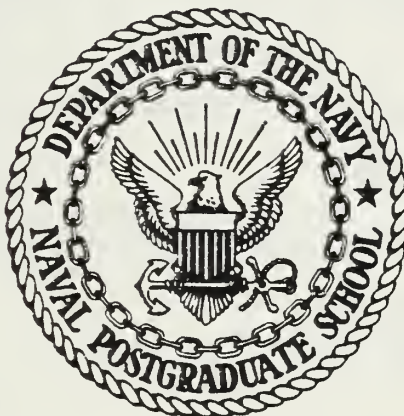


Quail Point Library, NPS
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

MICROCOMPUTER NETWORKING:
A CP/M-BASED APPLICATION

by

Thomas M. Carnahan

and

Michael K. Waters

September 1983

Thesis Advisor:

G. E. Latta

Approved for public release; distribution unlimited

T210151

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
 BEFORE COMPLETING FORM

1. REPORT NUMBER		2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Microcomputer Networking: a CP/M-Based Application		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis September, 1983	
7. AUTHOR(s) Thomas M. Carnahan Michael K. Waters		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		8. CONTRACT OR GRANT NUMBER(s)	
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE September, 1983	
		13. NUMBER OF PAGES 145	
		15. SECURITY CLASS. (of this report) UNCLASSIFIED	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) microcomputer, networks, CP/M, Local Area Network, Computer Interfacing, Computers Software, Software Engineering, Apple Northstar			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This thesis examines a topology and protocol for interconnection of microcomputers to allow them to communicate with each other. An application is offered based on the CP/M operating system and other industry standards. A low-cost hardwired implementation for connecting microcomputers is provided with software to allow file transfers. The implementation was developed by redesigning and assimilating existing software utilizing modern software engineering techniques.			

Approved for public release; distribution unlimited.

Microcomputer Networking: a CP/M-Based Application

by

Thomas M. Carnahan
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1971

and

Michael K. Waters
Lieutenant, United States Navy
B.S., Oregon State University, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
September 1983

Thesis
C261365
c.1

ABSTRACT

This thesis examines a topology and protocol for interconnection of microcomputers to allow them to communicate with each other. An application is offered based on the CP/M operating system and other industry standards. A low-cost hardwired implementation for connecting microcomputers is provided with software to allow file transfers. The implementation was developed by redesigning and assimilating existing software utilizing modern software engineering techniques.

TABLE OF CONTENTS

I.	INTRODUCTION -----	8
II.	NETWORKING -----	10
	A. NETWORK STRUCTURES -----	12
	B. NETWORK PROTOCOL -----	13
	1. The ISO/ANSI Open System Interconnection (OSI) Model -----	14
	a. Physical Layer -----	14
	b. Data Link Layer -----	15
	2. Types of Data Link Protocols -----	16
	a. Unrestricted Simplex -----	16
	b. Stop and Wait -----	17
	c. Positive Acknowledgment with Retrans- mission -----	17
	d. Stop and Wait Sliding Window -----	20
	e. Sliding Window with Pipelining -----	21
	f. Piggybacking -----	21
	3. Summary -----	22
III.	CP/M -----	24
	A. HISTORY OF CP/M -----	24
	B. ORGANIZATION OF CP/M -----	26
	C. SYMBOLOGY -----	28
	D. COMMANDS -----	31

1.	Built-in Commands	-----	31
2.	Transient Commands	-----	34
IV.	RS-232-C INTERFACE	-----	38
V.	PRINCIPLES OF SOFTWARE ENGINEERING	-----	47
A.	SOFTWARE DESIGN OBJECTIVES	-----	48
1.	Modifiability	-----	48
2.	Understandability	-----	49
3.	Reliability	-----	49
4.	Efficiency	-----	50
B.	SOFTWARE DESIGN PRINCIPLES	-----	50
1.	Modularity	-----	50
2.	Abstraction	-----	52
3.	Localization	-----	53
4.	Hiding	-----	53
C.	TOOLS FOR SOFTWARE DESIGN	-----	54
1.	Choice of Programming Language	-----	54
2.	Programming Style	-----	55
VI.	DESCRIPTION OF PROJECT SOFTWARE	-----	57
A.	DISCUSSION OF REQUIREMENTS AND ASSUMPTIONS	----	57
B.	DESIGN DECISIONS	-----	59
C.	SOFTWARE IMPLEMENTATION PLAN	-----	62
D.	THE MAIN PROGRAM (CROSS.ASM)	-----	64
1.	Actions by the Sending Computer During Transfer	-----	67
2.	Actions by the Receiving Computer During Transfer	-----	69

E.	DESCRIPTION OF THE MACRO LIBRARIES -----	70
1.	Macro Assembly -----	70
2.	Advantages of a Macro -----	71
3.	Criteria for Dividing Macros Between Libraries -----	72
VII.	CONCLUSIONS AND RECOMMENDATIONS -----	73
APPENDIX A:	UPLOAD PROGRAM -----	77
APPENDIX B:	DOWNLOAD PROGRAM -----	81
APPENDIX C:	FLOWCHARTS -----	85
APPENDIX D:	MAIN PROGRAM LISTING -----	91
APPENDIX E:	MAIN MACRO LIBRARY -----	109
APPENDIX F:	APPLE I/O MACRO LIBRARY -----	129
APPENDIX G:	NORTHSTAR I/O MACRO LIBRARY -----	135
LIST OF REFERENCES	-----	142
INITIAL DISTRIBUTION LIST	-----	144

LIST OF FIGURES

4.1	DB-25 Connector Pinout for RS-232-C -----	43
4.2	Project Network Protocol -----	45
C.1	CROSS.ASM Main Program -----	85
C.2	CROSS.ASM Main Program (continued) -----	86
C.3	CROSS.ASM SEND\$ROUTINE Program -----	87
C.4	SEND\$ROUTINE (continued) -----	88
C.5	CROSS.ASM RECEIVE\$ROUTINE Program -----	89
C.6	RECEIVE\$ROUTINE (continued) -----	90

I. INTRODUCTION

The trend of microcomputer capabilities has increased dramatically in the last few years, mainly due to improved technology and decreasing hardware costs, increasing the number of applications available to users who would otherwise have to utilize mainframes or simply do without computer power. A primary need of many small organizations, including schools, businesses, and military units is to be able to have more than one microcomputer share data and files. Since not all microcomputers use the same formatting of disks, the same programs, or the same operating systems, it is impossible to take the programs and data from one microcomputer disk and run them on another microcomputer made by a different manufacturer, otherwise known as transportability.

In many applications, a user may wish to communicate with distant users or share a peripheral connected to a different microcomputer. This must be done without disconnecting the original user from the peripheral and swapping cables every time someone needs to use it, as might happen when a printer is needed, for example. Unfortunately, there is no industry standard by which all diskettes are formatted, however a method for transferring information within standards that do

exist would solve many of these problems. This introduces the concept of a network to share files, data, and information. Many networking systems are commercially available [Ref. 1] but would be prohibitively expensive for the small microcomputer user. This thesis examines some of the industry standards accepted through common practice, discusses a few of the fundamentals of microcomputer networking, and provides a low-cost means of data transfer within those standards. The primary goal is to begin to study various means for allowing multiple manufacturer microcomputers to transfer files at low cost.

Modern principles of software engineering have been utilized to redesign and enhance some existing routines which are applicable to microcomputer intercommunication. Commercially available products were used where appropriate and inexpensive. Further enhancements are also offered such as low cost telephone communications.

A working model of the design was constructed in the Mathematics Department Microcomputer Laboratory at the Naval Postgraduate School as a foundation for an ongoing project to implement a local area network.

II. NETWORKING

The idea of a single large computer serving all of an organization's needs is being replaced by a situation where a large number of smaller computers share resources to meet all users' needs. These systems of separately interconnected computers are called networks. They don't necessarily have to be connected with wire, as telecommunications, microwaves, and satellites are frequently used to allow computers to exchange information. Many authors restrict the terminology of networks to systems where computers operate autonomously, independent of each other with no master/slave relationships [Ref. 2: p. 2]. Since many simpler computer connections evolve into larger systems of autonomous computers, no such distinction will be made here and the term networks will be applied to any system of interconnected computers.

There are a number of reasons that networking systems have come into existence. The first is that many organizations have a number of independent computers already and later have a desire to share information between them to avoid reinventing the wheel. Frequently each computer carries information peculiar to a small area of the organization and management wants to assimilate the data for the whole organization.

The second reason is that reliability may be increased through networking, particularly if the cost of lost data processing is high. In this situation, it may be catastrophic to have one of your computers fail. Rather than buy a spare computer which sits unused waiting for the primary computer to fail, a networking scheme allows the other computers in the system to compensate for one of the computer's failure, usually with no interruption in user services.

The third reason for networking is for purely economic considerations due to technology. Since about 1970, it has been more economical to allow dispersed field offices process data on their own inexpensive yet powerful microcomputers rather than collecting the data to be sent to some national processing center. Summaries may then be sent occasionally to the central management, reducing the communications costs of transmitting volumous raw data. Large mainframe computers are about ten times faster than microcomputers but cost about a thousand times more.

Finally, convenience of networking allows more than one person to have access to computer resources, even to the point of both of them working on the same files which can be later assembled to make one report. An extension of this process implies electronic mail and improved human communications using the network as a tool.

A. NETWORK STRUCTURES

Networks are basically composed of hosts, switching elements, and transmission lines. They are usually connected together using one of two general methods. The first is point-to-point, or store-and-forward, where any pair of hosts or nodes that wish to communicate with each other are connected together. If they don't share a communication line, they must communicate indirectly by sending the whole message to a third node which receives the message and then sends it on to the intended receiver. Topologies, or physical connection schemes, used in point-to-point networks may include the star, loop, or branch arrangements, or a combination of these.

The second major connection method is broadcasting. In this design, the nodes may share a communications channel which carries all messages. Each node is capable of receiving all information on the channel, so something in the message must specify the intended receiver, allowing all other nodes to ignore the rest of the message. The most common broadcast topologies for small networks are the ring and bus networks. A ring is different from a loop in that the loop requires the whole message be sent to the next node for receipt or forwarding, while the ring operates by having each bit travel around on its own, being received by everyone on the network but processed only by the intended receiver.

