CLARKSON COLLEGE OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

A MICROPROCESSOR BASED NETWORK:

The Clarkson Loop

A Thesis

by

RUSSELL NEIL NELSON

Submitted in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

(Electrical and Computer Engineering)

October 1980

Accepted by the Graduate School

_____ _____

Date Dean

Abstract

This thesis details the implementation of a distributed microcomputer system for the IEEE S-100 bus. The system consists of hardware and software whose function is to transmit and receive messages. The design of the interface hardware, the software driver, and the protocol are discussed. An example of usage of the system is given.

Acknowledgements

First, let me give special recognition to my thesis advisor, Dr. David Bray, and my good friend, Paul Austin, without whom I would not have attended graduate school in the first place. I am very grateful to the members of my thesis committee, Dr. Susan Conry, and Dr. Mohammed Arozullah, for the time that they spent reviewing this document. I would also like to thank my parents, Russell and Gladys Nelson for their undying support.

I thank the myriad unnamed faculty, graduate students, and undergrads who supported me with their friendship and interest in my thesis. Many thanks also to the good, kind, wonderful person who wrote the word processor used to print this thesis.

Table of Contents

1.1 Introduction	1
2.1 Network Architecture	4
3.1 Hardware Requirements	10
3.2 Functional Description	10
4.1 Software Requirements	16
4.2 North Star Operating System	17
5.1 Protocol	19
5.2 User Interface	20
5.3 Sending Messages	21
5.4 Receiving Messages	23
6.1 Example of Clarkson Loop Usage	27
7.1 Conclusions	30
References	32
Appendix A - Interface Schematic	33
Appendix B - I/O Interface routines	39
Appendix C - Loop Handlers	44
Appendix D - LOGDISK	55

List of Figures

2.1 Bus and Loop 6

2.2 Clarkson Loop 8

3.1 Block Diagram of Interface 11

5.1 Message Format 19

5.2 Reception State Diagram 24

6.1 Message Flow 28

List of Tables

3.1 Port Addresses 12

3.2 Interrupt Masks 14

6.1 Message Contents 29

1.1 Introduction

This thesis describes the design and implementation of a microcomputer network consisting of both hardware and software. The purpose of the network is to allow parallel processing and sharing of computer resources by using an inexpensive, simple, and expandable communication system.

The software developed includes an interrupt-driven version of the North Star Operating System, routines to send and receive messages, and an example program which exercises the capabilities of the operating system. The implemented hardware consists of an interface board which meets the design criteria.

The design criteria used in developing this system are:

1) The network itself must be low cost because the microcomputers of the network are themselves inexpensive. The interfaces between them must be proportional in cost, or the system will not be economical.

2) The network must be incrementally expandable so that more processors can be added without reconfiguring the network.

3) The software to handle network communication must not require large amounts of memory or computing time, otherwise the network will interfere with the normal operations of the processor.

4) Communications between any two processors must be independent of any other processors because some network processors may not be operating.

There are many ways to interconnect computers, only some of which are suitable in the context of microcomputers. To satisfy the requirement that the network be independent of any single computer, there must be a direct link between any two computers, with no intervening computers. This requirement eliminates many interconnection schemes. Among the exceptions are the fully connected network, the bus, and the

loop. The fully connected network can be removed from further consideration because the interface to connect each computer to all of the others is expensive.

The bus has been successful in its past usage, and several standards have been set for buses: CAMAC, IEEE-488, and MIL-STD-1553A [Weitzman 1980]. None of the standards are useful to us because their interfaces are too complicated, and the data transfer protocols are too deeply enmeshed in the buses application. The advantage of using a bus in our application is that any two processors can communicate over a bus provided that control can be arbitrated so that no more than one processor can transmit data at any one time.

Three types of loop architecture have achieved popularity: the Newhall type, Pierce type, and Delay Insertion type. These loops are distinguished by the way they control the flow of data from node to node. Loop architectures are advantageous because control is easy to arbitrate, and loops are easy to expand.

This thesis describes the design and implementation of the Clarkson Loop, a loop which can be considered an extension of the Newhall loop. Its merits are discussed in the first chapter.

Chapter Two develops the rationale behind the choice of network architectures. The alternatives are considered, and the ones which do not meet our requirements are eliminated. The actual architecture is explained in detail, and similarities are drawn between a loop and a bus.

Chapter Three describes the interface between each processor and the loop, and gives reasons why a special interface is needed. A block diagram of the interface is shown and explained. The operation of each of the blocks is given in detail.

Chapter Four sets forth the software requirements. The degree of

support required from the existing operating system is also described.

Chapter Five describes the data transfer protocol, the operation of the low level operating system software, and the user interface.

In chapter Six an example of a loop application is given: a disk drive simulator whih allows one computer to use another's disk drive as through the drive were the computer's own.

2.1 Network Architecture

There are many ways to interconnect computers, only some of which meet the constraints of our application. For the reasons given in the introduction, the interprocessor interface must have as its first criterion low cost because the microcomputers themselves are inexpensive. A second criterion should be ease of expansion, and a third should be simplicity of interconnection interface.

For the purposes of this discussion, we shall consider an interconnection of processors to be represented by a graph. Each processor is a node of the graph and each interconnecting communication link is an arc. The terms processor and node are used synonymously.

The most obvious and versatile of interconnections is the fully interconnected network. In such a network, every processor is connected to every other processor with a separate link for each. The problem with this scheme is that as the number of nodes in the network goes up, the interconnection cost goes up as n squared, where n is the number of processors. Each new processor adds n new interfaces, so the number of arcs in a fully interconnected network is of order n^2. Therefore, the fully interconnected network does not meet any of our three requirements.

A method similar to fully interconnected is the partially interconnected method, in which at least two arcs exist between each and every node of the network. For the same reasons as the fully interconnected network, the partially interconnected network is not suitable. The partially interconnected network also introduces the problem of routing a message. Seldom is there a direct path between each pair of computers, so a message usually must be sent to at least one intermediate node. This implies that each computer must have a descrip-

tion of the network graph in order to send messages to their proper destination.

The next network to be considered is the star, a network in which each node is only connected to one common node, called the central node. Although this architecture is very popular, there are several reasons why is not suitable for our purposes. First, each time the network is expanded, an additional interface for the central node must be purchased. Second, the central node must be capable of switching the messages from one node to another. This means that the central node must have enough processing power to interpret the destinations for messages and redirect them to these destinations. The central node must therefore be "more powerful" then the other processors. This is not intuitively appealing. Third, the processor must be able to handle each message as it comes in. If too many messages arrive at once, the processor can become saturated. This problem was encountered by Downer [Downer, 1980]. Some of the processing can be relegated to external hardware, however, this increases the cost.

The only types of interconnections meeting our criteria lie in the various forms of the bus, and the loop, as will be shown. A bus is characterized by a set of nodes each connected by a single arc to another set of special nodes, each of which is connected to each other. The latter type of node is not a processing node, but simply connects each entering arc together. The loop is characterized by a graph in which each node is connected to only two others, so that the arcs of the graph form a cycle. Both the bus and loop networks are shown in Figure 2.1 (a node which is not a processor is shown as a circle with an X in the center).

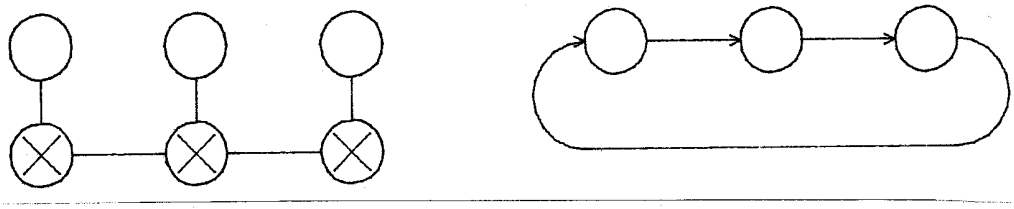


Figure 2.1 - Bus and Loop

In the bus, information flows from a single transmitter through the special nodes to all other receivers immediately. In the loop, the information is transmitted from node to node, with each node deciding independently whether or not to transmit the information on to the next node.

One characteristic of the bus architecture which makes it successful is its simplicity. All that is needed is a set of wires, an interface in each node which can listen, and sometimes transmit on the bus, and a protocol to control which processor is allowed to transmit. We would like to capture that simplicity.

The most well known of the loops include the Pierce, Newhall, and Distributed Loop Computer Network (DLCN). In both the Pierce [Jafari 1978] and the Newhall [Farmer 1969] loops, control is determined by the location of a token, which is usually a special character, but may be as simple as a logic level. When the loop is initialized, one of the processors is initialized to have the token. The loop handling software never creates another token, so there is never more than one token in the loop.

Messages are of fixed length in the Pierce loop, while the Newhall

and DLCN loops allow variable length messages. In order to give the user as much flexibility as possible, the length of the messages should be variable. This rules out the Pierce loop.

The DLCN [Liu 1977] uses a technique known as delay insertion. When an interface wishes to transmit, it waits for the end of the current message. The interface then transmits its message. If a message arrives during the transmit time, it is buffered, and passed on after the current message. Therefore, any number of messages may be in transit at once. The interface uses a bit slice processor and costs about five hundred dollars. We feel that this is too expensive for microprocessor applications. Of course, it is possible to implement the DLCN using the microprocessor, rather than a dedicated processor. The reason we do not desire to do this is that the loop would demand too much attention from the processor. This is contrary to our goals, therefore the DLCN is not usable.

The Newhall loop allows variable length messages, but only allows one message on the loop at a time. The processor which puts that message on the loop has posession of the token. When the processor is finished sending, it passes on the token to the next processor on the loop. Each processor receives the message in turn and determines if the message is addressed to itself. If so, the message is removed from the loop, and if not, the message is sent on to the next processor.

The Newhall loop meets most of our requirements. The loop is incrementally expandable, and can be made independent of any single processor. The loop selected for our application is a Newhall loop augmented by an additional control loop, as shown in Figure 2.2.

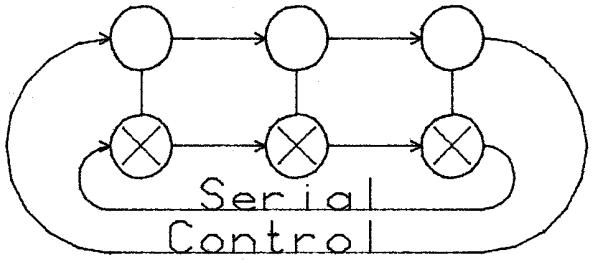


Figure 2.2 - Clarkson Loop

In the Clarkson loop there are in effect two loops. One line carries serial data, and the other arbitrates control. The serial data line is similar to a bus in that its information is received at all nodes at once. The control line has properties generally ascribed to a loop in that the token is sent from node to node, with each node determining when the token is sent on.

The serial data line is used to transmit characters serially. When a processor gains control of the loop, the serial data line is broken at that node only. The rest of the processors leave the line closed, so that characters that are transmitted are received immediately by all processors.

Once the transmission has been completed, the loop is free. This is because the time a message spends in transit is nearly zero. The receiving processor can then send an acknowledge character indicating that the message was received properly.

The control line is used to transmit a token from one node to another. The processor which currently holds the token is the processor in control of the loop. A processor may transmit a message only when it is in control of the loop. When a processor is finished transmitting,

it sends the token on to the next processor in the loop.

The details of hardware implementation are given in the next chapter.

3.1 Hardware Requirements

The Clarkson Loop is independent of any single microprocessor. This independence is made possible by the use of a special interface board. The interface is designed to be transparent to the loop if the interface does not have power applied or if the microprocessor is not ready to service the interface. The requirements of this interface board are given next.

