CHANNEL_TIMEOUT) THEN
    COMMAND_FLAG := FALSE;
  UNTIL (STATUS <> UNEX_INTERRUPT);
  REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
  IC := 01; STATUS := 01;
  INST[1] := LINE_ADDRESS + 11;
  INST[3] := TENABLE+TCTR+TWRT+TCHND1;
  IC := IO_WAIT_INTERRUPT;
  IO(IO_RECORD) :=
  UNTIL (STATUS <> UNEX_INTERRUPT);
  IF COMMAND_FLAG THEN RCOD := 50 ELSE RCODE := 51;
END;
END.
BEGIN "INIT"
WITH IO.RECORD DO
BEGIN
INSTR[0] := IO_CHANNEL_IMMEDIATE;
INSTR[23] := IO_COMMAND_IMMEDIATE;
INSTR[30] := NEW=TAGITS+T2STOPS;
INSTR[40] := IO_COMMAND_IMMEDIATE;
INSTR[73] := IO_RETURN;
END "SET UP CLASS";
END
TYPE RCV_CLASS = CLASS;

CONST
MAX_INSTRUCTION = 81;

TYPE
IO_TYPE = RECORD
IC: "IO INSTRUCTION COUNTER"
STATUS: "IO MACHINE STATUS ON RETURN"
XFER: INTEGER;
INTERNAL: ARRAY [1..13] OF INTEGER;
INST: ARRAY [0..MAX_INSTRUCTION] OF INTEGER;
END "IO_TYPE";

VAR IO_RECORD: IO_TYPE;
POS: INTEGER;
CHECKSUM: INTEGER;
IN_LENGTH: INTEGER;
IN_CHECKSUM: INTEGER;
TEMP_CHECKSUM: INTEGER;

PROCEDURE RBYTE(I: INTEGER; OUT_W: UNIV.CH_WORD); VAR OUT_W: UNIV.CH_WORD;
"BYTE IS USED TO PLACE THE UPPER BYTE OF AN INTEGER INTO THE LOWER ONE AND ZERO OUT THE UPPER BYTE."

BEGIN
OUT_W := "(0111)";
OUT_W := OUT_W;
END;

PROCEDURE ENTRY ACL_RCV(ACL: ACL_SYSTEM; INTEGER: INTEGER; STRING76: STRING; VAR LENGTH: INTEGER); MAXI: INTEGER; VAR CODE: RCV.RETURN_CODE)
BEGIN WITH IO_RECORD DO
BEGIN
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01;
INST1 := LIN_ADDRESS;
INST2 := IO_STATUS;
INST3 := IO_STATUS;
INST4 := IO_STATUS;
UO((IO_STATUS);
UNTIL ((STATUS < UNEX_INTERRUPT);
INST5 := IO_DATA;
IN_LENGTH := 01;
IF NOT ((STATUS DIV 2) THEN
BEGIN
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01;
INST1 := 11;
INST2 := 11;
INST3 := ADDRESS(IN_LENGTH);
INST4 := ADDRESS(IN_LENGTH);
INST5 := RECEIVE_LENGTH;
"RECEIVE LENGTH OF INCOMING SCB"
"THE LENGTH IS RECEIVED IN THE UPPER BYTE OF THE INTEGER SO BYTE IS CALLED TO PLACE IT IN THE LOWER BYTE."
BYTE(IN_LENGTH,LENGTH);
UNTIL (STATUS <> UNEX_INTERRUPT);
END;
IF LENGTH <= MAX THEN

BEGIN
TEMP_CHECKSUM := LENGTH;
IF NOT ODD (STATUS DIV 2) THEN
BEGIN
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01; STATUS := 01;
INST(3) := LENGTH;
INST(4) := ADDRESS (BUFFER);
IO(IO_RECORD) := "RECEIVE SCAN";
UNTIL (STATUS <> UNEX_INTERRUPT);
END
IF NOT ODD (STATUS DIV 2) THEN
BEGIN
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01; STATUS := 01;
INST(3) := 11;
INST(4) := ADDRESS (IN_CHECKSUM);
IO(IO_RECORD) := "RECEIVE CHECKSUM"
"THE CHECKSUM IS RECEIVED IN THE UPPER BYTE
OF THE INTEGER SO ANYTE IS CALLED TO PLACE
IT IN THE LOWER BYTE."
BYTE (IN_CHECKSUM; CHECKSUM);
UNTIL (STATUS <> UNEX_INTERRUPT) ;
END
TEMP_CHECKSUM := 01;
POS := 11;
WHILE (POS <= LENGTH) DO
BEGIN
TEMP_CHECKSUM := TEMP_CHECKSUM + ORD (BUFFER [POS]);
POS := POS + 11;
END
TEMP_CHECKSUM := TEMP_CHECKSUM + LENGTH;
TEMP_CHECKSUM := TEMP_CHECKSUM MOD 256;
IF TEMP_CHECKSUM = CHECKSUM THEN
RCODE := R0;
ELSE RCODE := R21;
END
IF ODD (STATUS DIV 2) THEN RCODE := R11;
END
ELSE RCODE := R41;
END
END
BEGIN "INIT"
WITH IO_RECORD DO
BEGIN
INST(0) := IO_CHANNEL_IMMEDIATE;
INST(3) := IO_WAIT_INTERRUPT;
INST(4) := IO_START_INPUT;
INST(7) := IO_COMMAND;
INST(10) := IO_RETURN;
END "WITH IO_RECORD"
END "RCV_CLASS";
ACL_XMT IS CALLED TO START THE TRANSMISSION OF A CONTROL MESSAGE AND SEE TO ITS COMPLETION.