Access to a broadcast network is allocated statically using time slices or dynamically by having someone centrally control access or allowing each node to determine their own eligibility to use the network.

For the project described by this thesis, it was decided to use a broadcast channel bus network structure with decentralized dynamically controlled access. This topology seems to allow the greatest flexibility for enhancements and growth to the system without a high overhead commitment nor major restructuring when new nodes are added in the future.

B. NETWORK PROTOCOL

In order for two or more computers to exchange information, there must be a mutually established set of rules and conventions governing the manner in which they will communicate. The collective grouping of these rules and conventions is known as a "protocol" [Ref. 2: p. 11].

Protocols are designed to accomplish different goals and overcome different problems.

The number in existence today is at least as large as the number of computer manufacturers. Computers with different protocols cannot communicate, therefore the need for an absolute standard is great.

In an attempt for standardization, the International Standards Organization (ISO) and the American National

Standards Institute (ANSI) proposed a model to serve as a guideline for protocol designers in achieving uniformity.

1. The ISO/ANSI Open System Interconnection (OSI) Model

In order to reduce the complexity of intercomputer communication, the model is divided into seven hierarchical layers. The layers are (in order from highest to lowest):

<u>LAYER NUMBER</u>	<u>NAME</u>
7	Applications Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Data Link Layer
1	Physical Layer

Each layer above the Physical Layer exchanges data with the next lower layer via well-defined interfaces. At the lowest level, the Physical layer of two computers exchange information. All seven layers perform different functions on data passed to them. Only the lower two levels have direct application to this thesis and the reader is referred to Tanenbaum [Ref. 2: p. 15-21] for an in-depth description of the upper five layers.

a. Physical Layer

It is the responsibility of the physical layer to transmit bits between computers. A few of the design issues

involved at this level include: choosing the medium to be used in the communications channel; picking voltage levels to differentiate "ones" bits from a "zeroes" bits; deciding whether or not data can be sent in both directions simultaneously; deciding how much time is occupied by a bit; and what pins will carry what signals on the network connectors.

It is assumed this layer is responsible only for accepting raw data and transmitting it across the channel without regard for structure or meaning contained within the bit stream. The physical layer has no responsibility for ensuring the transmitted information arrives at its destination or is not altered enroute.

b. Data Link Layer

As data is transmitted over a channel, there exists a very real possibility of some of it being lost or damaged. The task of this layer is to ensure that all data passed upwards from the Physical layer is screened. Only error-free output is passed from this layer upward to the Network Layer (Layer 3). This is normally accomplished in the sender by dividing the transmitted data into frames (a unit of data consisting of a set number of bytes), passing the frame to the Physical layer which transmits it. The other computer's Physical layer receives the frame and passes it up to its Data Link layer which in turn determines if the

frame was damaged or duplicated. An acknowledgement is sent for good frames received, and depending on the protocol in use, for the bad ones too. Depending on the acknowledgement received, the transmitting computer sends another frame with the process continuing until all frames have been received error-free. At this point, the Data Link layer passes the entire message up to the next higher layer.

2. Types of Data Link Protocols

One way of categorizing protocols is to group them by complexity. "Simplex" protocols only provide one-way transmission and are the most rudimentary. "Sliding Window" protocols are more efficient and also more complex. To better illustrate the functions of the Data Link layer, three Simplex and two sliding window protocols will be examined along with a technique called "piggy-backing". These just a few of a large variety of protocols in use.

a. Unrestricted Simplex

This first method is unrealistic and is presented only as a basis for understanding inherent problems with protocols. It includes the following assumptions:

1. Data is transmitted sequentially in only one direction.
2. The transmitting and receiving computers are always ready.
3. The time required to process data on either side is insignificant.
4. The transmission channel never loses or damages a frame.

The process is straightforward. The receiver enters a "waiting" loop prior to arrival of a frame. The sender fetches a file or other collection of data which it divides and forms into frames. The frames are then passed in sequential order to the Physical layer until all have been sent. Every frame arrives undamaged at the receiving computer's Physical layer and is passed upward to the other layers.

b. Stop and Wait

This Simplex method deals with the problem of the sender flooding the receiver with data faster than it can be processed. Still, it is assumed that the data flows in one direction and arrives error-free.

The easiest way to handle this problem is to require the receiver to provide feed-back to the sender. This can be done by requiring the sender to wait for a signal from the receiver indicating that it has completed its processing. Once received, the sender transmits another frame. During this short interval, the receiver processes data, clears its buffers, and reenters its waiting loop.

c. Positive Acknowledgement with Retransmission

It is unrealistic to assume that the communication channel is error-free. Static on the line or electro-magnetic pulses from a wide range of sources can change bits or damage frames. If the sender appends a

"checksum" (a special code) to a frame before transmission, it can be used on the receiving end to determine if any changes occurred in transmission. Depending on the code used, not only can errors be detected but they can also be corrected. This discussion will not deal with the generation of such codes. For more information on these, the reader is referred to Tanenbaum [Ref. 2: p. 125-132] for an excellent discussion on Hamming and cyclic redundancy codes.

Given that a receiver has the ability to detect errors, several methods are used to replace a damaged frame. One way is to have the receiver withhold the acknowledgement if the frame received was damaged. After the sender has transmitted a frame, it waits a predetermined amount of time. If an acknowledgement is received, it sends the next frame in the sequence. If no signal is returned, the sender retransmits the same frame. This would continue until either an acknowledgement was received or a certain number of retransmissions had been sent (in which case, it would abort the process). One problem with this method is the possibility of duplicate frames being received if the acknowledgement signal itself is damaged or not recognized by the sender. One way to deal with this is to give each frame a sequence number. This number would be contained within a field of a "header" attached to the beginning of the frame. By checking the header field containing that frame's sequence

number, the receiving computer could determine if a duplicate had been received and simply discard it.

A second method involves positive acknowledgement whether the frame is damaged or not. This time, it is the sender that enters a waiting loop. Upon completion of processing, the receiver acknowledges with a "success" or "failure" code to indicate receipt of a good or bad frame. If the sender receives a "failure", it retransmits the same frame. If it receives "success", the sender transmits the next sequential frame. Again, the possibility exists for the acknowledgement to be lost and a duplicate received.

The next two examples are of protocols belonging to the "sliding window" class. In these, both sender and receiver maintain lists. The sender's list contains sequence numbers of the frames it is allowed to send and the receiver's list contains frame numbers it is allowed to receive. The lists are referred to as the "sending window" and "receiving window" respectively. For each number contained in the sending window, there is a corresponding frame kept in the sender's buffer, the number of frames depending on buffer size. When an acknowledgement is received, the lower limit of the sending window is incremented by one. When a new frame is brought into the buffer from peripheral storage, the upper limit is incremented by one. Effectively, the sequence numbers

represent the frames that have been sent but which have not been acknowledged.

The receiving window contains sequence numbers corresponding to the maximum number it can hold in its buffer. Unlike the the sender, this window's size is constant. Any frame received with a sequence number falling outside its range is discarded. When a good frame is received with a number corresponding to the lower limit of the window, the receiver processes or stores the frame, returns an acknowledgement with the sequence number of that frame to the sender, and advances both boundries of the window by one. This arrangement allows great flexibility in that each side has more control over the communication.

d. Stop and Wait Sliding Window

This is essentially the same as the Simplex form. The only difference is the fact that both sending and receiving windows contain only one sequence number (i.e. the window size of both is one). After sending a frame, the sender waits for an acknowledgement of that sequence number before sending the next. Again, the receiver evaluates the packet for damage, checks the sequence number for duplication, stores or processes the good frame, and returns an acknowledgement of the good frame to the sender.

e. Sliding Window with Pipelining

"Pipelining" is a technique that can be employed to solve the problem of long propagation times in the communication channel. A typical case would be where one's computer is communicating with another via a satellite link. Round-trip transmission times of 20 msec or more would not be uncommon under these circumstances. A stop and wait protocol would be very inefficient if it had to wait that long between sending and acknowledging a frame. Instead, the sender transmits all frames for which there exists a sequence number in the sender's window. Window size is predetermined so that the sender just finishes transmission of the last frame before the acknowledgement for the first frame arrives. An obvious problem with this is the case where a frame in the middle of the transmission is damaged. Two solutions have been offered. One, the receiver discards and does not acknowledge the defective frame and all frames after it. The second involves the receiver saving and acknowledging all good frames while not acknowledging the bad frames. With this technique, frames could arrive out of sequence, but additional complexity would be involved for the receiver to put them back in order.

f. Piggybacking

One desirable feature is the ability to transmit data in both directions over one channel so that processes

running simultaneously on two computers can exchange data during execution. A technique called "Piggybacking" allows two computers to share the line and conduct synchronized communication. This operation would proceed as follows:

1. Computer number one sends a frame containing data from its currently running process to computer number two.
2. In the meantime, computer number two prepares to send a frame for its currently running process. In its header, there is a one-bit "acknowledgement" field. A "one" bit contained here means the last frame received by this computer (number two) was good. A "zero" means the last frame was bad.
3. Number two receives number one's frame, checks it for errors, and puts a corresponding "one" or "zero" bit in the acknowledgement field of the frame it is about to send.
4. Number two sends its frame.
5. When number one receives it, number two's frame is checked for errors in the data. Then it checks the header for the acknowledgement. If the acknowledgement was "good", it increments the sequence number field and sends the next sequential frame. If "bad", it will resend its last frame. In addition, number one will update the acknowledgement field to reflect the status of the frame it just received from number two. This process continues until all data has been transmitted.

3. Summary

The variety in Data Link protocols is wide. Each is designed to solve a certain type of problem. The efficiency of the one in use depends on a number of factors including: whether the protocol is pipelined or stop-and-wait, whether piggybacking is used or not, whether the line is half- or full-duplex, and the characteristics of transmission errors.