In our software architecture we shall consider the normal activity of the host computer to be called the background job. The purpose of the operating system is to allow the user's background job to execute while allowing other users to access his peripherals [Brinch-Hansen 1973]. To avoid disrupting the host computer's activities, the transfer of data to and from the interface must be done using the interrupt system rather than a software sense loop, since the background job would stop running if the processor had to wait for characters in a sense loop. To ensure that the interface will function on a computer without an interrupt generator, interrupt generation circuitry is included on the Clarkson Loop interface board.

A special interrupt must be caused when a character which marks the beginning of message is received. This ensures that a processor can recognize the beginning of a message. Software must be written which compares the address in the destination field of the message with the address of the processor. This function could be performed in hardware, but this would unnecessarily complicate the board. If a message is not addressed to the host, the processor should not be bothered by a message generated interrupt. This is in keeping with the goal of minimum interruption of the host processor. An interrupt mask is desirable to eliminate unnecessary interrupts.

3.1 Hardware Requirements

3.2 Functional Description

The interface circuit consists of several major sections. These are, in the order in which they will be explained: the address decoder, the UART, the baud rate generator, the message detector, the token handler, the interrupt generator, and the current loop drivers. A block diagram of the interface is given in Figure 3.1 and its schematic is given in Appendix A.

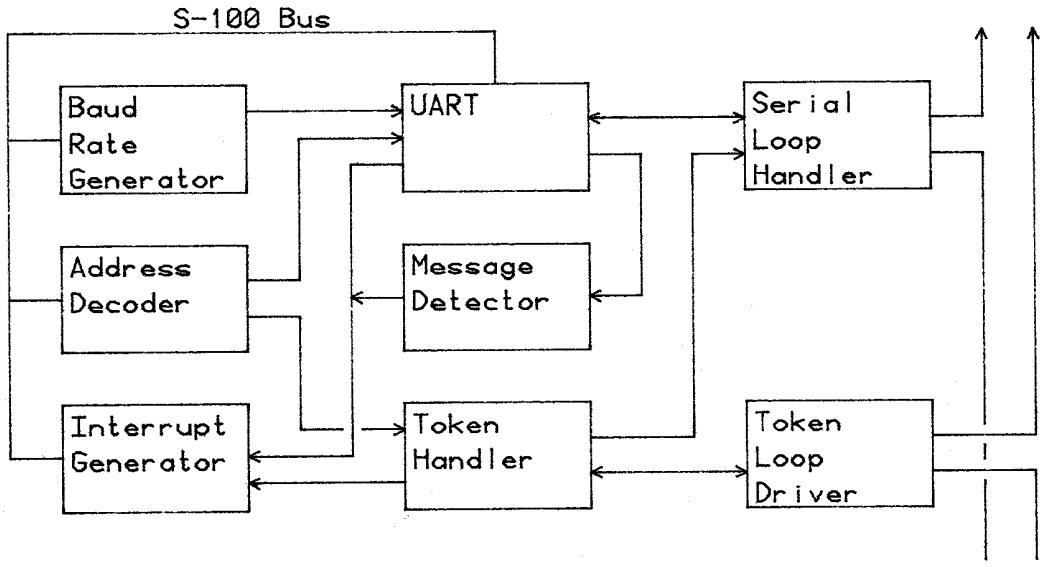


Figure 3.1 - Block Diagram of Interface

Address Decoder

The 8080A has an I/O address space which uses eight bits. The interface board responds to four of the 256 possible addresses. Address bits A0 and A1 (the two low order address bits) are sent to the decoder to determine which of the four ports are being addressed. The rest of the address bits, A2 through A7, are compared to the contents of a DIP switch using six exclusive-or gates. The outputs of the exclusive-or

gates form the board addressed signal. To ensure that the board is only activated on input and output instructions of the 8080A, the board addressed signal is ANDed with SIN/ and SOUT/ to form the board enable signal (notation: SIN/ denotes the complement of SIN). One of the two signals SIN or SOUT is brought high by the processor whenever an input or an output instruction takes place, respectively.

The board enable signal, the processor write signal (PWR-L), and the processor read signal (PDBIN) are sent to the three to eight decoder. A pulse is formed when either PWR-L or PDBIN become true. These two signals are shorter than other signals sent to the decoder, hence they determine the length of the pulse.

The output of the address decoder is a set of eight active low pulses. Port zero will cause the UART to be enabled, and port one will cause the status port to be read or the interrupt mask to be written. Ports two and three do not use the data bus; they are control signals. Simply reading or writing to these two ports will set or clear the token handler flip flops, called Loop Available and Loop Desired. The port assignments are shown in Table 3.1

Port	Input	Output
0	UART status	interrupt mask
1	character in	character out
2	clear LA	set LD
3	set LA	clear LD

Table 3.1 - Port Addresses

UART

The UART section consists of one integrated circuit, which is complicated enough to warrant its own description. The Universal Asynchronous Receiver and Transmitter, UART, is a five volt version of the industry standard 1602 [Osbourne 1977]. The UART is configured to

send and receive eight bits of data, with even parity, and one stop bit. The parallel input data to the UART is received from the 8080A bus through buffers. The parallel output data is sent to the 8080A bus through Tristate buffers. The input serial data and output serial data go to the loop signal drivers which implement a 20 milliampere current loop. The UART has several status lines, which can be read by the computer using port zero. The receive and transmit clocks, which must be set at sixteen times the desired baud rate, come from the baud rate generator circuit which is described below.

Baud Rate Generator

The baud rate generator is a simple circuit, consisting only of a counter chip, dividing down the system clock (2 Mhz) by thirteen. This generates sixteen times the baud rate of 9600. We chose to use this transmission speed because it is slow enough to eliminate errors. The UART can operate at up to 300 kilobaud.

Message Detector

The message detector is a set of exclusive-or gates which compare the output of the UART to the contents of a DIP switch. The output of this circuit goes high when a beginning of message character (BOM), which is encoded into a DIP switch, is received. This output is sent to the status port, and is also sent to the interrupt generator. To ensure that the interrupt will no longer be requested once it has been acknowledged, the interrupt request is ANDed with the character received bit from the UART.

Token Handler

The token handler determines what happens to the token once it is received. The token is a one hundred microsecond long pulse which travels from computer to computer. There are two flip flops which control the token pulse's travel. These two flip flops are called LOOP

DESIRED and LOOP AVAILABLE. They can be directly set and cleared using ports two and three. If they are both clear, the token pulse will travel on to the next computer immediately. If LOOP DESIRED is set, the token pulse's arrival sets LOOP AVAILABLE, and the token pulse is held. When they are both set, an interrupt is generated. The interrupt handler will clear LOOP DESIRED to reset the interrupt. The token pulse (and control of the loop) will remain until LOOP AVAILABLE is cleared.

Interrupt Handler

The interrupt generator is responsible for raising the 8080A interrupt line and placing the proper restart instruction on the 8080A bus when the interrupt is acknowledged. Four interrupts are generated, with four more reserved for future expansion. These are, from zero to three, beginning of message (BOM), loop available (LPA), character available (RDA), and transmitter buffer empty (TBE). Interrupt zero is the highest priority, and interrupt three is the lowest priority.

A mask is available which can disable interrupts of a given priority and lower. A mask of zero will disable all interrupts, a mask of one will allow interrupt zero and no other, etc. This mask can be set by an output to port one. A table giving the interrupts allowed for each particular mask is in Table 3.2

Mask	Interrupts allowed
0	none
1	0 (BOM)
2	0 (BOM), 1 (LPA)
3	0 (BOM), 1 (LPA), 2 (RDA)
4	0 (BOM), 1 (LPA), 2 (RDA), 3 (TBE)

Table 3.2 - Interrupt Masks

The interface board must be initialized at power up. Both of the token control flip flops are reset. If operation with the loop is not

desired, the interrupt mask can be set to zero, which will turn off all interrupts. If operation with the loop is to be allowed, the interrupt mask is set to one. This allows only beginning of messages to be received.

The next chapter describes the software requirements.

4.1 Software Requirements

A message consists of two parts, the header, and the body. The header is used to convey information about the body of the message, such as its destination and length. The body of the message is the actual information which is transferred. A string consists of any number of bytes concatenated together. These bytes are often ASCII characters, in which case a string is termed text.

Each computer is identified by a unique number which is programmed into its read only memory. This prevents having to re-identify each computer whenever it is powered up. This number is termed the processor address.

The receipt of a message is acknowledged by the receiver sending a character, any character, back to the transmitting processor. If the message is not received correctly, the character is not transmitted, and the transmitting processor knows that the message was not received. When a message is to be sent to all computers at once, a processor address of zero is used and no acknowledge is expected. This operation is termed broadcasting.

One basic design criterion for the software was that it should be capable of sending and receiving variable length messages. Also, the bytes of the string may take on any value from zero to 255. Variable length messages are desirable because they allow the most flexibility for the user. Additionally, such messages do not have to be split into blocks to fit into a fixed size, thus saving operating system software.

There are several methods of delimiting a string commonly in use. The method most often used for text is to group it into a set of lines, each ending in the ASCII carriage return symbol. This method is not appropriate in our situation because a message may contain a data byte

whose value is the ASCII carriage return. Unfortunately, the problem cannot be solved by choosing another character to delimit the string, because any character may appear in binary data strings.

Another string delimiting method involves maintaining a count of bytes contained in the string. The count is then transmitted along with the string itself. This is the method implemented in the Clarkson Loop, because it is the most versatile.

Recall that in Chapter Two, when the hardware was discussed, a message detector circuit was introduced. The purpose of this circuit is to generate an interrupt when an ASCII STX (02 Hex), which we call BOM, is received. This creates a problem when binary data is transmitted. If the value 02 Hex is transmitted as a byte of a string, the receiver interface will generate a BOM interrupt. This BOM interrupt will cause an error since it is not the beginning of a message.

There is a simple solution for this problem. Whenever a 02 Hex is to be sent as part of a string, two characters are sent in its place: an ESC (1B Hex) and a capital B (42 Hex). When an ESC is to be transmitted, two characters are sent: an ESC, and a [(5B Hex). The receiver routine can then detect when an ESC arrives and check the next character to determine the message's content. The binary data can then be recovered.

4.2 North Star Operating System

The computers which are to be used in this application are S-100 bus computers: IMSAI 8080's, each with a North Star floppy disk system. The North Star Operating System (DOS) is a minidisk based operating system. In this operating system, routines are included to create and modify a disk directory and read and write files or arbitrary sectors.

All parameters to these routines are passed through the 8080A registers.

It has already been shown that the interrupts must be enabled for the distributed network to function. However, the North Star operating system does not normally support interrupts. The reason for this (as far as DOS is concerned) is that interrupts must be turned off during disk access because there is not enough time to acknowledge them.

The North Star operating system used for the Clarkson Loop has been modified to turn the interrupts off and on again at the proper times. To create an interrupt driven North Star operating system, the original DOS supplied by North Star Computers was disassembled. Instructions to disable and enable interrupts were then placed at the proper points and DOS was reassembled.

Included in DOS is an empty area of memory in which the user places terminal I/O routines. In this area (which was expanded when DOS was reassembled) the interrupt driven I/O routines were placed. Not only does this provide the necessary interrupt handling but it also allows the user to enter text even while the loop is being serviced.

In addition to the I/O routines, the loop driver routines (described below) are included as part of DOS. To ensure that both DOS, the terminal I/O, and the loop drivers are all loaded at the same time and at the proper locations, they are all assembled into the same object file. When a loop computer is initially powered up the augmented interrupt driven DOS is loaded into memory. Thus the loop driving routines are available without additional action.

A listing of the DOS interrupt driven I/O software is given in Appendix B.

5.1 Protocol

The data transfer protocol has deliberately been kept as simple as possible to allow the user flexibility in his own design. Information is sent in messages of variable length, up to 2^{16} bytes.

A header usually must contain more than the destination address, and the message length. For generality, information to differentiate between message types must also be included in the header part of the message. This can be done by the user, but it is easier for the operating system to provide this service. Therefore, a Clarkson Loop message header also contains a command byte which differentiates between message types.