TYPE XMT_CLASS = LASS;
CONST
RSY_STAT = 001; "PASLA BUSY STATUS"
MAX_INSTRUCTION = 141; "MAXIMUM IO INSTRUCTIONS"

TYPE
IO_TYPE = RECORD
IC, "IO INSTRUCTION COUNTER"
STATUS, "IO MACHINE STATUS ON RETURN"
IN_XFER, INTEGER; "NUMBER OF BYTES TRANSFERRED"
INTERNAL_ARRAY[1..13] OF INTEGER;
INST, ARRAY 0..MAX_INSTRUCTION OF INTEGER
END "IO_TYPE";
VAR IO_RECORD: IO_TYPE; POS: INTEGER; CHECKSUM: INTEGER;
OUT_CHECKSUM: INTEGER; OUT_LENGTH: INTEGER;

PROCEDURE XBYTE(IN_LW: UNIV.CH_WORD; VAR OUT_LW: UNIV.CH_WORD);
"XBYTE IS USED TO PLACE THE LOWER BYTE OF AN INTEGER INTO THE UPPER BYTE."
BEGIN
OUT_LW2 := T(IN_LW1);
OUT_LW1 := IN_LW2;
END;

PROCEDURE ENTHY ACL_XMT(LINE_ADDRESS: INTEGER; VAR BUFFER: STRING761;
VAR LENGTH: INTEGER; VAR ICODE: XMT_RETURN_CODE);
BEGIN WITH IO_RECORD DO
BEGIN
"CHECK STATUS FOR LOSS OF CARRIER"
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01; STATUS := 01;
INST(11) := LINE_ADDRESS;
INST(12) := IO_SENSE;
INST(13) := IO_RETURN;
ID(IO.RFCORD);
WHILE (STATUS <> UNEX_INTERRUPT) DO
IF NOT (STATUS = 012) THEN
BEGIN
INST(11) := LINE_ADDRESS + 11
INST(12) := IO_DATA;
CHECKSUM := LENGTH;
POS := 11;
" FIGURE CHECKSUM "
WHILE (POS <= LENGTH) DO
BEGIN
CHECKSUM := CHECKSUM + ORD(BUFFER(POS));
POS := POS + 11
END;
" XBYTE IS CALLED SO THAT THE LENGTH WHICH IS ONE BYTE LONG CAN BE SENT OUT OF THE UPPER BYTE."
XBYTE(CHECKSUM,OUT_CHECKSUM);
" TRANSMIT LENGTH "
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01; STATUS := 01;
INSTRU := 11;
INSTRU[0] := ADDRESS([OUT,LENGTH]);
"XBYTE IS CALLED SO THAT THE LENGTH WHICH IS ONE
BYTE LONG CAN BE SENT OUT OF THE UPPER BYTE;"
INSTRU[1] := ADDRESS([OUT,LENGTH]);
IO(IO_RECORD); "TRANSMIT LENGTH OF SCB"
UNTIL (STATUS <> UNEX_INTERRUPT);
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01; STATUS := 01;
INSTRU := LENGTH;
INSTRU[0] := ADDRESS([BUFFER]);
IO(IO_RECORD); "TRANSMIT SCB"
UNTIL (STATUS <> UNEX_INTERRUPT);
REPEAT "IGNORE ANY PREVIOUS INTERRUPTS"
IC := 01; STATUS := 01;
INSTRU := 11;
INSTRU[0] := ADDRESS([OUT,CHECKSUM]);
IO(IO_RECORD); "TRANSMIT CHECKSUM OF SCB"
UNTIL (STATUS <> UNEX_INTERRUPT);
RCODE := 10
END
ELSE RCODE := 11
END;

BEGIN "INIT"
WITH IO_RECORD DO
BEGIN
INSTRU[0] := IO_CHANNEL_IMMEDIATE;
"IF BUSY THEN WAIT_INTERRUPT"
INSTRU[1] := IO.Compare_IMMEDIATE;
INSTRU[8] := IO.Compare_IMMEDIATE;
INSTRU[9] := IO.Compare_IMMEDIATE;
INSTRU[10] := IO.Compare_IMMEDIATE;
INSTRU[12] := IO.START_OUTPUT;
INSTRU[14] := IO.START_OUTPUT;
END "WITH IO_RECORD";
END "INIT";
END;
HOST/CC AND CC/CC ASYNCHRONOUS CONTROL LINE DRIVER IN THE MIMICS NETWORK

by

ERWIN LYNN REHME

B.S., Kansas State University, 1975

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1977
Abstract

This report contains a description of the design and implementation of an asynchronous control line driver in the MIMICS network. The driver handles the functions necessary for transmitting and receiving of control information between computers within a cluster of the network. In the report we give a brief description of the MIMICS network and how the driver is used in that network. We then describe the use of asynchronous lines for communication, why they were chosen for this particular project, and how they are programmed on the Interdata 35 and the Interdata 7/16. It also tells how the computers were wired together to insure that the interface boards could detect abnormal conditions of the line. The implementation of the driver on the Interdata machines using assembler language and PASCAL is then presented, followed by a summary of the work completed and some extensions to conclude the report.