A protocol designer must consider these and other factors as he makes tradeoffs between efficiency, effectiveness, and cost.

III. CP/M

CP/M stands for Control Program for Microcomputers. It is not a computer language, nor is it a way for computers to control other machines, as some would have you believe. Instead, it is an operating system, and more specifically, a disk operating system which takes care of all of the details involved in controlling computer hardware to complete the many steps required by each simple command entered. For example, if you give the command to load a program, the operating system must determine what the load command means, tell the disk drive to spin the disk and move the heads to the correct position, read the directory and locations of all of the parts of the program, determine where in memory to put the program, and so forth. The operating system is the interface between computer hardware and the applications software. In general, if two different computers are each using the same operating system, both computers will be able to run almost all of the same software.

A. HISTORY OF CP/M

CP/M was developed in 1973 by Gary Kildall, a former faculty member of the Naval Postgraduate School in Monterey, California while serving as a consultant to Intel Corporation, a large manufacturer of integrated circuit

chips. Intel chose another operating system, so Kildall went on to market CP/M on his own, forming Digital Research Corporation in nearby Pacific Grove. CP/M was based on the Intel 8080 microprocessor chip which many microcomputers use, so manufacturers lined up to adopt CP/M rather than develop their own operating systems. As CP/M was revised, flexibility increased and its popularity snowballed. As more manufacturers adopted CP/M, more software was written to run on it. Acceptability soon made CP/M a de facto industry standard for microcomputers. The characteristics of CP/M which contributed the most to its popularity are portability and compatibility with a wide variety of microcomputers. These qualities also make CP/M a desirable choice for the basis of microcomputer networking.

Portability enables very different microcomputers to exchange disks, except in cases where manufacturers use different formats for storing data on disks. The registers within the computers are manipulated in the same ways regardless of manufacturer. This makes different computers behave as if they were identical. This portability is possible because the CP/M system actually consists of several relatively independent parts with clearly defined interfaces. Only one of these parts, the Basic Input/Output System (BIOS), deals with the host computer's hardware. Only the BIOS must be changed to adapt CP/M to another computer. In

this way, a program may say to print a message and the BIOS will provide the printer address unique to the particular computer. Programs may be written to run on CP/M in general, and may be run on any computer, with CP/M doing the work of figuring out the differences between all of the computer peculiarities.

Compatibility has allowed older CP/M version programs to be run on new versions of CP/M, calming the worries some users have about buying a system one day and finding out tomorrow that the industry standard has changed and he wasted his money on an obsolete system. A program prepared to run on CP/M version 2.2 should easily run on a CP/M 3.0 system.

B. ORGANIZATION OF CP/M

There are four main parts to CP/M. BIOS (Basic Input/Output System) provides the translation necessary for the computer to reach the disk drives and to communicate with the printer, telephone modem, and other peripherals. BDOS (Basic Disk Operating System) manages the floppy disk drives using the directory of information stored on each disk. The CCP (Console Command Processor) provides the needed translation between you at the keyboard and the rest of the system. The TPA (Transient Program Area) is like a storage bin in the computer that holds specific applications programs. A word processing program, for example, would be loaded into the TPA section of CP/M.

When you initially load CP/M from your master system disk, CP/M itself is loaded at the topmost free memory block which is about seven Kbytes long. Also, two jump instructions are placed at memory address 0000h (The h stands for hex, or base sixteen. This notation will be used throughout the thesis.) and 0005h. The space up to 0100h is taken with file control information, disk drive information, buffers, and other system parameters. The user may load his program into the TPA from 0100h to the beginning of CP/M high in memory, starting with CCP which is about 800h long. Next is BDOS and finally BIOS. CP/M uses the upper 18FFh of memory.

The CCP module interprets the CP/M commands you type. This is generally relevant only when you see the CP/M prompt "A>" (or the prompt for a drive other than A) on the screen. The CCP recognizes six or seven CP/M commands and several special subcommands. These will be explained later. When CCP doesn't recognize your command, it checks the directory for a file written to run under a single title word (.COM files) and loads the matching file into the TPA and runs it.

All disk drive activity passes through the BDOS section of CP/M. BDOS is accessed by having the transient program or CCP place an appropriate function code in register C and then a CALL to 0005h is executed. BDOS performs a few dozen functions involving general device-independent routines for

interacting with the console, printer, and disk drives. To aid file management, it uses the File Control Block (FCB), a file subdirectory, or stored data string, that has information used to describe the location, size, and allocation of the sectors of a file on the disk. The Digital Research Corporation manuals [Refs. 3 and 4] provided with CP/M explain details of how BDOS works together with BIOS to provide hardware-independent access to CP/M facilities.

The Basic Input/Output System (BIOS) is provided by the manufacturer or software vendor rather than Digital Research Corporation. It is the device-dependent part of CP/M responsible for disk drive head positioning and is similar to BDOS since it contains numerous function calls to read or write to input or output devices and peripherals. BIOS must be tailored to the particular microcomputer model on which it is to operate.

C. SYMBOLOGY

There are just a few special notations which are peculiar to CP/M.

The "greater than" symbol (>) is the prompt for CP/M. It indicates the computer is waiting for input from the keyboard and appears after the logged disk drive. When the system is started up, otherwise called booted, the copyright logo is displayed followed by a line saying "A>". This means you are logged onto drive A, the initial default drive, and the

computer is waiting for a command. The cursor will be next on the line. In a very few systems, such as Cromemco's CDOS, the prompt is a period vice a ">".

File names may have from one to eight characters which will be followed in most cases by a period and a filetype extension of three characters. The file names may contain upper or lower case, but most systems convert lower to upper case, so it is a good idea to stick to upper case characters. The name may not use a few special characters which have unique meanings such as <>.,;:=?*{} nor control characters.

Filetype extensions identify what the file is to be used for, since the system treats many types of files differently, such as interpreting it as a command, as text, or as binary data. It consists of three characters with the same restrictions as mentioned above. There is a quasi-standard list of extensions [Ref. 5: pp. 17-18] which are commonly used such as .COM, .HEX, .PRN, .TXT, and .\$\$\$. About twenty-two of these represent about ninety percent of all extensions, although you can make up your own, or omit it altogether. The .COM extension is used to denote a command file which is a program file which may be run simply by entering the filename without the extension on the command line.

CP/M addresses its available disk drives by capital letters A, B, C, D, and so on through P. On some systems, M,

N, O, and P are for hard disk drives. To differentiate a disk drive from a one-letter command, the colon is used immediately after the drive letter. For example, if you entered "A" followed by return, the computer would interpret this as a command to run the program A.COM. If you entered "A:" followed by return, the computer would make A the currently logged drive.

Another set of characters represents ambiguous characters and file references. These are known as the wild cards. The asterisk is used to represent an entire filename or filetype string. The question mark is used to represent a single character. This is why you can't use them in the titles of the filename or filetype. For example, if you want to erase all of the files relating to your UPLOADER program, you could type ERA UPLOADER.* which would erase UPLOADER.ASM, UPLOADER.HEX, UPLOADER.PRN, UPLOADER.COM, and UPLOADER.DOC. Notice the "*" can stand for any string in the field specified. The question mark means any character in that position. If you wanted to list all of the versions of a program on the disk, you could type DIR THESIS?.DOC which would display THESIS1.DOC, THESIS2.DOC, THESIS3.DOC, and so on, with the question mark being replaced by any existing character in that position of the file name. The meanings of the DIR and ERA commands will be explained later. For now,

just understand that the "*" stands for a string and the "?" stands for a single character.

D. COMMANDS

CP/M is composed of two basic groups of commands. The first set of instructions is known as resident or built-in commands which are always present in memory as long as the CCP is present. The second set of commands is called the transient commands which exist as programs on disk and are immediately loaded and executed when called. Each time it is used, the command program is brought in from the disk unlike the built-in commands which require no consulting from the disk to run.

1. Built-in Commands

The resident commands are loaded into memory with the CCP. They are executable whenever the prompt for a drive is present (such as "A>" for drive A). The commands may not be preceded by a disk name, such as "A:", because they are not associated with commands or files from a particular disk drive. Hence "A: DIR" would be improper. Some of these commands may use disk drives as parameters in which case the drive may follow the command, as in "DIR A:". The following paragraphs explain the six built-in commands.

"DIR" This command is used to list the file directory on a particular disk. CP/M can control up to sixteen separate disks. The default disk (the one you'll get

if you don't specify which drive you want) is the currently logged drive. A class of files may also be selected using wild card substitution characters or certain parameter restrictions. These parameters may also be used for other commands. The * is used as a wild card for a string (a filename or filetype in this case), and a ? is used as a wild character. For example, DIR *.ASM will list all of the files on the default disk with filetype ASM. SORT?.ASM could refer to SORT.ASM, SORT1.ASM, and SORT2.ASM. DIR does not tell where on the disk the files are located nor how big they are.

"ERA" This command is used to erase a file and free disk space for other use. In reality, the file is not really erased, but a change is made in the unseen part of the directory to prevent listing the file name and also making its space on the disk available for future storage. If the file has not been overwritten, there are means for recovering it.

"REN" Rename. This command is used to change the name of a file. This command requires specific names, so the wild cards * and ? aren't permitted. For example, REN NEW.ASM=OLD.ASM changes the name of the file OLD.ASM to NEW.ASM.

"SAVE" This creates a file on the disk and stores the image of the file from the contents of computer memory on the disk. It overwrites a file by the same name. SAVE takes

two parameters. The first is the number of 256 byte blocks, otherwise called the number of pages, to be saved. The second is the file name. For example, SAVE 4 NEW.ASM creates a file (or overwrites one) named NEW.ASM using the first 1K bytes (4 pages times 256 bytes) of transient program area. Unchanged files need not be saved since reading a file from the disk does not erase it.

"TYPE" is used to view an ASCII file on the console. Control-S is used to stop scrolling. Any other key terminates the output. Scrolling is resumed by pressing any key. This command may only be used on text or assembly files. Attempting to type a command file will result in special control characters being sent to the screen which may produce unintelligible garbage such as random characters, clear screen, flashing letters, bell, and random cursor movement.