In order to process messages with different command bytes differently, each command byte has a buffer and message processor associated with it. As a message arrives, it is stored in the buffer for that command byte. When the message is complete, the proper message processor is called to dispose of the message.

The message format is shown in Figure 5.1, in this figure, DEST denotes the destination byte, CMD indicates the command byte, Len low gives the least significant byte of the length, and Len high gives the most significant byte of the length.

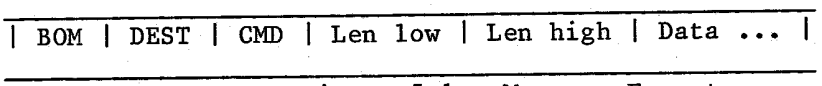


Figure 5.1 - Message Format

The message length is a two byte number stored in standard 8080A low-high fashion [INTEL 1977]. The command byte's value may be any byte value. These desired values are defined at the time the loop drivers

are assembled. The currently defined values are zero through seven.

5.2 User Interface

There are two user callable routines associated with Clarkson Loop use. They are GRAB and ADDR, which are used to transmit and receive messages respectively. These routines, along with the different interrupt handlers, comprise the loop driver software.

Routine: GRAB

Address: 29E2H

Entry: B register = command byte

DE register = address of SEND

HL register = address of DONE

Exit: No special conditions

GRAB is used to send a message. Operation of the two routines SEND and DONE are dependent upon the user's application. They are written by the user, and the minimum function that they must perform is to return the values indicated.

Routine: SEND

Address: determined by user

Entry: No special conditions

Exit: HL register = address of message

SEND is called when the operating system is ready to send the message. SEND merely returns the address of the message.

Routine: DONE

Address: determined by user

Entry: B register = Boolean value

Exit: HL register = address of message

DONE is called when the message has been sent. The B register indicates whether or not the message was acknowledged by the receiver. A Boolean variable is true if it is not zero, and false if it is zero. DONE should be written to retransmit the message if it was not acknowledged.

Routine: ADDR

Address: 29E5H

Entry: B register = command byte

DE register = address of the receive buffer.

HL register = address of RECV

Exit: No special conditions

ADDR defines the receive buffer and message processor for the specified command byte. The routine RECV must be written by the user.

Routine: RECV

Address: determined by user

Entry: HL register = address of message received

Exit: No special conditions

RECV must dispose of the message before it exits because the message may be written over when RECV exits. The operating system prevents the reception of another message until RECV returns.

5.3 Sending Messages

The GRAB routine is the only sending routine which is explicitly called by the user. It causes an LPA interrupt to occur. The LPA in-

terrupt handler causes a TBE interrupt. The TBE interrupt handler causes more TBE interrupts until the message has been transmitted completely. In this manner, the background job continues to execute between loop interrupts.

Four routines are explained next: the GRAB routine, and three interrupt handlers. The words set or clear will be used in conjunction with hardware, the symbol := will be used to denote the replacement of a program variable, and curly braces will be surround comments.

```
GRAB routine:
  set interrupt priority to 2; {enable LPA}
  set LOOP DESIRED; {hold token when it arrives}
  return;
```

When the token arrives LOOP AVAILABLE is set.
A LPA interrupt is generated.

```
LPA interrupt:
  clear LOOP DESIRED; {reset interrupt}
  set interrupt priority to 4; {enable TBE}
  call SEND; {determine address of message}
  byte_count:=number of bytes in message
  byte_point:=address of message.
  Transmit BOM; {start chain of interrupts}
  return;
```

When a character has been sent, a TBE interrupt is generated.

```
TBE interrupt:
  if byte_count = 0 then
    set interrupt priority to 0; {done transmitting}
    set Timer; {wait for acknowledge character to arrive}
  else
    transmit contents of byte_point; {transmit next character}
    byte_count:=byte_count-1;
    byte_point:=byte_point+1;
  return;
```

When the timer times out, a timer interrupt is generated.
The timer is provided by the TU-ART board.

```
Timer interrupt:
  clear LOOP AVAILABLE; {release token}
  if acknowledge received then B:=1 else B:=0;
  call DONE;
  return;
```

5.4 Receiving Messages

Receiving messages is more complicated than transmitting them, since each of the characters in the header must be treated differently. To accomplish this in the loop driver routines, a finite state machine is simulated. A variable called STATE indicates the state of the machine. When a character is received, this state variable is used to determine how the character is treated.

The state diagram showing the possible state changes is given in Figure 5.2. The initial state is zero, indicating that a BOM character is expected next. There are two transitions not shown on the state diagram: state zero is entered as explained in the next paragraph, and state one is entered whenever a BOM is received.

Both overrun and parity errors cause an immediate branch to state zero and the rest of the message is ignored. An overrun error will occur if a new character is received before the previous character has been accepted. The previous character is now lost. A parity error will occur if some bits are garbled in transmission. In either case, the message has been adulterated, and is ignored.

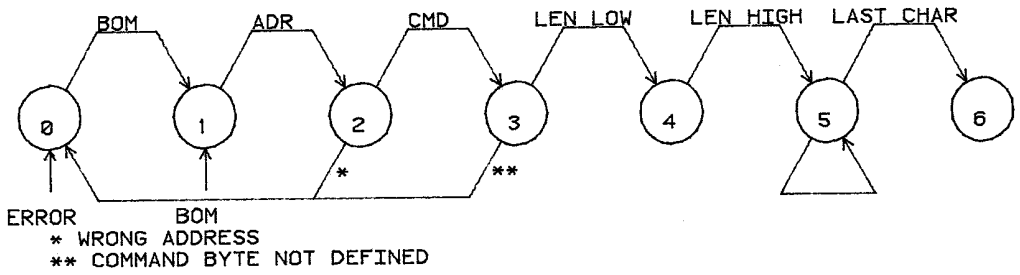


Figure 5.2 - State Diagram

Incoming messages are handled by the BOM and RDA interrupt handlers.

```
BOM interrupt handler:
  if STATE = 6 then return; {if
already processing message, return}
  STATE:=1;
  set interrupt priority to 3; {enable RDA}
  return;
```

```
RDA interrupt handler:
  dispose of character according to STATE as detailed below
  return;
```

The action of the state machine is described as follows.

State zero - Waiting for BOM.

Receipt of a BOM will cause the BOM interrupt which sets STATE equal to one, and sets the receive interrupt priority to three, so that characters can be received.

State one - Waiting for destination.

If the message is not addressed to this computer, and is not being broadcast, the message is ignored. This is done by returning to state zero and setting the receive priority back to one. If the message is addressed to this computer, or is a broadcast, STATE is set to two.

State two - Waiting for command byte.

The table containing the buffer addresses for each command byte is referenced. If the address has not yet been defined by a call to ADDR, the message is rejected, otherwise the first byte received (the destination byte), and the second (the command byte) are stored in the buffer and STATE is set to three.

States three and four - Waiting for size.

The length is two bytes long. The first byte is the least significant, the second is most significant. The message length indicates the number of data bytes following. They are received by setting STATE to five.

State five - Waiting for data.

The routine stays in state five until the message is completed. When the last character has been received, STATE is set to six.

State six - Message received.

An ACK character is transmitted to acknowledge the receipt of the

message. The RECV routine associated with the command byte of this message is called. When it returns, STATE is set to zero.

A listing of the loop driver routines is given in Appendix C.

6.1 Example of Clarkson Loop Usage

As an example of the use of the Clarkson Loop, a demonstration user program has been implemented for the Clarkson Loop operating system. The program, called DISKSIM, simulates multiple disk drives on a single disk drive system. One processor, designated the master, uses another processor's disk drives. The other processors are termed slaves.

DISKSIM is useful because multiple disk drives are useful. It is possible, but not convenient to copy files from one floppy disk to another on a single drive system. By simulating two drives, system routines can be used to copy files, or even a whole disk. The true beauty of DISKSIM is that nondistributed programs can use it without modification.

A North Star disk controller can control up to three drives, therefore to differentiate between the physical floppy disk drives, any disk access must specify a drive number. The computers used in the loop each have only one disk drive.

DISKSIM works in conjunction with DOS by overlaying the DOS disk driver entry point. When a user program calls the disk driver, DISKSIM is entered.

When DISKSIM gains control, the registers have been loaded with parameters for the disk driver routine. One of these parameters is the drive number, which is used as an index into MAPTBL. MAPTBL contains entries for four drive numbers. Drive number one is the drive which is connected to the master processor. Drives two through four are mapped into processors one through three using a table kept in DISKSIM.

If drive one is accessed, the master processor's drive is used. If drives two through four are accessed, messages are transmitted to and from the slave processor corresponding to that drive number. These

messages are listed in Figure 6.1. The contents of each of the messages are listed in Table 6.1.

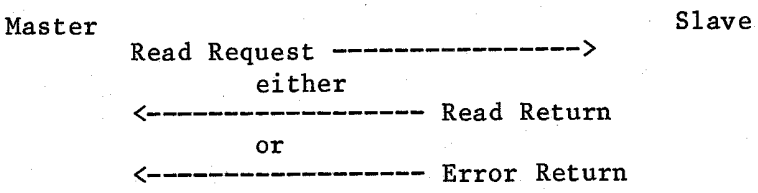
It is easy to see that no modification is necessary to a user program in order to use DISKSIM. All that a user program need do is to access the second or third disk drives as if the disk drives were attached to the master computer.

In order to hold buffer space to a minimum, one sector (256 bytes) is transmitted on the loop at a time. If the disk command requests more than one sector to be read or written, a separate message is sent for each sector.

An Error return message is sent if a hard disk error occurs on either a read or a write. When the message arrives at the master processor, the hard disk error routine is called.

A listing of the routine is given in Appendix D.

Messages sent to read a sector:



Messages sent to write a sector:

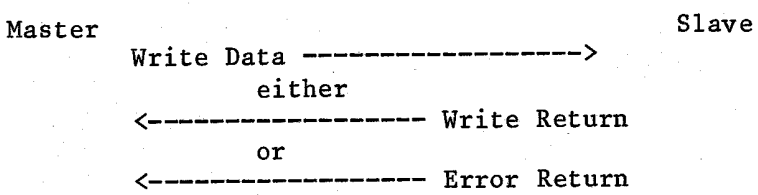


Figure 6.1 - Message Flow

Five messages can be sent and received:

Type of message	Contents
Read data	disk address.
Read return	disk data.
Error return	bad track,bad unit,bad sector.
Write data	disk address, sector data.
Write return	none

Table 6.1 - Message Contents

Conclusions

The interface board was specified, designed, and built by Paul Austin and the author. Since we were gaining experience in designing computer interfaces, a number of errors were made. Some of these prevented the operation of the board, and were removed. Others still existing make usage of the board more inconvenient, but not impossible.

In particular, the board is not initialized at power up. When the operating system is initialized, the interface must also be initialized, otherwise spurious interrupts will be generated. This problem can be alleviated by using the power on clear line of the S-100 bus.

When the processor is reset, execution resumes at location zero. The interface uses this location for one of its interrupt addresses. Moving all of the interrupts up by one (i.e. making the BOM a restart one, the LPA a restart two, etc.) will eliminate the conflict.

There is still one awkward situation remaining with the token. When the loop is first powered up, the token is started by manually initializing only one processor to have the token. This processor then releases the token without sending a message. The token then travels normally around the loop. Unfortunately, the token disappears when a computer is powered down, and must be manually restarted.

In a laboratory environment this problem is not restrictive, as all the computers in the loop are in the same room. The problem of the lost token must be solved because it is inconvenient to restart it manually in a practical application. One possibility is to suspend loop activity when a computer is being turned off. Another is to never turn computers off, a practice followed with mainframe computers.

A second distributed processing application has already been developed by Paul Austin [Austin 1980]. A LISP interpreter was modified

to include two functions, write message and read message. LISP functions were written to distribute the evaluation of LISP sub-functions.

The Clarkson Loop is useful any time that data must be transmitted from one computer to another. Applications include distributed processing and multiuser programs. If the computers were located some distance from each other, another possible application might involve electronic mail transmission.

References

Austin, Paul R. "Distributed Processing of Applicative Expressions" - Master's Thesis, ECE Dept., Clarkson College, Potsdam, N. Y., May 1980

Downer, Gregory "a Microcomputer Network for Studying Distributed Data Base Problems" - Master's Thesis, ECE Dept., Clarkson College, Potsdam, N. Y., April 1980

Brinch Hansen, Per "Operating Systems Principles" - Prentice Hall, 1973

INTEL Corporation "MCS80 User's Manual" - Intel Corporation, Oct. 1977

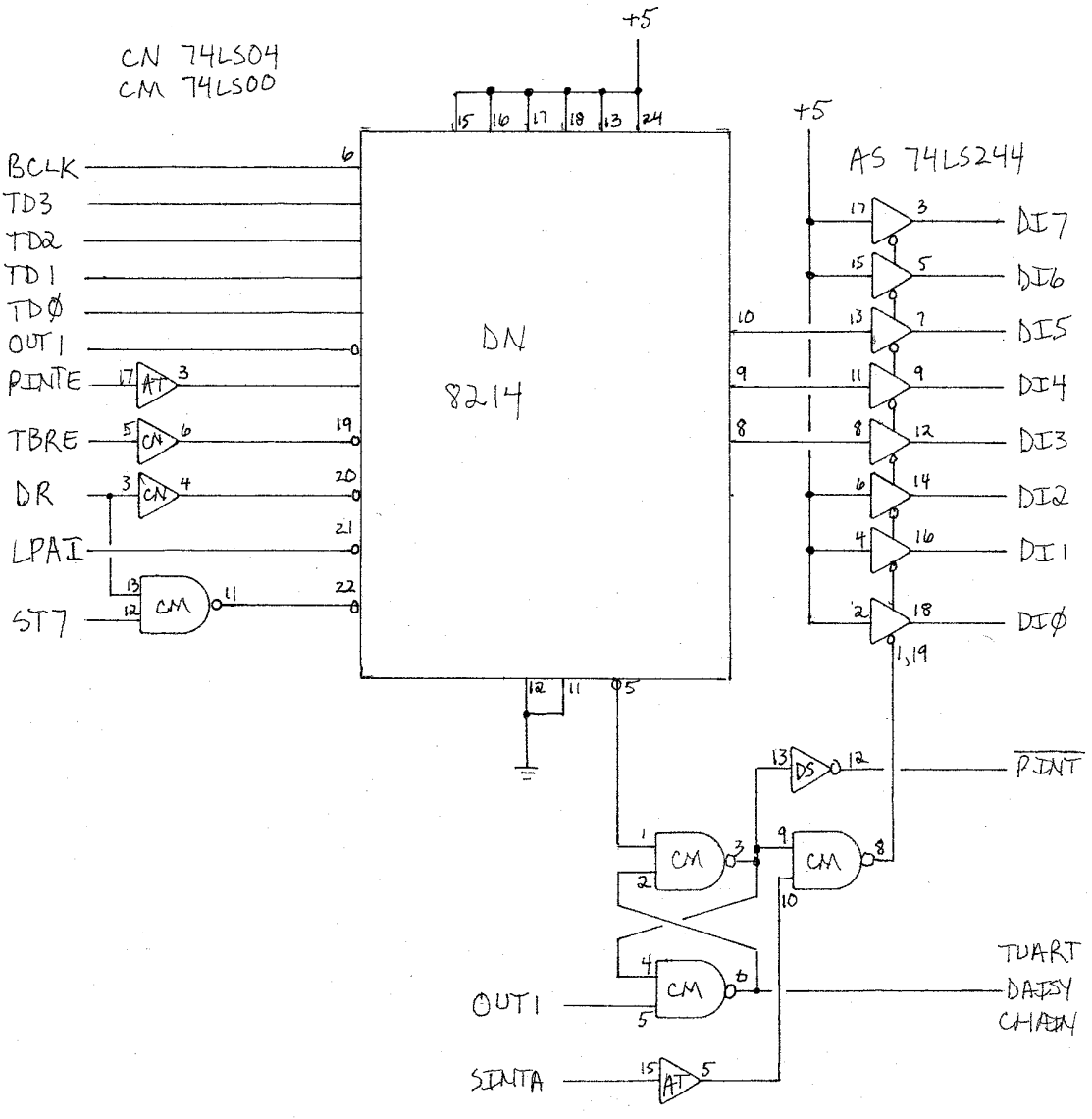
Liu, Ming T. "Distributed Loop Computer Networks" - Advances in Computers, Vol. 17, 1977

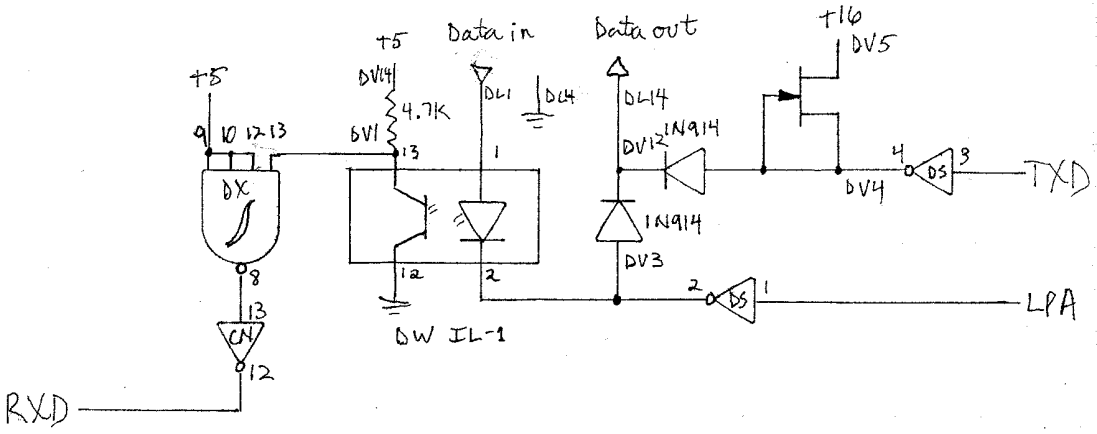
Farmer, W. D. and Newhall, E. E. "An Experimental Distributed Switching System to Handle Bursty Computer Traffic" - Proc. ACM Symposium on Problems in the Optimization of Data Communication Systems, Pine Mountain, Georgia, Oct. 1969

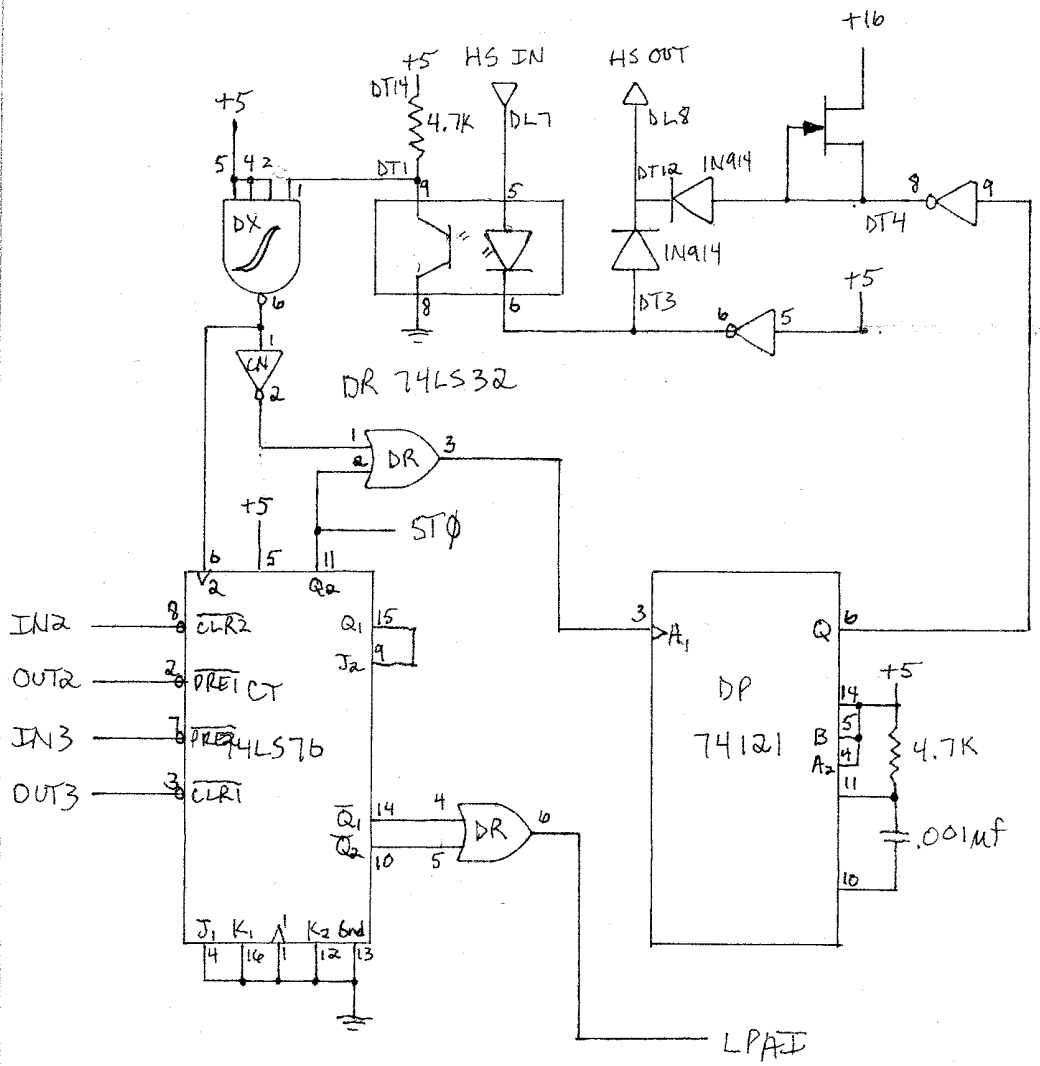
Osbourne, Adam "An Introduction to Microcomputers, Volume II, Some Real Products" - Adam Osbourne and Associates, 1977