"USER" allows up to sixteen people to share a system and the disks at different times and store files under their own number, so they can see their own directories without the confusion of others' files, even by the same name. This does not allow simultaneous use of one computer by several people. USER 0 is normally selected when CP/M is initialized.

"X:" is used where X stands for one of the sixteen disk drives (A through P), changing the logged drive to the letter substituted for X.

Control characters are recognized by CP/M for special functions. Chapter 4 of Hogan [Ref. 5] explains these immediately interpreted line editing commands.

2. Transient Commands

CP/M allows some programs to be executed as commands. These transient commands are stored on the disk, so they must be loaded at execution time. How does CP/M know if what you type is a command? It first checks to see if it is a built-in command described above, and then looks at the file directory to see if there is a command file (with the filetype extension .COM) with the first word in your command. If the .COM file is not found, CP/M will respond by displaying your command followed by a question mark.

CP/M requires a number of housekeeping utilities to manage files data, and text. The built-in commands do much of this, but not enough. Fortunately, Digital Research Corporation provides additional programs to aid the programmer. Some of these are described below.

ASM assembles a source file with the extension .ASM and produces a .HEX assembled object code file and a .PRT print listing file. The HEX file must be converted by the LOAD command to binary COM file in order to be executed.

DDT is a debugging tool used to locate and correct errors, make simple changes, and trace the execution of a program with .HEX or .COM extensions. It displays machine

language files in both hex and ASCII codes at the same time. It disassembles the program and recreates the assembler file from machine language.

DUMP is used to display files, including a hexadecimal representation of a .COM file. Remember TYPE doesn't allow this.

ED is a text editor which creates or modifies text or assembler files. It is rarely used since most people prefer to use commercially available word processors.

LOAD converts the .HEX listing produced during assembly, and yields a .COM file which is executable merely by entering the program title as a command.

MOVCPM changes the size of memory available to the Transient Program Area as long as your hardware is able to accommodate. The CP/M system is originally configured for 20K of memory.

PIP (Peripheral Interchange Program) copies files between disks and devices. PIP may be used to change names of files when transferring or perform a number of other functions on files depending on the parameters which follow the command.

STAT is a multipurpose program which tells you about disk space and program lengths. It can set files to "read only" to protect programs from accidental erasure, make a

program invisible to the DIR command, and redirect input and output among devices.

SYSGEN transfers the actual CP/M operating system program to a disk, allowing it to start up the computer under CP/M. This is placed on the outer tracks of the disk.

SUBMIT is used to link compilation or executable file management commands together. A file named XXXXX.SUB may contain commands such as STAT *.COM, ERA *.HEX, DIR *.HEX. When you type SUBMIT XXXX, the file XXXX.SUB is executed which is just like you typed the three above commands, one after the other. Various parameters and inputs may be included in the SUBMIT command.

XSUB is a subset of SUBMIT and appears only in the .SUB file and not in response to a CP/M prompt. When placed at the beginning of a .SUB file, it relocates to the area directly below CCP in order to process the command lines of the .SUB file and thereby provides buffered console input to the programs executed within the submit operation. Programs that read buffered console input get it directly from the .SUB file.

There are a number of other transient commands and programs available commercially which perform similar functions, many of which are used in place of the ones Digital Research Corporation provides. These can generally be organized into four classes. First, utilities are

programs which help maintain your disk collection such as disk copiers, formatters, disk viewers, and the like. High level languages are the second class and allow the user to program in commands which closely resemble English. Reading the program is then easier and more understandable than trying to make sense out of registered manipulation and machine instructions, although you have less control over exactly what happens in the computer hardware. Most programmers aren't concerned with hardware control, so high level languages are better, particularly if they must communicate with other's concerning their program. The third class is applications programs. This class is farthest removed from the hardware and machine instructions. Most are written in higher level languages and perform special functions. Examples are business programs for financial accounting, real estate, inventory management, or investment planning. The fourth class is word processing, a subset of applications programs. These are text editors which are more versatile than ED and allow easy editing and revising as well as providing help menus and print formatting.

IV. RS-232-C INTERFACE

Once it is decided which data to transfer and what format to use, it is necessary to actually send the data between computers. This requires some sort of physical connection. The type of network structure may already be decided (bus topology in the case of the thesis project), but a computer only knows to put data on the microprocessor data lines internally and not how to get it onto the network. For this, an interface is required.

There are two basic types of data transfer between computers and peripherals. Parallel transfers put the entire eight-bit word from the data bus onto eight wires of the network at the same time, requiring the receiver to accept the entire eight bits simultaneously. Serial transfers send one bit at a time over one wire to the receiver which reassembles the bits to make the word. Since the microprocessor data bus is parallel, a method is needed to convert the words into serial bits for transfer and back into parallel at the receiving end. Although this sounds complicated, the serial method of transfer has some distinct advantages over the parallel method in low cost expandable networks. Since parallel transmissions must be transmitted and received simultaneously, the length of cable connecting a computer with a peripheral or another computer may be very limited.

The longer the cable, the greater the chances of distortion on the line. In the microcomputer industry, some standards for serial transmission have been widely accepted. These standards make it easy to develop the Physical Layer interface between microcomputers made by different manufacturers.

Both synchronous and asynchronous modes of transmission are accepted schemes for serial transfers. Synchronous transmission requires a special pattern of bits to be sent before the data telling the receiver to divide subsequent groups of bits into words. This requires a high degree of synchronization between the transmitter and receiver. Asynchronous transmissions add a start bit and one or two stop bits to each word, allowing the receiver to automatically decode the middle bits into words. The only problem then is to ensure that both the transmitter and receiver are transferring at about the same rate. Common usage has defined some quantum rates among serial transfers between computers and devices. The RS-232-C industry standard doesn't specify particular rates, but establishes an upper limit of about 20,000 bits per second. Most common uses are 110, 300, 1200, 9600, and 19200 bits per second. The designed project was operated successfully at 300, 1200, and 9600 baud, although no anticipated failures would occur at higher rates. In general, the more computers attached to the network and the longer the network bus cables, the slower

the limits may be due to interference, line losses, and distortion. The important thing is that all computers on the network be set to the same rate, whatever that may be.

Just being able to send or receive data is not enough to allow the network to operate. Each computer must be able to communicate to others when it is physically capable of receiving or transmitting. This is done using "handshaking" signals. For the physical interface, these handshaking signals are separate from the handshaking signals referred to in Section VI and Appendix D which are treated as message data as far as the Physical Layer is concerned. Because of all of the possible schemes for handshaking, the Electronics Industry Association created the RS-232-C interface specifications which are widely accepted among microcomputer manufacturers and users [Ref. 6]. The wide acceptance and ease of use of these standards in satisfying the network requirements made the RS-232 serial interface the optimum choice for a low cost expandable microcomputer network.

The RS-232 standards defined two sides of the interface. The computer end was called the Data Terminal Equipment (DTE) end, and the equipment on the other end which is usually connected through the interface was called Data Communications Equipment (DCE) since a modulator/demodulator, or modem, is usually used to convert the data to signals for transmission over telephone lines for long distance

communications. For shorter distances, as would be the case in local area networks, one side of the interface is made to think it is the DCE.

There are a large number of possible protocols supported by the RS-232 [Ref 7]. The first is a one-way transmit-only configuration as might be used by a keyboard. The second is also one-way transmit-only, but contains more handshaking as in the case of a paper tape reader. Third is a one-way receive-only which might be used by a serial printer. The fourth configuration is two-way, although only one side may transmit at a time (half duplex), frequently used by the two-wire modem for telephone communications. The fifth configuration is a full duplex protocol that can transmit in both directions at the same time as in the case of four-wire modems. Nine other configurations are specified which are mostly combinations of the first five but support primary as well as secondary channel operations [Ref. 6: p. 21]. Some computers do not come with serial interfaces built in, so interface boards may be constructed or purchased for about one hundred dollars to support the various protocols and even offer some software support for controlling peripheral devices [Ref. 7: pp. 3-16 to 3-22].

The EIA standard provides ample description of the technical aspects of the RS-232 [Ref. 6]. It describes the electrical signal characteristics, mechanical interface

characteristics (for male and female connectors), functional interchange circuits, and communications configurations. The RS-232-C standard is applicable to both synchronous and nonsynchronous systems, so many of the circuits would never be used in the network proposed, allowing future enhancements to incorporate unused or inappropriate lines for other purposes such as handshaking.

One of the electrical signal standards that may be confusing to someone not familiar with the RS-232 is the definition of binary ones and zeroes. A "mark" or binary "one" is defined as a voltage more negative than minus three volts with respect to ground. A "space" or binary "zero" is greater than plus three volts. The region between plus three and minus three volts is considered a transition state. Most circuits use minus twelve and plus twelve volts with the ability to withstand a 25 volt limit in an open- or short-circuit condition.

The popular DB-25 connectors are used to attach various RS-232 interfaces. The pinout specifications are shown in Figure 4.1. The International Telephone and Telegraph Consultative Committee (CCITT) has also formulated standard circuits which are equivalent to the EIA standards. These are shown with the circuit descriptions and purposes in Table 4.1. The circuits are also broken into the four categories of ground, data, control, and timing.

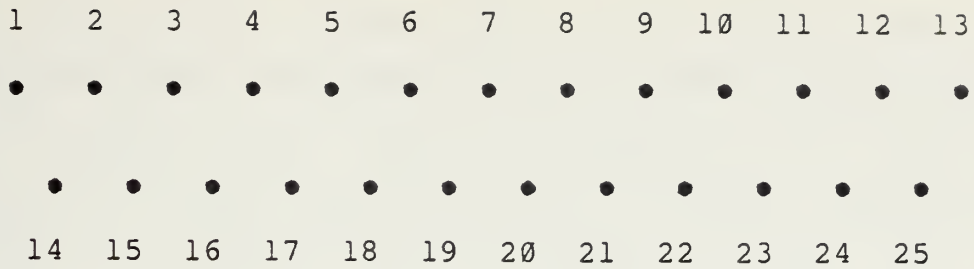


Figure 4.1 DB-25 Connector Pinout for RS-232-C

Table 4.1 RS-232 Signals

<u>Pin</u>	<u>EIA</u>	<u>CCITT</u>	<u>Name</u>	<u>Source</u>	<u>Category</u>	<u>Function</u>
1	AA	101	FG		Ground	Frame Ground
2	BA	103	TD	DTE	Data	Transmitted Data
3	BB	104	RD	DCE	Data	Received Data
4	CA	105	RTS	DTE	Control	Request to Send
5	CB	106	CTS	DCE	Control	Clear to Send
6	CC	107	DSR	DCE	Control	Data Set Ready
7	AB	102	SG		Ground	Signal Ground
8	CF	109	DCD	DCE	Control	Data Carrier Detect
9				DCE		Usually +12v
10				DCE		Usually -12v
11	208A	Bell	QM	DCE		Equalizer Mode
12	SCF	122	DCD2	DCE	Control	Secondary DCD
13	SCB	121	CTS2	DCE	Control	Secondary CTS
14	SBA	118	TD2	DTE	Data	Secondary TD
15	DB	114	TC	DCE	Timing	Transmitter Clock
16	SBB	119	RD2	DCE	Data	Secondary RD
17	DD	115	RC	DCE	Timing	Receiver Clock
18	208A	Bell	DCR	DCE		Divided Rcvr Clock
19	SCA	120	RTS2	DTE	Control	Secondary RTS
20	CD	108.2	DTR	DTE	Control	Data Terminal Rdy
21	CG	110	SQ	DCE	Control	Signal Quality Det
22	CE	125	RI	DCE	Control	Ring Indicator
23	CH	111		DTE	Control	Data Rate Selector
23	CI	112		DCE	Control	Data Rate Selector
24	DA	113	TC	DTE	Timing	Transmitter Clock
25				DTE		Busy

The main three lines used on the RS-232 are the Transmitted Data, Received Data, and Signal Ground (pins 2, 3, and 7 respectively). When two computers are connected together, the output from one (pin 2) must be connected to the input to the other (its pin 3) and vice versa. Pin 7 of both computers are then connected together. Other pins may be used as required. The project software was written to minimize the number of lines required. Alternative measures may be used as the network grows in size in that hardware may be traded for software by using signals on lines to determine such matters as whether the network bus is occupied or busy which might otherwise be determined using software. Specific signal descriptions are available for choosing the most appropriate lines for future enhancements [Refs. 6 and 7].

Figure 4.2 shows the simple schematic for the recommended project. The switches are used to ensure that pins two and three are crossed for transmitting and receiving computers. The notation of "sending" and "receiving" computers is arbitrary in that all that is required is to have pins two and three crossed. The convention was selected merely to ensure this and to allow for conditions where more than one computer may want to receive the message on the bus, with the understanding that no more than one computer may transmit files onto the bus using the project software. Since the choice is arbitrary, the system would also work if every

computer on the network reversed the position of their respective switch. It may be desired to use center-off switches as a "privacy" switch which essentially disconnects a computer from the network. Otherwise, it is assumed the switches will be in the "Receive" position except when initiating a transfer of files.

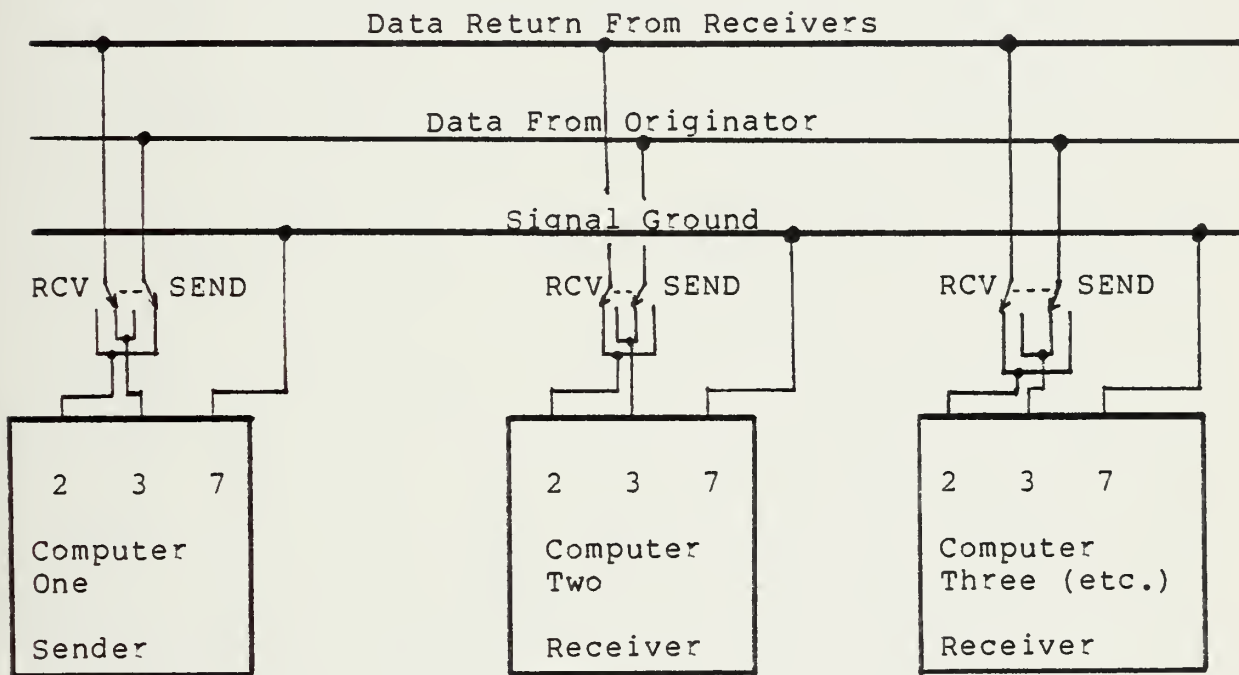


Figure 4.2 Project Network Protocol

Implementation of the network allows for either ribbon cable or multi-conductor wires. Each computer treats itself as a DTE terminal and the network is viewed as a single DCE device. Numerous DB-25 connectors may be connected to the bus to allow simple expansion of new computers to the network. If the bus is broken and spliced using DB-25

connectors, the network may temporarily become essentially two isolated independent networks. This method may be extended to offer nearly unlimited flexibility in networking the various microcomputers in the laboratory.

V. PRINCIPLES OF SOFTWARE ENGINEERING

Software Engineering deals with such issues as, writing software so that it can be understood, modified, and debugged easily by others, writing programs so they can run on a large number of computers with minimal reconfiguration ("transportability"), writing code so that it can be reused in other programs, and increasing the productivity of programmers. Brooks divides software into four categories: program, programming product, programming system, and a programming systems product [Ref. 8: pp. 4-6]. A program is a self-contained piece of software that runs alone, usually on one machine, and is maintained and used by one person. It differs from a "programming product" in that the latter can be run, tested, repaired, and extended by anyone. On the other hand, the "programming system" is a collection of interacting components that have been structured with precisely defined interfaces that allow the system to run on a variety of computers. Finally, the "programming systems product" is a combination of the preceding two categories with the characteristics of running on a variety of machines and being easily understood and enhanced by others. To achieve this level, it is estimated that nine times the amount of effort for a simple program is required for a programming systems product [Ref. 8: p. 6]. It is highly

desireable to plan for this latter type of software product when one is designing a network since many people will have to make an input to its design and it is only useful if it can work on a number of computers.

A. SOFTWARE DESIGN OBJECTIVES

The design and coding of complex software can be difficult unless one has clearly defined objectives. Ross suggests that the resultant software should have the following desired properties: modifiability, understandability, reliability, and efficiency [Ref. 9: p. 57].

1. Modifiability

The idea that the only thing constant in life is change itself, should be recognized when one begins to design software. It is estimated that between thirty to eighty percent of data processing budgets are spent on redesign and software changes [Ref. 10: p. 16]. There are many reasons for making changes. Changes are made because user requirements are often not clearly understood (even by the user) until after the project is coded. Changes are made to correct errors. A few errors don't surface until long after the system has been in operation. Then, there is the program that works so well that other users want to be added to the network and modification is necessary for it to be compatible with their computers. In any case, software designed for a

network should be easily modified to ensure that inevitable changes are "controllable".

2. Understandability

Network software is characterized by multiple, concurrently-running processes that interact with one another. This interaction is responsible for complexity that directly opposes the goal of modifiability. In order for others to maintain the system in operating condition, they must be able to understand how it works. It is the responsibility of designer and programmer to provide appropriate structure and organization in order that this complexity be manageable.

3. Reliability

This property requires that the system produce consistent results each time it is run. Applied to a network, this says that the system should be thoroughly tested and debugged. Features of the system that have the potential for causing errors should be partitioned off or isolated from the rest of the system so their performance can be monitored.

In addition, the system should have the ability to recover from failure during operation. As many potential error conditions as possible should be anticipated and handled in order to provide the system with "robustness".

4. Efficiency

A massive, slow running program can defeat the purpose for computerizing the application if speed was an issue. However, the goal of efficiency is often cited as an excuse for sacrificing the other goals. Clearly, efficiency must be carefully balanced with other trade-offs. For example, understandability is very important to efficiency during program testing. The extra execution overhead paid to make the program understandable must be weighed against cost of a systems analyst trying to debug the program when it doesn't run. Efficiency cannot be ignored nor can it be allowed to become all-important.

B. SOFTWARE DESIGN PRINCIPLES

Not all of the preceding goals are realizable in their entirety. As suggested before, many tradeoffs are necessary to achieve an optimum mix. There are a number of principles that when '...applied in various combinations within the fundamental process, will work to...achieve our goals during all of the various phases of software development' [Ref. 9: pp. 58-60]. These are discussed briefly as follows:

1. Modularity

The principle of modularity suggests that programs be built from a number of independent, "building-block" modules. Each sub-program composing one of these modules has little if any dependence on other modules except through explicit,

well-defined interfaces. Just as a building is constructed, modules are built on top of other modules in a hierarchical structure. Modules making up the lowest layer are dependent on the facilities of the individual computer. However, the purpose of these "foundation" layer modules is to perform the transformation that permits the modules in the next layer to be machine "independent". Thus, "dependence" is isolated or "encapsulated" in the lowest modules. By simply rewriting the foundation modules, the program can be used on other machines without rewriting the entire program.