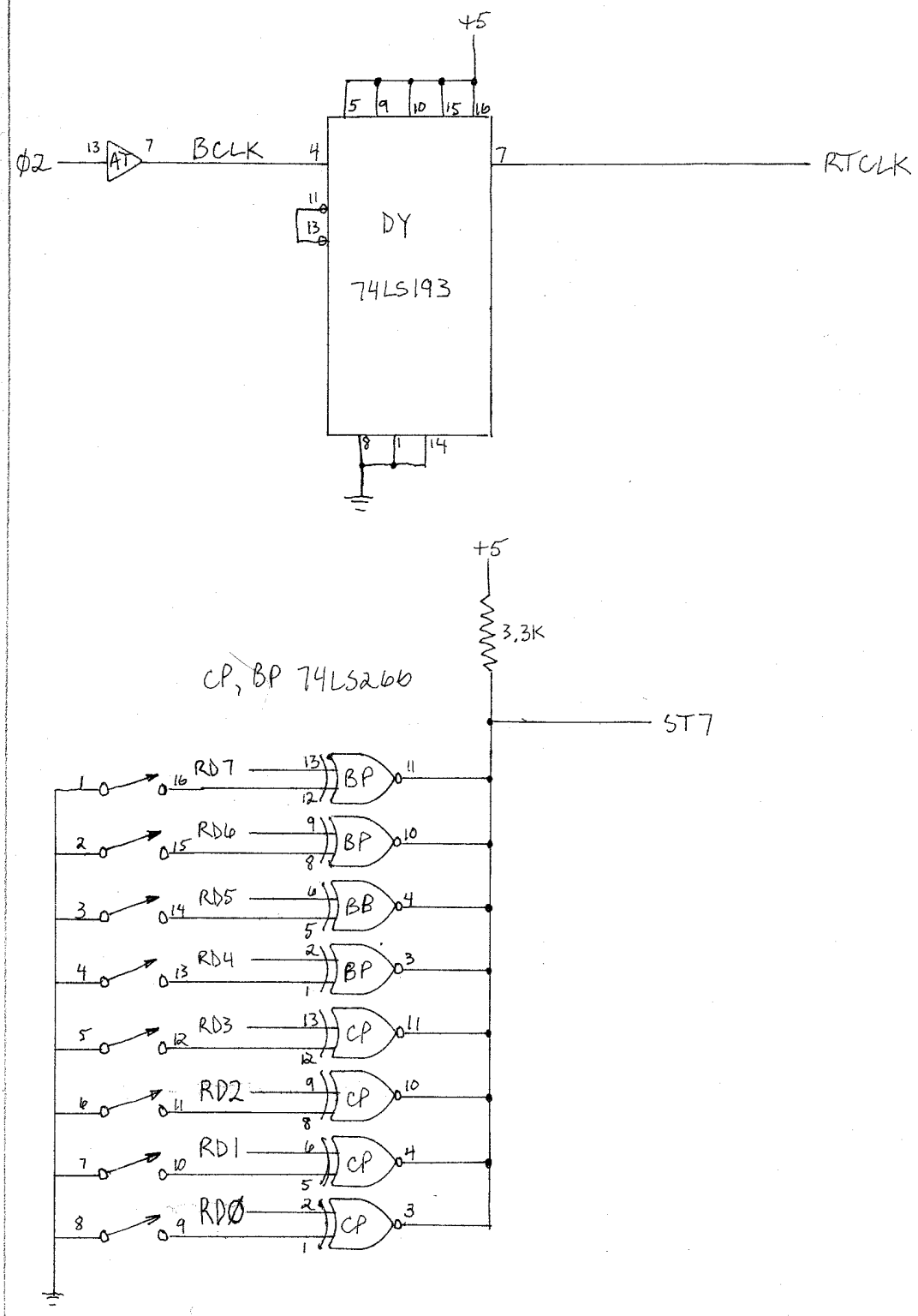
Jafari, H., Spragins, J. and Lewis, T. "A New Modular Loop Architecture for Distributed Computer Systems" - IEEE Trends and Applications in Distributed Processing, May 1978

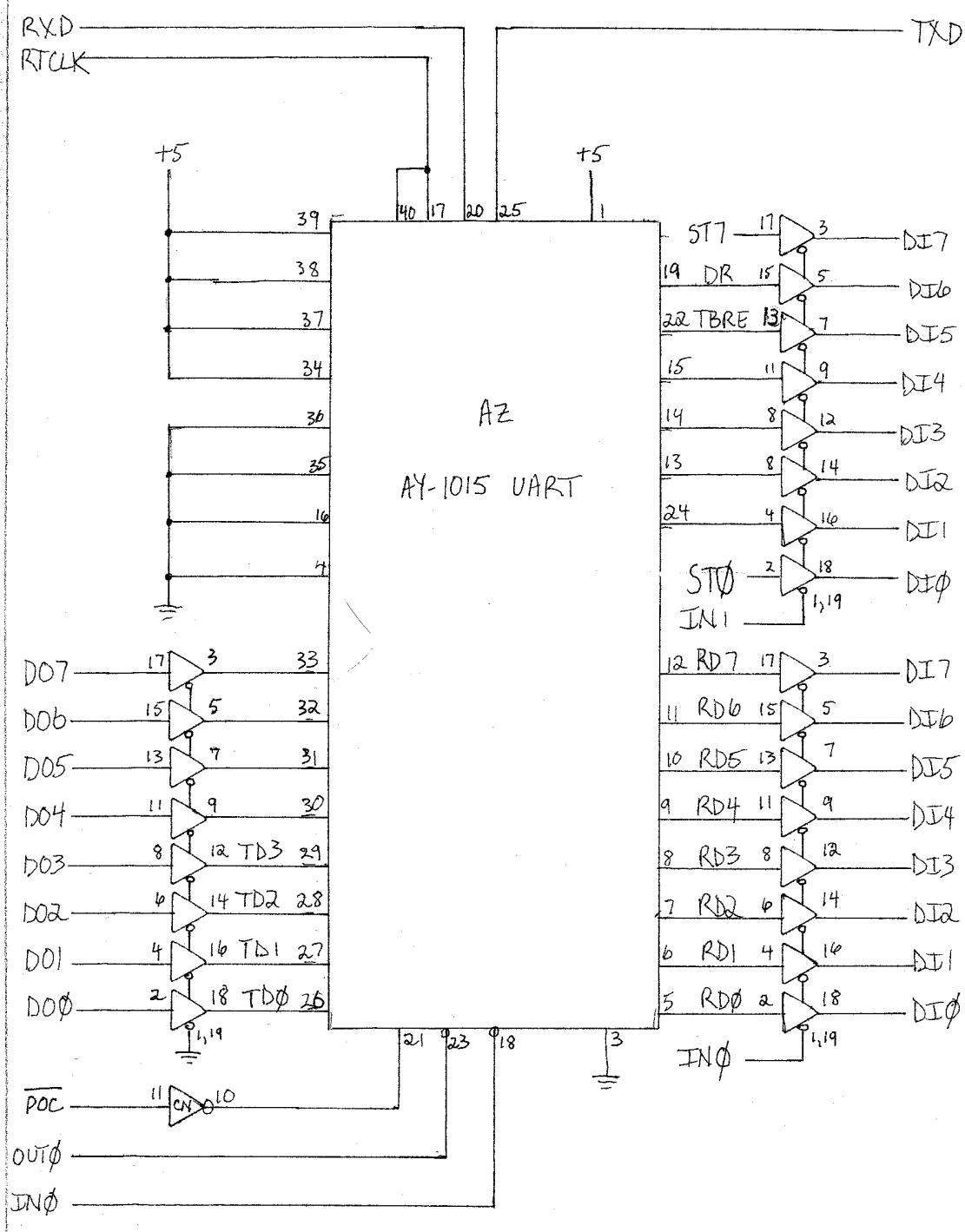
Weitzman, Cay "Distributed Micro/Minicomputer Systems" - Prentice-Hall Inc., Englewood Cliffs, N. J., 1980

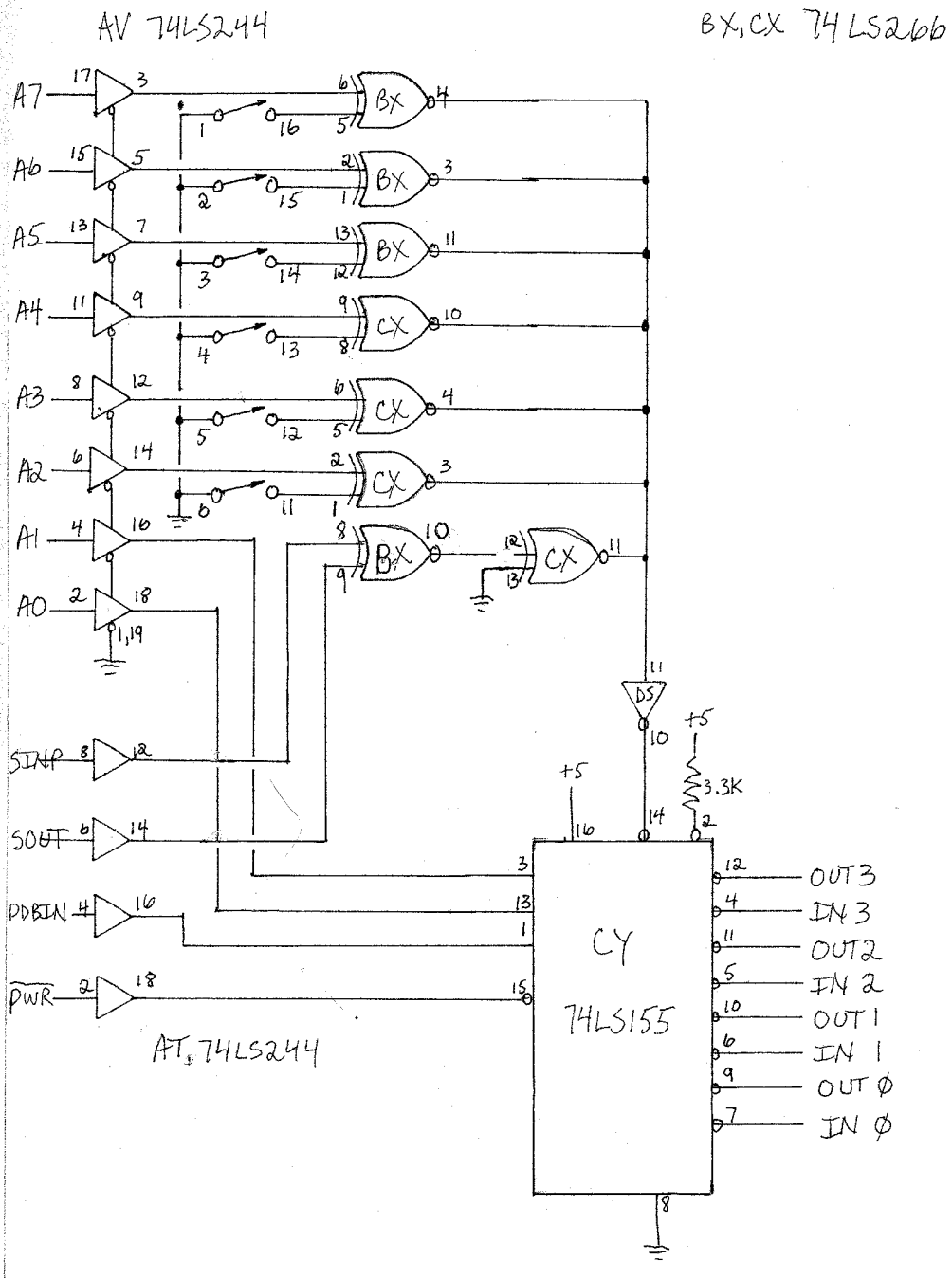

CN 74LS04
CM 74LS00
+5
+5
AS 74LS244
DN
8214
BCLK
TD3
TD2
TD1
TD0
OUT1
PINTE
TBRE
DR
LPAI
ST7
DI7
DI6
DI5
DI4
DI3
DI2
DI1
DI0
PINT
OUT1
SINTA
TUART
DAISY
CHAIN


+5
Data in
Data out
+16
DV5
4.7K
DV1
DL1
DL4
DL14
DV2
IN914
DV4
DS
TXD
DX
IN914
DV3
DW IL-1
LPA
CN
RXD


+16
+5
HS IN
HS OUT
DT14
4.7K
DL7
DL8
DT1
DT12
IN914
DT4
DX
IN914
DT3
+5
DR 74LS32
DR
STϕ
IN2
OUT2
IN3
OUT3
CLR2
PRE1
74LS76
CLR1
DP
74121
4.7K
.001μf
LPAI


+5
5
9
10
15
16
φ2
13
AT
7
BCLK
4
7
RTCLK
11
13
DY
74LS193
8
1
14
+5
3.3K
CP, BP 74LS266
ST7
1
16
RD7
13
12
BP
11
2
15
RD6
9
8
BP
10
3
14
RD5
6
5
BB
4
4
13
RD4
2
1
BP
3
5
12
RD3
13
12
CP
11
6
11
RD2
9
8
CP
10
7
10
RD1
6
5
CP
4
8
9
RDØ
2
1
CP
3