After one module has been coded and debugged, it can be assembled or compiled by itself. Subsequent changes to other parts of the program do not require reassembly or recompilation of a completed module. Also, these completed modules can be placed in a module library for re-use by other programs, thereby saving additional development time on other projects.

Modularization has a parallel in shipbuilding. By sectioning the ship into water-tight compartments, when one module or compartment springs a leak, the damage can be quickly found and isolated, saving the ship from sinking. The doors and hatches between compartments are the interfaces that define how things may pass between compartments. They provide the control necessary to prevent water from spreading when there is a leak, that is, provided they are closed

when the leak starts. In program modules, the interfaces define what data may pass between modules. Thus, when errors occur, they are isolated and are easier to find.

Another feature of modularization is that it allows one to use the module without having to completely understand it. The real-life parallel to this is a computer. Not many computers would exist if every technician had to understand every circuit in the system. As is, the technician merely follows standard trouble-shooting procedures until he finds a circuit board that is not providing the correct output for the right input. To repair it, he could go and learn everything he could about the board in order to make the repair, but more likely, he will just replace the board or module. However, in both the real-world application as well as software development, this is only possible if the interfaces (range and domain of inputs and outputs) are well defined and unchanging. An excellent discussion of the advantages and disadvantages of modularity is contained in Shooman [Ref. 10: pp. 107-114].

2. Abstraction

The idea of abstraction is not to remove essential properties from something but to extract the inessential detail. A programmer writing in Pascal need not know that his "DO WHILE" statement will cause a positive fifteen volts to be applied to a certain wire at a certain time. This

inessential detail has been "abstracted" out of his problem. All he needs to know is when he follows certain rules, the desired results will be obtained. An opposite analogy can be found in the Army. When soldiers wear camouflage, it is difficult to distinguish them from the surrounding terrain because of the abundance of non-essential detail added to their bodies.

In software development, abstraction goes along with the goal of understandability. Unless abstraction is used to reduce complexity, the program will be incomprehensible.

3. Localization

The principle of localization is concerned with bringing similiar things into physical proximity. Grouping code according to its function (e.g. subroutines, arrays, and records) is an example of software localization. The ability of the average human to remember a number of unrelated items is very limited. This becomes obvious when one looks up a long-distance telephone number and has to walk across the room to dial it. When items are organized in some meaningful structure, more can be remembered for a longer time. In a program, this can add greatly to comprehension. The unrestricted use of 'GO TO's is a violation of this principle.

4. Hiding

This principle conforms to the concept of "information hiding" as conceived by D. L. Parnas

[Ref. 11: p. 223]. The purpose of hiding is to make features of a module invisible or inaccessible to other modules, thus preserving their independence. An analogy of this idea can be found on any large computer system. The user is only permitted to access features of the machine through the operating system, not directly. Because the operating system "hides" features of the computer, it is able to intercede and prevent undesirable effects that could occur if the user was allowed direct access. In software, features are "hidden" within modules. Within a module, only those that are specified by the interface can be accessed from another module. Because of this, complex interdependencies are avoided making the modules independent and the program more easily modified.

C. TOOLS FOR SOFTWARE DESIGN

The productivity of the network software programmer can be increased in a number of ways. Two of these are 1) choosing the right programming language for the job, and 2) the use of an effective programming style.

1. Choice of Programming Language

Programmers have been discussing the advantages and disadvantages of using assembly language versus a higher order language (HOL) in their work for a long time. On one hand, it is argued that assembly language executes faster and uses less memory to run. On the other, it is said that

HOL's are easier to understand and modify. In defense of assembly language is the fact that they can directly manipulate I/O ports and registers, a requirement for network software. However, studies have shown that the use of HOL's can increase productivity as much as five times over that of assembly language [Ref. 8: p. 94]. A reasonable choice for programming network software might be a HOL with some capability for direct control of I/O operations. A few HOL's offer this capability (e.g. PL/I and "C", both systems programming languages). However, one must also look at the system on which it is to be run. If implementation is to be on microcomputers, one must consider the memory required by the compiler during its operation. In the final analysis, one must consider all facts and choose based on the merits of the language and requirements of the application.

2. Programming Style

As mentioned before, understandability of a program affects the amount of effort required to keep a software product viable as it grows and evolves. Programming style is a factor in its understandability. Style is more than just good documentation. It includes such things as using unique, meaningful mnemonics for variables and program labels. It calls for reducing unnecessary branching and the use of library functions. It says that multiple exits from loops and subroutines should be avoided. To assist the human

mind with its limited short-term memory, good programming technique dictates the use of indentation to provide logical grouping of code for easy comprehension. More information on programming style is available [Refs. 10 and 12].

VI. DESCRIPTION OF PROJECT SOFTWARE

The goal of this thesis project was to conduct preliminary planning for a Local Area Network (LAN). In addition, experimentation was conducted on a prototype for the Data Link Layer of the network software. A description of that effort follows.

A. DISCUSSION OF REQUIREMENTS AND ASSUMPTIONS

The microcomputer lab is slowly expanding as funds become available and as faculty/student requirements increase. The lab's microcomputers presently number less than a dozen but could dramatically increase in the future depending on a number of factors. The present configuration provides for individual microcomputers to have their own printers, a situation which is currently satisfactory. However, this is wasteful and could be eliminated if sharing of printers were available. While mechanical switching could be used to permit this sharing, it was felt that sharing of peripheral devices should be a feature of any future LAN.

One long-range goal is to set up an electronic "bulletin board" that would permit students to exchange software between computers in the lab and their own home computers. Peripheral sharing would be beneficial in this situation,

especially if a common hard-disk were available to computers on the network for mass storage.

Since experience has shown that most computer systems experience "mushroom" effect in growth once users are comfortable using them, it was felt that a networking scheme designed for this application would require flexibility and ease of expansion. On the other hand, since funding was uncertain, the design would have to be relatively inexpensive, at least in its early stages. Because permanent personnel were not used on the project, the software would have to be as simple and understandable as possible to accommodate a high rate of personnel turnover and the inevitable loss of knowledge. In summary, the requirements for the future LAN were assumed as follows:

- The system would have to be relatively inexpensive.
- Software would have to be simple, reliable, and easy to understand.
- Software would need to be flexible in order to easily accommodate an unknown number of future enhancements.
- Connection of additional computers and devices to the network would have to be convenient and inexpensive.
- The capability to share common peripherals and to implement a future electronic "bulletin board" would have to be included in preliminary planning.
- The system would have to be easy to use.

B. DESIGN DECISIONS

A number of successful LAN's existed on the market at the time of this project. One of these, Xerox Corporation's ETHERNET, had a number of design features that looked promising [Ref. 1: p. 395]. By using a bus topology with network control shared by all microcomputers, overall reliability of the system was high. Since there was no central computer to go bad, failure at one node meant that the network could continue to operate. For communication between computers, a method called "common broadcast" was employed. Using this, when one computer desired to communicate with another, it simply broadcast its message over the net. The message would be copied at all nodes, but only the one node with the address specified in the message would respond. This was advantageous in that new nodes could be added easily by simply adding a new address. Ethernet also provided each node with the ability to "sense" if the bus was "busy" in order to prevent nodes from simultaneously trying to use the net. Unfortunately, Ethernet and other available networks were relatively expensive for this application due to the requirement for special hardware controllers used at the nodes. As a result, the decision was made to design a system in-house that would be similar to Ethernet. To hold down cost, it was decided that a loss

of performance would be accepted where software was substituted for more expensive hardware.

The microcomputer lab was established to support user experimentation and familiarization with different types of computers and equipment. Because existing computers were of dissimilar manufacture, it became apparent that the network software would have to facilitate communications between heterogeneous operating systems also. Due to the complexity of this requirement, it seemed easier to require a standard operating system. Since many microcomputers use Zilog Corporation's Z-80 microprocessor, which in turn, supports the widely-used Digital Research CP/M operating system, it was decided that all computers on the network would be required to have this combination of Z-80 and CP/M. For those computers which did not have the Z-80 (e.g. Apple II computer), participation on the network was possible by use of commercially available interface circuit boards that would "emulate" or make the computer appear to be a Z-80 as "seen" by the network. It was felt that these decisions would allow a flexible yet relatively inexpensive system.

The network program would be designed for ease of use by employing an interactive or "conversational" dialog with the user. In simpler terms, this means that the program would anticipate the user's desired actions, present him a "menu" of available options, ask him for his choice, and carry out

the action. This technique has gained wide acceptance in user communities and has proven to be more "friendly" to the casual user than previous methods which required the user to spell out his desires. A number of articles on man/machine interaction were used as guidelines for this part of the endeavor [Refs. 13,14, and 15].

To support the requirements that software be flexible and easy to understand, it was decided to apply the principles of modularity, abstraction, localization and hiding. Several "macro" libraries would be created and used by the network program. (Macros will be explained in detail later in this section. For now, let it suffice to say that a macro is similar to a subroutine). This would save the programmer from rewriting duplicate code and would "hide" much of the unnecessary detail from anyone seeking to understand the program. The largest library would contain machine-independent functions while other smaller libraries would contain functions specific to a particular computer. In this case, small libraries were written for the Input/Output (I/O) functions of the Apple II and Northstar Horizon computers [Appendices F and G]. With only small changes and minimal effort, an existing copy of one of these I/O libraries could be adapted to work for a new computer type. Because of their modular structure, macros could be added easily to support future expansion. Also, because of their

well-defined interfaces, the macros are reuseable, thus saving program development time and effort.

C. SOFTWARE IMPLEMENTATION PLAN

The common function performed by all networks is data transfer. Regardless of whether the process is to share peripherals or set up an electronic bulletin board, there is a need for error-free data to be exchanged. The programs written for this portion of the long-term project perform only basic file transfers between two dissimilar computers. Allowances were made for future work to be done to expand to multiple computer interaction. Rather than start from scratch, the decision was made to modify and adapt existing file transfer programs so that they would support network features. The programs selected were UPLOAD.ASM and DOWNLOAD.ASM written by Neil Konzen of Microsoft Corporation [Appendices A and B]. Attention is directed to these programs to make a point. While they efficiently perform their function of transferring files, they are hard to understand. Even an assembly language programmer with a trained eye would have to read slowly to comprehend them.

In order to keep from "reinventing the wheel", it was decided to use existing macro routines written by Alan R. Miller [Ref. 16] to implement the majority of macros needed in the main macro library (CPMMAC.LIB) however, the microcomputer I/O libraries were developed simultaneously.

While a HOL (higher order language) would have been a better choice for understandability and productivity, the decision was made to use Intel 8080 Assembly code. Languages such as "C" were commercially available for microcomputer application, but were not chosen because, 1) the funds were not available to purchase the "C" compiler, 2) the authors had had little experience with "C" and a fair amount of 8080 experience, and 3) it was felt the amount of time to learn a new language would be excessive. Information on Intel 8080 assembly code is available [Ref. 17].

The Digital Research "MAC" macro assembler was chosen for several reasons. One, it not only supported the use of macro's but it allowed the programmer to call them from an external file (a library) at the time of assembly. This would keep the program from being "cluttered" with code and would reduce the size requirements of the source program. Second, MAC allows mnemonics that are fifteen characters long. Here it was assumed that a longer name would carry more meaning and therefore be more understandable. One disadvantage to MAC is that it was designed only for Intel 8080 machine code and certain unique functions of the Zilog Z-80 microprocessor cannot be directly invoked. One way around this was to directly insert Z-80 machine code into the 8080 assembly code using the 'DB' pseudo operation (Define

Byte assembler directive). More information on other features of MAC and Z-80 code are available [Refs. 4 and 18].

Again, CP/M was chosen because of its universality among eight-bit microcomputers described previously. More information is available on CP/M [Refs. 3 and 16].

D. THE MAIN PROGRAM (CROSS.ASM)

This discussion will assume that the reader has had experience in programming Intel 8080 assembly language and some familiarity with CP/M. Flowcharts of the program are provided [Appendix C] and extensive documentation is in the program listing [Appendix D], but minimal explanation will be made as to how the macros work. For the ambitious, detailed explanation is available. [Appendices E, F, and G and Refs. 3 and 16].

To use the program for file transfer, one must load and run the CROSS program on both computers. Assuming that physical connections have been made correctly (described in the previous sections), one is greeted with a prompt that informs the user on what computer this version of the program is designed to run. This was necessary because of a tendency for the program's object code to be cross-loaded to other computers where it would not run without change of I/O parameters and reassembly.

After being informed of the computer type, the user is presented a menu of three choices: exit from the program,

send a file to another computer, or receive a file from another computer. If the program is on the wrong type of computer or was loaded by mistake, typing the letter 'E' (upper or lowercase) will allow the user to exit. Typing 'S' will put the user in the "SEND" mode, while typing 'R' will put him in the "RECEIVE" mode. Any other letter will cause an error message to be displayed and the user returned to the menu. (In the future, other choices could be made available here). The user is then asked to type in the name of the file and the program then checks to see if the file exists on the current disk. In the "SEND" mode, if no file by that name exists, a prompt on the screen informs the user, and he is returned to the menu. In the "RECEIVE" mode, if the file already exists on the current disk, the user is informed by a prompt and asked if he still desires to receive the named file. Typing the letter 'Y' will allow execution to continue with the existing file being overwritten by the received file. Typing an 'N' would return the user to the menu with no transfer taking place.

Bear in mind, transfer of a file requires the user to specify 'S' or SEND mode on the sending computer, and 'R' or RECEIVE mode on the receiving computer. At this stage of development, the two modes run independently on their own respective computers. One could send a file called "A.ASM"

and tell the receiving computer that it is getting "B.ASM". The receiving computer cannot tell the difference.

Prior to transfer, the sending program "opens" the desired file. Basically, this process gives the program access to the disk directory [Ref. 3: p. 100]. Using the directory, a program can find all sectors of data on disk belonging to the file in question. The process that does is contained in the macro "OPEN" (use of the hiding principle mentioned earlier).

On the receiving end, a similiar process takes place. Once it has been determined that the file to be received does not already exist, the receiving program "makes" the file [Ref. 3: p. 103]. This process records the new file name on to the disk directory and permits a file by that name to be added to disk. This process is hidden within the macro "MAKE".

One procedure performed by both sending and receiving computers prior to transfer is to set up Direct Memory Access (DMA) areas [Ref. 3: p. 104]. During transfer, the sending computer will read a sector of data from the specified file on its disk and write the data into a "designated location" in its memory. At the receiving computer, a similiar but opposite process occurs. When a full sector of data has been received and placed in a "designated location" in its memory, the sector is read from

that location and written to the receiving computer's disk for storage. DMA is the name given to these "designated locations" and a special function of CP/M allows the user's program to specify where in memory the DMA area will be located. In the CROSS program, the macro "SETDMA" uses CP/M to perform this function.

At this point, both computers go through a process called "handshaking" which basically synchronizes them. The macro "HANDSHAKE" performs this process for both computers. Once handshaking is complete, the transfer begins.

1. Actions by the Sending Computer During Transfer

The first action of the sending computer is to invoke the macro "READSECTOR". Each time READSECTOR is called, the next sequential sector of data in the file on disk is read, and copied into the sending computer's memory at the DMA location. If all sectors in the sequence have been read (end of file condition), READSECTOR notifies the main program.

After reading a sector from disk, the program attaches a number of bytes of data, called the "header", to the beginning of the sector, and one byte of data, the "checksum", to the end. The data in the header can contain such information as the network address of the sending and receiving computers, a sequence number, a code from the sending program to the receiving program signaling 'end of

file transfer', and many other pieces of information. At this time, only one byte of the header is used. The rest of the header is saved for future enhancements. The checksum is used by the receiving program to verify whether or not the data has been damaged in transfer. The whole package including header, sector, and checksum is called a "frame".

If the end of file has not been reached, the sending program invokes the macro "SENDFRAME". SENDFRAME goes to the location in its memory containing the first byte of the header (labeled "START\$OF\$FRAME"). That byte is sent to the RS-232 port to be transmitted over the network bus to the receiving computer. Then the byte following START\$OF\$FRAME is transmitted and the next and so forth until the whole frame has been sent.

Afterwards, the sending program enters a loop waiting to receive acknowledgement from the receiving computer. As the sending program waits, it repeatedly checks the RS-232 port for receipt of the characters 'B' or 'G'. Anything else received is ignored. A 'B' acknowledgement causes the sending program to retransmit the same frame, whereas a 'G' causes the sending program to reinvoke READSECTOR and transmit the next sequential frame. Once all sectors have been read and all frames sent, the sending program transmits one last "dummy" frame containing garbage data and an 'end of file' code in its header. Upon receipt

of this, the file is considered transferred and the sending program returns to the menu.

2. Actions by the Receiving Computer During Transfer

Upon commencement of transfer, the receiving program invokes the macro "GETFRAME". GETFRAME enters a loop checking for a byte of data received at the RS-232 port, or for an abort signal (control-C) from the user via the keyboard. If the abort signal is received, the program exits the loop and returns to the menu. If data is received at the the port, it is copied into the receiving computer's memory (beginning at the label START\$OF\$FRAME) and the loop is re-entered. This continues until a whole frame has been received. Then, the program compares the transmitted checksum with its own internally generated checksum. If they don't match, the data is bad (damaged). The receiving program returns a 'B' (bad) acknowledgement signal to the sending program and prepares for retransmission of the same frame. If the checksums do match, the frame was good (undamaged). The receiving program then checks the header for an 'end of file' code. If it is not found, the receiving program then invokes the macro "WRITESECTOR".

WRITESECTOR performs in a similiar but opposite function to that of READSECTOR macro. The macro determines the location in memory that corresponds to the sector of data and copies it onto the disk leaving behind the header and

checksum. Once the undamaged sector is safely stored on disk, the receiving program returns a 'G' (good) acknowledgement signal to the sending program, and prepares to receive the next frame.

When the "dummy" frame containing the 'end of file' code in the header is received, it is left in memory and the macro "CLOSEFILE" is executed. CLOSEFILE updates the directory with the sector locations of the new file, and records the information onto the disk. At this point, the file is considered received and the program returns the user to the menu.

E. DESCRIPTION OF THE MACRO LIBRARIES

As mentioned before, a macro library is a separate file containing macros (subprograms) used by an assembler during its operation.

1. Macro Assembly

The assembler takes the English-like instructions contained in the original file, e.g. CROSS.ASM (also called the "source" file), translates them into machine code, and puts this code into a new file called the "object" code or "object" file. It is the "object" code that one loads and runs to execute the program.

When the MAC assembler encounters a macro name (e.g. READSECTOR) during assembly, it branches to the library file and locates the subject macro. MAC then takes the English-

like instructions found there and translates them into machine code. For every occurrence of that macro name found in the source file, MAC substitutes the macro's machine code into its corresponding location in the object file.

The similiarity between a macro call and a subroutine call exists only in the source program. Whereas the occurrence of a subroutine call causes branching to take place during execution of the object file, the occurrence of a macro call causes branching only during assembly.

2. Advantages of a Macro

The macro has the advantage of being "inline" code. During execution of the object file, the program does not have to jump when it encounters the machine code generated by a macro. This tends to make the program run faster. However, the side effect of this is that macros "expand" during assembly and create more machine code than a subroutine. For example, given a macro and a subroutine with approximately the same amount of code, four macro calls will generate roughly four times as much code as four subroutine calls. There are ways to reduce this side effect and one of these was used in CROSS.ASM [Appendix D].