RXD
RTCLK
TXD
+5
+5
AZ
AY-1015 UART
39
38
37
34
36
35
16
4
40
17
20
25
1
ST7
DR
TBRE
STØ
IN1
DI7
DI6
DI5
DI4
DI3
DI2
DI1
DIØ
DO7
DO6
DO5
DO4
DO3
DO2
DO1
DOØ
TD3
TD2
TD1
TDØ
RD7
RD6
RD5
RD4
RD3
RD2
RD1
RDØ
INØ
POC
OUTØ
INØ
21
23
18
3


AV 74LS244
BX, CX 74LS266
A7
A6
A5
A4
A3
A2
A1
A0
BX
CX
DS
+5
3.3K
SINP
SOUT
PDBIN
PWR
AT 74LS244
CY
74LS155
OUT3
IN 3
OUT2
IN 2
OUT 1
IN 1
OUT Ø
IN Ø

```
;INTERRUPT ROUTINES.   REFERENCE: SIMULTANEOUS INPUT
; AND OUTPUT FOR YOUR 8080.    BY W.D. MAURER
; IN THE MAY 1979 BYTE.

ABASE   EQU     00H
BBASE   EQU     10H
LOOP    EQU     7CH
LPDATA  EQU     LOOP
LPSTAT  EQU     LOOP+1
LPAV    EQU     LOOP+2  IN CLEARS LPAV. OUT SETS LPREQ.
LPULSE  EQU     LOOP+3  IN SETS LPAV. OUT CLEARS LPREQ.
LPABIT  EQU     01H
LPORE   EQU     10H
TTYST   EQU     ABASE
TTYBR   EQU     ABASE
TTY     EQU     ABASE+1
TTYCM   EQU     ABASE+2
TTYIR   EQU     ABASE+3
TIMER4  EQU     ABASE+8
BST     EQU     BBASE
BBR     EQU     BBASE
BTER    EQU     BBASE+1
BCM     EQU     BBASE+2
BIR     EQU     BBASE+3
TBE     EQU     80H
RDA     EQU     40H
IPG     EQU     20H
ORE     EQU     02H     OVERRUN ERROR FLAG BIT.
BAUD96  EQU     0C0H    1 STOP BIT 9600 BAUD
BAUD3   EQU     084H    1 STOP BIT  300 BAUD
RESET   EQU     9       RESET COMMAND

INIT$
        MVI     A,80H
        OUT     ABASE+4
        MVI     A,RESET GET RESET COMMD
        OUT     TTYCM
        OUT     BCM
        MVI     A,BAUD96        GET BAUD RATE
        OUT     TTYBR
        MVI     A,BAUD3 GET B BAUD
        OUT     BBR
        MVI     A,0FFH  ALL ONES TO AREG
        OUT     0FFH    ALL ZERO LEDS
        MVI     A,074H  ENABLE TBE,RDA,SENS-L,TIMER 4
        OUT     TTYIR    FOR PORT A
        MVI     A,020H  ENABLE TBE
        OUT     BIR      FOR PORT B
        IN      TTY
        IN      TTYIR   CLEAR IPG
        IN      BIR     . .
        IN      LPAV    CLEAR LOOP GRAB
        OUT     LPULSE  CLEAR LOOP REQUEST
        MVI     A,1     ALLOW RECEIVES.
```

```
        STA     RXPRI
        MVI     A,0
        STA     TXPRI
        CALL    SETPRI
        LXI     H,RECTAB
        LOOP    B,MAXREC+MAXREC+MAXREC+MAXREC
         MVI    M,0
         INX    H
        END     LOOP
        XRA     A       SET TXBUSY TO FALSE
        STA     TXBUSY
        MVI     A,0
        STA     STATE
        CALL    FLUSH
        EI
        RET

FLUSH
        PUSH    H
        LXI     H,BIQ   CLEAR INPUT QUEUE
        SHLD    FIQ
        SHLD    EIQ
        LXI     H,BOQ   CLEAR OUTPUT QUEUE
        SHLD    FOQ
        SHLD    EOQ
        POP     H
        XRA     A       NOT TYPING CHARACTERS, AND
        STA     TBESET
        STA     INPFULL INPUT NOT FULL
        STA     STOP    NOT STOPPED
        RET

CHIN$
        PUSH    H
        IFE     =,0
         LHLD   FIQ
         DO     NZ      WAIT FOR Q NOT EMPTY
          LDA   EIQ
          CMP   L
         END    DO
         MOV    A,M     GET CHAR
         PUSH   P
         XRA    A       CLEAR STOP
         STA    STOP
         CALL   SUCCI
         SHLD   FIQ
         POP    P
        END     THEN
         IFE    =,3
          LHLD  FIQ
          LDA   EIQ
          SUB   L
         END    THEN
         END    ELSE
        END     ELSE
        POP     H
        RET
```

```
CHOUT$
        PUSH    H
        LHLD    EOQ
        MOV     M,B
        CALL    SUCCO
        DO      NZ      WAIT FOR EMPTY SLOT IN Q
         LDA    FOQ
         CMP    L
        END     DO
        SHLD    EOQ
        DI
        LXI     H,TBESET
        MOV     A,M     TEST AND SET.
        MVI     M,1
        ORA     A
        IF      Z       IF ZERO NOW, Q IS EMPTY.
         CALL   RST5
        END     IF
        EI
        POP     H
        MOV     A,B
        RET

CCONT$
;VALUES OF STOP AND THEIR MEANING:
; 0     CONTINUE PRINTING NORMALLY
; 1     WAIT FOR STOP TO CHANGE
; 2     STOP LISTING.
; 3     STOP LISTING AND CLEAR BUFFERS.
;WAIT FOR HIM TO TYPE ANOTHER ^S OR ^C (WHICH CLEARS STOP)
        DO      NZ
         LDA    STOP
         CPI    1
        END     DO
        CPI     3
        JZ      FLUSH   IF HE TYPED ^C, ALSO CLEAR BUFFERS.
        IF      =,2
         XRA    A
         STA    STOP
        END     IF
        RET

RST4
        EI
        PUSH    P
        IN      TTY     A:=INP(TTY) AND 7FH
        ANI     7FH
        IFE     =,'S'-40H       IF CHAR=^S THEN
         LDA    STOP    FLIP STOP FROM 0 TO 1 OR 1 TO 0
         XRI    1
         STA    STOP
        END     THEN
         PUSH   P
         IFE    =,'C'-40H       IF CONTROL C,
          MVI   A,3     SET STOP TO CLEAR BUFFERS AND STOP
          STA   STOP
```

```
         END     THEN
          LDA    STOP     IF WAITING FOR STOP,
          IF     =,1       SET STOP TO 2
           MVI   A,2
           STA   STOP
          END    IF
         END     ELSE
         IN      TTYST
         ANI     ORE      OVERRUN ERROR?
         IF      NZ
          MVI    A,'?'    YES - STORE A QUESTION MARK.
          CALL   RDASTOR
         END     IF
         POP     P
         CALL    RDASTOR
        END      ELSE-CTRL S
        POP      P
        RET

RDASTOR
        PUSH     H
        LHLD     EIQ      STORE CHAR
        MOV      M,A
        CALL     SUCCI
        LDA      FIQ      IF FIQ<>L THEN EIQ=HL
        CMP      L
        IFE      NZ       IF NOT FULL,
         SHLD    EIQ      UPDATE POINTER
        END      THEN     ELSE IF FULL,
         LXI     H,INPFULL        SCHEDULE A BELL TO BE TYPED.
         INR     M
         LDA     TBESET   IS A CHARACTER BEING PRINTED
         ORA     A        OR SHOULD WE START ONE NOW?
         IF      Z        IF TBESET=FALSE, PRINT IT NOW.
          DCR    M        AND DESCHEDULE THE BELL.
          MVI    A,'G'-40H
          OUT    TTY
         END     IF
        END      ELSE
        POP      H
        RET

RST5
        PUSH     P
        PUSH     H
        LXI      H,INPFULL
        MOV      A,M
        ORA      A
        IFE      NZ       IF ANY BELLS TO TYPE,
         DCR     M        DECR COUNT
         MVI     A,'G'-40H
         OUT     TTY
        END      THEN
         LHLD    FOQ      TEST AMOUNT OF DATA IN QUEUE
         LDA     EOQ
         CMP     L
         IFE     NZ       IF NOT EMPTY,
```

```
          MOV   A,M      OUTPUT CHARACTER
          OUT   TTY      INTERRUPTS MUST BE TURNED OFF HERE,
          CALL  SUCCO     OTHERWISE, THE SAME CHAR MAY BE
          SHLD  FOQ       TYPED TWICE.
         END    THEN
          XRA   A        IF EMPTY, CLEAR TBESET
          STA   TBESET
         END    ELSE
        END     ELSE
        POP     H
        POP     P
        EI
        RET

SUCCI   INX     H        INC HL
        MOV     A,L
        CPI     BIQ+LIQ,>
        RNZ              RETURN IF L<>END OF INPUT Q
        LXI     H,BIQ
        RET

SUCCO   INX     H
        MOV     A,L
        CPI     BOQ+LOQ,>
        RNZ
        LXI     H,BOQ
        RET

        END
```

```
;INTERRUPT FOR LOOP AVAILABLE
RST1
        PUSH    P
        PUSH    B
        PUSH    D
        PUSH    H
        OUT     LPULSE  CLEAR INTERRUPT REQUEST.
        CALL    SEND
        LDA     THISADR ARE WE SENDING IT TO
        CMP     M        OURSELVES?
        IFE     Z
         INX    H
         MOV    A,M     GET THE COMMAND BYTE
         PUSH   H
         CALL   RECTADR GET RECTAB ADDRESS
         IF     NC
          CALL  GET2    LOAD DE WITH THE ADDR
          XCHG            OF RECEIVE PROCESSING ROUTINE.
          SHLD  RXPROC
          XCHG
          CALL  GET2
          POP   H
          MOV   A,D     WE CAN RECEIVE THIS MESSAGE ONLY IF
          ORA   E        WE CALLED ADDR FIRST.
          IF    NZ
           INX  H
           MOV  C,M     GET THE SIZE
           INX  H
           MOV  B,M
           DCX  H
           DCX  H
           DCX  H
           INX  B       ADD FOUR TO IT.
           INX  B       . .
           INX  B       . .
           INX  B       . .
           DO   Z
            MOV A,M
            INX H
            STAX        D
            INX D
            DCX B
            MOV A,B
            ORA C
           END  DO
           MVI  A,0     RESET TXPRI
           STA  TXPRI
           CALL SETPRIOUT
           XRA  A       CLEAR TXBUSY
           STA  TXBUSY
           IN   LPAV    RELEASE LOOP
           EI
           MVI  B,1     TELL HIM HIS MESSAGE IS SENT,
           LHLD DONEADR
```

```
          CALL INDIRECT
          CALL RECV     AND PROCESS IT.
         END   IF
        END    IF
       END     THEN
        SHLD   TXPTR
        MOV    A,M
        STA    TXDEST
        INX    H
        INX    H        SET TXCNT
        CALL   GET2
        XCHG
        INX    H        ADD FOUR TO IT.
        INX    H        . .
        INX    H        . .
        INX    H        . .
        SHLD   TXCNT
        MVI    A,BOMCHR         SEND BOM
        OUT    LPDATA
        XRA    A        MAKE SURE NOT ESC OR BOMCHR
        STA    LASTCHR
        MVI    A,4      ALLOW LPTBE'S.
        STA    TXPRI
       END     ELSE
       CALL    SETPRI
       POP     H
       POP     D
       POP     B
       POP     P
       EI
       RET

;INTERRUPT FOR LOOP TRANSMITTER BUFFER EMPTY
RST3
       PUSH    P
       PUSH    B
       PUSH    D
       PUSH    H
       CALL    RST3DO  TYPE NEXT CHAR, IF ANY LEFT.
       CALL    SETPRI
       POP     H
       POP     D
       POP     B
       POP     P
       EI
       RET

RST3DO
       LDA     LASTCHR
       IFE     =,ESC
        MVI    A,FESC
       END     THEN
        IF     =,BOMCHR
         MVI   A,FBOMCHR
        END    IF
       END     ELSE
       IFE     Z        IF ONE OF THE ABOVE, TRANSMIT
```

```
 OUT    LPDATA  IT NOW.
 XRA    A       MAKE SURE LASTCHR IS NOT
 STA    LASTCHR  BOMCHR OR ESC.
END     THEN
 LHLD   TXCNT
 MOV    A,H
 ORA    L
 IFE    Z
  DO    NZ      WAIT FOR LAST CHAR TO BE TRANSMITTED
   IN   LPSTAT
   ANI  02H
  END   DO
  MVI   A,0
  STA   TXPRI
  MVI   A,TIMEOUT
  OUT   TIMER4
 END    THEN
  DCX   H       DECREMENT TXCNT
  SHLD  TXCNT
  LHLD  TXPTR
  MOV   A,M
  INX   H       GET NEXT CHAR
  STA   LASTCHR
  SHLD  TXPTR
  IF    =,BOMCHR        BOMCHR GETS CONVERTED
   MVI  A,ESC   TO ESC,FBOMCHR, AND ESC BECOMES
  END   IF       ESC,FESC.
  OUT   LPDATA  SEND IT
 END    ELSE
END     ELSE
RET

RST6
        PUSH    P
        PUSH    B
        PUSH    D
        PUSH    H
        MVI     B,1     ASSUME IT WAS SENT.
        IN      LPAV    RELEASE LOOP
        LDA     TXDEST  EXPECT ACKNOWLEDGE ONLY
        IF      <>,0     IF NOT BROADCAST.
         IN     LPSTAT  IF A CHARACTER WAS INPUT,
         ANI    40H
         IF     NZ
          IN    LPDATA   INPUT IT.
         END    IF
         IF     <>,' '  IF IT'S NOT A BLANK,
          MVI   B,0      IT WASN'T RECEIVED.
         END    IF
        END     IF
        EI
        XRA     A
        STA     TXBUSY
        LHLD    DONEADR
        CALL    INDIRECT
        POP     H
        POP     D
```

```
        POP     B
        POP     P
        EI
        RET

GRAB
        DO      Z
         LDA    TXBUSY
         ORA    A
        END     DO
        INR     A
        STA     TXBUSY
        SHLD    DONEADR
        XCHG
        SHLD    SENDADR
        MVI     A,2     ALLOW LPA'S
        STA     TXPRI
        CALL    SETPRI
        OUT     LPAV    HOLD TOKEN NEXT TIME IT GETS HERE.
        RET

SEND
        LHLD    SENDADR
        PCHL

ADDR
        PUSH    H
        CALL    RECTADR
        RC
        CALL    PUT2
        POP     D
        CALL    PUT2
        LDA     THISADR
        RET

;FORM INDEX INTO RECTAB. ENTER WITH A=COMMAND BYTE,
; EXIT WITH HL=>PROPER ENTRY AND CY=0, OR CY=1 IF COMMAND
; OUT OF RANGE.
RECTADR
        CPI     MAXREC
        CMC
        RC
        ADD     A
        ADD     A
        LXI     H,RECTAB
        CALL    ADD2
        RET

;INTERRUPT FOR BEGINNING OF MESSAGE
RST0
        PUSH    P
        PUSH    B
        PUSH    D
        PUSH    H
        IN      LPDATA  CLEAR INTERRUPT
        LDA     STATE   ARE WE ALREADY PROCESSING MESSAGE?
        IF      <>,-1
```

```
         CALL    NEXTSTATE       NO, GO TO STATE ONE.
         MVI     A,3      ALLOW LPRDA'S
         STA     RXPRI
        END      IF
        CALL     SETPRI
        POP      H
        POP      D
        POP      B
        POP      P
        EI
        RET

;INTERRUPT FOR SENSE A AND LOOP RECEIVER BUFFER FULL
RST2
        PUSH     P
        PUSH     B
        PUSH     D
        PUSH     H
        IN       BST      GET B-STATUS
        ANI      IPG
        IFE      NZ
         CALL    RST2TU
        END      THEN
         CALL    RST2LP
        END      ELSE
        POP      H
        POP      D
        POP      B
        POP      P
        EI
        RET

RST2TU
        IN       BIR      GET INTERRUPT ADDRESS
        RET

RST2LP
        IN       LPDATA   CLEAR INTERRUPT REQUEST.
        MOV      B,A      SAVE INPUT BYTE.
        LDA      STATE    IF WE'RE IN STATE ZERO, WE
        IF       <>,0      SHOULDN'T RECEIVE THIS.
         CALL    LPCHIN   PROCESS THIS CHAR.
         IFE     NZ       IF LPCHIN RETURNS Z=0,
          CALL   ENDRDA   STOP RECEIVING MESSAGE
         END     THEN
          LDA    STATE    IF MESSAGE COMPLETE,
          IF     =,6
           LDA   RXDEST   WAS THIS NOT A BROADCAST?
           IF    <>,0     IF NOT BROADCAST,
            MVI  A,' '     SEND A CHAR.
            OUT  LPDATA
           END   IF
           MVI   A,-1     REMEMBER THAT WE'RE
           STA   STATE    PROCESSING A MESSAGE
           MVI   A,1
           STA   RXPRI
           CALL  SETPRI
```

```
            EI
            CALL RECV    AND PROCESS MESSAGE.
            CALL ENDRDA
           END   IF
          END    ELSE
         END     IF
         CALL    SETPRI
         RET

DCOM$$
         PUSH    P
         MVI     A,0
         CALL    SETPRIOUT
         POP     P
         CALL    DCOM$
         CALL    STRTMSG
         RET

RECV
         LHLD    RXPROC
INDIRECT
         PCHL

ENDRDA
         MVI     A,1
         STA     RXPRI
         MVI     A,0
         STA     STATE
         RET

;FIGURE OUT WHAT TO DO WITH THE CHARACTER WE JUST RECEIVED.
;THE CHAR IS IN THE B-REG.
LPCHIN
         MOV     A,B
         IF      =,ESC
          LDA    STATE
          ORI    80H
          STA    STATE
          XRA    A       RETURN Z=1
          RET
         END     IF
         LDA     STATE   TEST STATES.
         ANI     80H
         IF      NZ
          LDA    STATE
          ANI    7FH
          STA    STATE
          MOV    A,B
          IFE    =,FBOMCHR
           MVI   B,BOMCHR
          END    THEN
           IF    =,FESC
            MVI  B,ESC
           END   IF
          END    ELSE
         END     IF
         LDA     STATE   BRANCH TO STATES 1,2,3,4,5
```

```
        DCR     A
        JZ      LPCHIN1
        DCR     A
        JZ      LPCHIN2
        DCR     A
        JZ      LPCHIN3
        DCR     A
        JZ      LPCHIN4
        DCR     A
        JZ      LPCHIN5
        JMP     $       TRAP BAD STATES.

;ACCEPT DESTINATION.
LPCHIN1
        CALL    RXCHECK MAKE SURE CHARACTER IS OK.
        RNZ
        MOV     A,B     ARE WE BEING BROADCAST TO?
        IF      <>,0    NO -
         LDA    THISADR  MAKE SURE IT'S FOR US,
         IF     <>,0     BUT ONLY IF WE'RE NOT EAVESDROPPING
          CMP   B
         END    IF
        END     IF
        RNZ             IF NOT EQUAL, IT'S NOT FOR US.
        STA     RXDEST  =0 IF BROADCASTING OR EAVESDROPPING.
        CALL    NEXTSTATE       ELSE GO TO STATE 2
        XRA     A       RETURN OK.
        RET
;ACCEPT COMMAND BYTE
LPCHIN2
        MOV     A,B
        CALL    RECTADR
        IF      C
         ORI    1       MAKE SURE NOT ZERO (Z=0)
         RET
        END     IF
        CALL    GET2
        MOV     A,D     IF ENTRY IN RECTAB IS ZERO,
        ORA     E        WE CAN'T RECEIVE THIS MESSAGE.
        IF      Z
         INR    A       A IS NOW NOT ZERO, HENCE Z=0.
         RET
        END     IF
        XCHG
        SHLD    RXPROC
        XCHG
        CALL    GET2
        XCHG
        SHLD    RXPTR
        CALL    NEXTSTATE
        MOV     C,B
        LDA     RXDEST
        MOV     B,A
        CALL    RXSTOR
        RNZ
        MOV     B,C
        CALL    RXSTOR
```

```
        RET
;ACCEPT SIZE LOW
LPCHIN3
        MOV     A,B
        STA     RXCNT
        CALL    NEXTSTATE
        CALL    RXSTOR
        RET
;ACCEPT SIZE HIGH
LPCHIN4
        MOV     A,B
        STA     RXCNT+1
        CALL    NEXTSTATE
        LHLD    RXCNT
        JMP     LPCHIN5.1       TEST FOR SIZE=0.
;ACCEPT DATA BYTES.
;GOES TO NEXT STATE WHEN ALL DATA HAS BEEN RECEIVED.
LPCHIN5
        LHLD    RXCNT
        DCX     H
        SHLD    RXCNT
LPCHIN5.1
        MOV     A,H
        ORA     L
        CZ      NEXTSTATE
;       CALL    RXSTOR
;       RET

RXSTOR
        CALL    RXCHECK
        RNZ             FATAL TO MESSAGE IF NOT ZERO.
        PUSH    H       STORE INPUT CHARACTER
        LHLD    RXPTR
        MOV     M,B
        INX     H
        SHLD    RXPTR
        POP     H
;BE SURE TO RETURN Z=1 SO THAT WE CONTINUE RECEIVING
        RET

;RETURN NZ IF CHARACTER NOT OK.
RXCHECK
        IN      LPSTAT  CHECK ERROR FLAGS.
        ANI     LPORE
        RET

NEXTSTATE
        LDA     STATE
        INR     A
        STA     STATE
        RET

SETPRI
        PUSH    H
        LXI     H,RXPRI
        LDA     TXPRI
        CMP     M
```

```
        IF      <       IF TXPRI < RXPRI,
         MOV    A,M      USE RXPRI
        END     IF
        POP     H
        STA     LASTPRI
SETPRIOUT
        OUT     LPSTAT
        CMA
        OUT     0FFH
        RET

STOPMSG
        PUSH    P
        WHIL    NZ      WAIT FOR MESSAGE SILENCE
         DI
         LDA    TXPRI
         IF     =,0
          LDA   STATE
          ORA   A
         END    IF
        END     COND
         EI
        END     WHILE
        MVI     A,0     DON'T ALLOW MESSAGES
        CALL    SETPRIOUT
        EI
        POP     P
        RET

STRTMSG
        PUSH    P       SAVE FLAGS
        LDA     LASTPRI
        CALL    SETPRIOUT
        POP     P       RESTORE FLAGS
        RET

THISADR EQU     3FH

BOMCHR  EQU     'B'-40H
FBOMCHR EQU     'B'
ESC     EQU     '['-40H
FESC    EQU     '['

TIMEOUT EQU     36      APPROX EQU TO 2 CHAR TIMES. (2.3MS)
LAST    EQU     $

        ORG     2A00H-24-12
FREE    EQU     $-LAST

        JMP     STOPMSG
        JMP     STRTMSG
        JMP     GRAB
        JMP     ADDR
        JMP     RST0
        JMP     RST1
        JMP     RST2
        JMP     RST3
```

```
        JMP     RST4
        JMP     RST5
        JMP     RST6
        JMP     $

MAXREC  EQU     6       MAXIMUM NUMBER OF COMMAND BYTES.

RECTAB
        DS      MAXREC+MAXREC+MAXREC+MAXREC

LIQ     EQU     20
LOQ     EQU     80

FIQ     DS      2
EIQ     DS      2
FOQ     DS      2
EOQ     DS      2
TBESET  DS      1
STOP    DS      1

RXPTR   DS      2
RXCNT   DS      2
RXPRI   DS      1
STATE   DS      1
RXDEST  DS      1
RXPROC  DS      2

TXBUSY  DS      1
TXPTR   DS      2
TXCNT   DS      2
TXPRI   DS      1
TXDEST  DS      1
LASTCHR DS      1
SENDADR DS      2
DONEADR DS      2

LASTPRI DS      1
DCOM$$A DS      1

INPFULL DS      1
FRSTFRE DS      2
FNAME
        DS      8
UNIT    DS      1
LLFLAG  DS      1
LASTDIR DS      2       LAST DIRECTORY SECTOR READ.

MAXSCT  EQU     350

INSIZE  EQU     40
INBUF
        DS      INSIZE
INPTR   DS      2
STADR   DS      2

CR      EQU     0DH
LF      EQU     0AH
```

```
BIQ     DS      LIQ
BOQ     DS      LOQ
        DS      100
STACK
SAVSTK  DS      2
DIRBUF  DS      256
FREE2   EQU     $
        END
```

```
;       THE PURPOSE OF THIS MODULE IS TO PROVIDE SIMULATED
;DISKS ON THE DISTRIBUTED LOOP SYSTEM.   IN THE NORTH STAR
;DOS, DISKS ARE REFFERED TO BY UNIT.   IN THE LOOP DOS,
;A DISK UNIT MAY BE REASSIGNED TO A PHYSICAL DRIVE ON
;A DIFFERENT COMPUTER.
;       WHEN A CALL IS MADE TO DCOM, IT GOES, NOT TO THE DCOM
;ROUTINE, BUT TO THIS ROUTINE.   IF THE LOGICAL UNIT IS PHYSICALLY
;RESIDENT IN THIS SYSTEM, IT IS EXECUTED LOCALLY, ELSE IT IS SENT
;TO A DIFFERENT COMPUTER.   EACH SECTOR IN THE REQUEST IS
;REQUESTED SEPERATELY.   THE INFORMATION NECESSARY TO READ A SECTOR
;IS: THE DISK ADDRESS, AND TO WRITE A SECTOR: THE DISK ADDRESS,
;AND A SECTOR'S WORTH OF DATA.

;INITIALIZE FOR READING.
        CALL    ALLOWRECV
        STA     THISADR
        LHLD    2023H   GET THE PHYSICAL DCOM ADDRESS
        LXI     D,DCOML IF IT'S ALREADY SET TO OUR ROUTINE,
        CALL    DCMP
        RZ              DON'T SET IT AGAIN.
        SHLD    DCOMA
        LXI     H,DCOML AND SUBSTITUTE THE LOGICAL DCOM ADDR.
        SHLD    2023H

ALLOWRECV
        LXI     D,READDATA
        LXI     H,RDBUFF
        MVI     A,RDCMD
        CALL    ADDR
        LXI     D,READRET
        LXI     H,RRBUFF
        MVI     A,RRCMD
        CALL    ADDR
        LXI     D,ERROR
        LXI     H,ERBUFF
        MVI     A,ERCMD
        CALL    ADDR
        LXI     D,WRITEDATA
        LXI     H,WDBUFF
        MVI     A,WDCMD
        CALL    ADDR
        LXI     D,WRITERET
        LXI     H,WRBUFF
        MVI     A,WRCMD
        CALL    ADDR
        RET

DCOML
        PUSH    P       SAVE THE SECTOR COUNT
        PUSH    H       AND THE DISK ADDRESS.
        MOV     A,C     FIND OUT THE PHYSICAL ASSIGNMENT
        LXI     H,PHYS
        CALL    ADD2
        MOV     A,M
```

```
        POP     H
        ORA     A       IF IT'S ZERO, IT'S ALREADY PHYSICAL.
        JZ      ALLPHYS
;THE NUMBER IN A IS THE NUMBER OF THE COMPUTER WHICH OWNS THE DISK.
        STA     DESTIN
        MOV     A,B     GET COMMAND (ONLY READ OR WRITE)
        ORA     A       WRITE IS ZERO.
        JZ      WRITE
        POP     P
        WHIL    NZ      READ LOOP
         ORA    A
        END     COND
;        DEBUG  /READ
         PUSH   P       SAVE SECTOR COUNT
         SHLD   RDBUFF+5
         PUSH   H
         LXI    H,RDLEN
         SHLD   RDBUFF+2
         PUSH   D
         XCHG           GET MEMORY ADDRESS
         SHLD   MEMADR
         MVI    B,RDCMD
         LXI    H,RDBUFF
         CALL   TRANSFER
         POP    D       GET MEM ADDRESS
         INR    D       INC BY 100H.
         POP    H
         INX    H
         POP    P
         DCR    A
        END     WHILE
        RET

WRITE
        POP     P
        WHIL    NZ      WRITE LOOP.
         ORA    A
        END     COND
;        DEBUG  /WRITE
         PUSH   P       SAVE SECTOR COUNT
         SHLD   WDBUFF+5
         PUSH   H
         PUSH   D
         LXI    H,WDLEN SET THE SIZE
         SHLD   WDBUFF+2
         LXI    H,WDBUFF+7
         CALL   MOVESCT MOVE THE SECTOR TO THE BUFFER.
         MVI    B,WDCMD
         LXI    H,WDBUFF
         CALL   TRANSFER
         POP    D       GET MEM ADDRESS
         INR    D       INC BY 100H.
         POP    H
         INX    H
         POP    P
         DCR    A
        END     WHILE
```

```
        RET

;PHYSICAL DISK.
ALLPHYS
        POP     P
        JMP     DCOM

;SEND A MESSAGE AND WAIT FOR IT'S RESPONSE TO COME BACK.
;ENTER WITH HL=> THE BUFFER TO SEND.
TRANSFER
        LDA     DESTIN
        MOV     M,A
        MVI     A,0
        STA     FLAG
        PUSH    H
        INX     H
        INX     H
        INX     H
        INX     H
        LDA     THISADR
        MOV     M,A
        POP     H
        CALL    GRABBER
        DO      NZ
         LDA    FLAG
         ORA    A
        END     DO
        RP              RETURN IF OK.
        LXI     H,ERBUFF+4
        MOV     B,M
        INX     H
        MOV     C,M
        INX     H
        MOV     D,M
        CALL    HDERPR
        JMP     HDER1

;SEND A MESSAGE.
GRABBER
        SHLD    BUFADR
        INX     H
        MOV     M,B
        MOV     A,B
;       DEBUG   /GRAB
        LXI     D,SEND
        LXI     H,DONE
        CALL    GRAB
        RET

SEND
        LHLD    BUFADR
        RET

DONE
        MOV     A,B
        IF      <>,1
         LHLD   BUFADR
```

```
          INX     H
          MOV     B,M
          DCX     H
          CALL    GRABBER
         END      IF
         RET

READDATA
;        DEBUG   /READDATA
;SWITCH TO OUR STACK
         LXI     H,0
         DAD     S
         SHLD    STACK
         LXI     S,STACK
         LDA     RDBUFF+4
         STA     RDBUFF
         MVI     A,1
         LXI     B,11   DO IT TO UNIT ONE.
         LHLD    RDBUFF+5        DISK ADDR.
         LXI     D,RDBUFF+4
         CALL    DCOM
         LDA     RDBUFF
         JC      HDERL
         LXI     H,RRLEN
         SHLD    RDBUFF+2
         LXI     H,RDBUFF
         MVI     B,RRCMD
         JMP     READDATA1
WRITEDATA
;        DEBUG   /WRITEDATA
;SWITCH TO OUR STACK
         LXI     H,0
         DAD     S
         SHLD    STACK
         LXI     S,STACK
         MVI     A,1
         LXI     B,01   DO IT TO UNIT ONE.
         LHLD    WDBUFF+5        DISK ADDR.
         LXI     D,WDBUFF+7
         CALL    DCOM
         LDA     WDBUFF+4
         JC      HDERL
         STA     WDBUFF
         LXI     H,WRLEN
         SHLD    WDBUFF+2
         MVI     B,WRCMD
         LXI     H,WDBUFF
         JMP     READDATA1
HDERL
;        DEBUG   /HDERL
         STA     ERBUFF
         LXI     H,ERBUFF+4
         MOV     M,B     SAVE TRACK
         INX     H
         MOV     M,C     SAVE UNIT
         INX     H
         MOV     M,D     SAVE SECTOR
```

```
        LXI     H,ERLEN
        SHLD    ERBUFF+2
        MVI     B,ERCMD
        LXI     H,ERBUFF
;RESTORE HIS STACK.
READDATA1
        XCHG
        LHLD    STACK
        SPHL
        XCHG
        CALL    GRABBER
        RET

ERROR
;       DEBUG   /ERROR
        MVI     A,-1    ERROR!
        STA     FLAG
        RET

READRET
;       DEBUG   /READRET
        MVI     A,1     TELL THAT DATA HAS BEEN RETURNED.
        STA     FLAG
        LHLD    MEMADR  DISPOSE OF THE BUFFER BY MOVING
        LXI     D,RRBUFF+4      IT TO IT'S DESTINATION ADDR.
        CALL    MOVESCT
        RET

WRITERET
;       DEBUG   /WRITERET
        MVI     A,1
        STA     FLAG
        RET

;SIMULATE A NORMAL CALL TO DCOM.
DCOM
        PUSH    H
        LHLD    DCOMA   GET PHYSICAL DCOM ADDRESS
        XTHL
        RET             JUMP TO IT WITHOUT CHANGING HL.

;MOVESCT MOVES A SECTOR FROM DE TO HL, INCREMENTING BOTH BY 256.
MOVESCT
        PUSH    B
        LOOP    B,0
         LDAX   D
         INX    D
         MOV    M,A
         INX    H
        END     LOOP
        POP     B
        RET

;HL:=HL+A
ADD2
        ADD     L
        MOV     L,A
```

```
        RNC
        INR     H
        RET

;CY=1 IF DE<HL. Z=1 IF DE=HL.
DCMP
        MOV     A,D
        CMP     H
        RNZ
        MOV     A,E
        CMP     L
        RET

GRAB    EQU     29E2H
ADDR    EQU     29E5H

HDER1   EQU     202CH
HDERPR  EQU     202FH

DCOMA   DS      2       ADDRESS OF PHYSICAL DCOM.
MEMADR  DS      2
BUFADR  DS      2
DESTIN  DS      1       OWNER OF DISK DRIVE.
FLAG    DS      1       FLAG, TRUE WHEN ACK MESSAGE RECEIVED.
THISADR DS      1       ADDRESS OF THIS COMPUTER.
DRIVES  EQU     2       NUMBER OF LOGICAL DRIVES
PHYS    DB      0
        DB      0
        DB      1
        DB      2
        DB      3
        DB      4

        DS      60
STACK
        DS      2       USED TO HOLD HIS STACK POINTER.

;FIVE MESSAGES CAN BE SENT AND RECEIVED:
RDCMD   EQU     0
; READ DATA: DISK ADDRESS.
RRCMD   EQU     1
; READ RETURN: DISK DATA.
ERCMD   EQU     2
; ERROR RETURN: BAD TRACK,BAD UNIT,BAD SECTOR.
WDCMD   EQU     3
; WRITE DATA: DISK ADDRESS, SECTOR DATA.
WRCMD   EQU     4
; WRITE RETURN: .
RDLEN   EQU     3
RRLEN   EQU     256
ERLEN   EQU     3
WDLEN   EQU     1+2+256
WRLEN   EQU     0

RDBUFF
        DS      RRLEN+4
RRBUFF
```

```
        DS      RRLEN+4
ERBUFF
        DS      ERLEN+4
WDBUFF
        DS      WDLEN+4
WRBUFF
        DS      WRLEN+4
ENDRAM

        END
```