The other advantage of a macro is the ability to specify parameters. This allows a macro to perform several functions and only specify the desired function at time of assembly. As an example, HANDSHAKE macro operates for the

sending program when the parameter TRANSMIT is specified. The same macro performs a different action for the receiving program when the parameter RECEIVE is used. A subroutine has no equivalent.

3. Criteria for Dividing Macros Between Libraries

The objective is to put all macros that are universal, that is, able to execute on any CP/M computer without modification, into the main library file, CPMMAC.LIB. The small number of macros that remain are those that are machine-dependent. The majority of these are dependent because they contain code that is used for I/O on a specific type of computer. All of these macros are placed into libraries bearing the name of the computer type of which they are dependent (e.g. APPLE.LIB and NSTAR.LIB). The number of macros and the functions they perform are the same in the I/O libraries, the only difference between them is a small number of variations in the code they contain.

VII. CONCLUSIONS AND RECOMMENDATIONS

The thesis project was designed to be the basis for a microcomputer interconnection scheme. To provide for further expansion and standardization, the project was designed to have applicability to other organizations desiring low cost practical networking of microcomputers. Certain design considerations included the choice of network structure, operating system, and interface standards. For reasons discussed in previous sections, a broadcast channel bus topology was chosen using RS-232 serial interfaces and the CP/M operating system. During both hardware and software design, consideration was given for future change and enhancements beyond the scope of the initial project which included a working file transfer program developed as a prototype for the Data Link Control Layer of a future microcomputer-based local area network (LAN).

As the basis for future development in this area, the following recommendations are provided:

1. Improve Programmer Productivity: Ways of improving programmer productivity should be investigated. As mentioned previously, use of a HOL should improve productivity and understandability. Consideration should be given to how much control over registers is available to the

programmer and how much microcomputer memory would be used by the HOL compiler.

2. Reduce Assembly/Compiler Time: As the program size increased, it became glaringly obvious that an excessive amount of time was being spent in re-assembling previously debugged and assembled code. Consideration should be given to using a "linking loader".

3. Relocate The Object Code In Memory: Consideration should be given to giving the object code the ability to move itself into high-memory upon running but prior to data transfer. Several methods for doing this can be found. The reason for this is that CP/M requires all executable programs to be located at memory location 0100h. Once CROSS is loaded at 0100h, no other programs can be executed by CROSS. If CROSS could move, then such commands as STAT, TYPE, DIR, etc. could be run by CROSS.

4. Experiment With Multi-Computer Operations: The program developed here dealt only with two computers using the network bus. Work needs to be conducted with one computer interrupting and sending a message to one or more computers. Work is needed to decide the best way to control one computer from another. Also, experimentation is needed to define how the computers will decide which computer will use the network bus and how to settle disputes. It may be

desired to replace the hardware switches which control access to the bus with software control.

5. Design I/O Libraries for Other Microcomputer Models:

The software has been designed to modularize the code specific to each manufacturer model. The existing libraries may be used as a guide along with technical data unique to any microcomputer to expand the network to any model capable of serial transfer of ASCII characters.

6. Allow Microcomputers to Share Peripherals: One of the goals of any network is to allow stand-alone operation of each computer yet the convenience of virtual control of peripherals without any knowledge of what is "behind the keyboard." To eliminate wasted resources and impatience on the part of users who must wait for someone to "get off the only machine connected to the printer," the network could control resources by allowing anyone at any node to have access to peripherals such as printers.

7. Interface with External Networks: By connecting this network into larger networks and systems via modem, the capabilities become limitless. Adding mass storage would even allow electronic mail, long distance file transfers, and media exchanges. Personnel with valuable information and new software could send their discoveries to the laboratory from across the country. Many such software packages exist for

this purpose which are simple to install and at costs reasonable to the small organization with limited funds.

APPENDIX A: UPLOAD PROGRAM

Written by Neil Konzen (C) 1980 Microsoft

Modified and adapted for the NorthStar Horizon
by Professor Gordon Latta, Naval Postgraduate School, 1983.

```
BOOT    EQU 0           ;Warm boot address
BDOS    EQU 5           ;Bdos address
FCB     EQU 5CH         ;File control block
BUFFER  EQU 80H         ;Default dma address, and buffer

ORG 0100H

UPLOAD:
    JMP ENTRY

INIT:
    RET                ;if any specific initialization is
                       ;needed for proper I/O it goes here.
                       ;Most machines are already
                       ;initialized, hence RETURN.

    MVI A,4EH
    OUT 5
    MVI A,37h         ;specifics for North Star Horizon;
                       ;8 data bits.
                       ;2 stop bits, no parity
    OUT 5             ;N* control port
    IN 4              ;clear the input port by a read
    RET

INPSTS:
    lda STATUS        ;input status port
    ANI RXRDY         ;receive data available bit
    RZ                ;loop here till data available
    lda data          ;get the data byte
    RET

OUTPUT:
    PUSH PSW          ;save output on stack
OTLUP:
    lda status        ;read port status
    ANI txrdy         ;test output buffer empty
    JZ OTLUP         ;loop here till ready
    POP PSW           ;recover data from stack
    sta data          ;ship it out
    RET
```



```

ENTRY:
    LXI D,INMSG
    mvi c,09h
    call bdos           ;display signon message
    call crlf
    call crlf

INLUP:
    mvi c,01h
    call bdos           ;wait here for <cr>
    cpi 0dh
    jnz inlup
    call crlf
    call crlf
    LDA BUFFER         ;read first byte of buffer
    ORA A              ;test for a zero
                        ;(meaning no file name entered).
    LXI D,CMDMSG       ;prepare to display command message
    JZ EXIT            ;but abort if no file was entered
    CALL INIT          ;do any initialization now
    MVI C,15           ;cpm function to open file of
                        ;desired name.

    LXI D,FCB
    CALL BDOS          ;do so now
    INR A              ;ff=failure to find file, ff+1=0
    LXI D,FNFMSG       ;show file not found if acc=0 now
    JZ EXIT            ;and abort

RDYLP:
    mvi a,'R'          ;we're all set, send out an 'R'.
    CALL OUTPUT
    CALL INPSTS        ;and wait for an
                        ;answering 's' before going on.

    CPI 'S'
    JNZ RDYLP          ;loop till communication established
    mvi a,'G'          ;we're in contact , 'G' for go ahead
    CALL OUTPUT
    LXI D,WRKMSG       ;show "uploading message" on screen
    CALL PRMSG

READ:
    MVI C,20           ;read sequential from opened file
    LXI D,FCB
    CALL BDOS          ;this reads one 128 byte sector
    ORA A              ;a=0 for successful read and
                        ;not at file end.
    JNZ EOF            ;a not zero says end of file

```


TRYAGN:

```
LXI H,BUFFER      ;dma address in h,l
MVI C,0           ;initial checksum in c
MVI D,80H        ;sector count (128 bytes)
```

LOOP1:

```
MOV A,M
CALL OUTPUT      ;transfer the sector to
                 ;the output port.
XRA C           ;computing new checksum as we go
MOV C,A         ;saving it in c
INX H
DCR D           ;updating the registers
JNZ LOOP1       ;and looping till d,e=0,0
MOV A,C
CALL OUTPUT      ;then send across the checksum
```

VFYLP:

```
CALL INPSTS
CPI 'B'         ;now wait for
                 ;answer (handshaking signal).
JZ TRYAGN       ;if it is 'b' for
                 ;bad read, try it again.
CPI 'G'
JZ READ         ;if it is 'g' for
                 ;good read, get next sector.
JMP VFYLP       ;anything else is to be ignored
```

EOF:

```
LXI D,DONMSG     ;get ready to print done message
```

EXIT:

```
CALL PRMSG       ;print same
JMP BOOT         ;and exit gracefully
```

PRMSG:

```
MVI C,9
JMP BDOS         ;cpm function call to
                 ;print message.
```

crlf:

```
mvi c,02h
mvi e,0dh
call bdos
mvi c,02h
mvi e,0ah
jmp bdos
```



```
CMDMSG:
  DB 'COMMAND ERROR$'
FNFMSG:
  DB 'FILE NOT FOUND$'
DONMSG:
  DB 13,10,'UPLOAD COMPLETE$'
WRKMSG:
  DB 'UPLOADING...$'
INMSG:
  db 'apple upload program'
  db 0dh,0ah
  db 'Type in UPLOAD FILE.NAM'
  db 0dh,0ah, 'where file=name of desired file'
  db 0dh,0ah, 'and nam is the type'
  db 0dh,0ah
  db 'Enter <cr> when ready to procede$'
status: equ 0e09eh
data:    equ 0e09fh
rxrdy:  equ 01h
txrdy:  equ 02h
```


APPENDIX B: DOWNLOAD PROGRAM

Written by Neil Konzen (C) 1980 Microsoft

Modified and adapted for the Apple II
(with Microsoft CP/M) by Professor Gordon Latta,
Naval Postgraduate School, 1983.

```

BOOT    EQU    0000H           ;warm boot address
BDOS    EQU    0005H           ;bdos address
FCB     EQU    005CH           ;file control block
BUFFER  EQU    0080H           ;default dma address and
                                ;buffer.

COMSTS  EQU    0E09EH
COMDAT  EQU    0E09FH
RXRDY   EQU     01H
TXRDY   EQU     02H
APPKBD  EQU    0E000H

ORG 0100H
    lxi d,inmsg
    mvi c,09h
    call bdos                 ;print signon message
    call crlf
    call crlf                 ;pretty up the screen
INLUP:
    mvi c,01h
    call bdos                 ;wait for keyboard input
    cpi 0dh
    jnz inlup                 ;must be <cr> to continue
    call crlf
    call crlf

DWNLOD:
    LDA BUFFER                 ;read first byte of buffer
    ORA A                     ;zero says no
                                ;file name entered
    LXI D,CMDMSG              ;print 'downloading' message
    JZ EXIT                   ;unless no file was entered
    MVI C,19                  ;delete file of same name
                                ;(if any).

    LXI D,FCB
    PUSH D                    ;save d,e for later
    CALL BDOS                 ;do the delete
    POP D                     ;get d,e back
    MVI C,22                  ;create file of same name
    CALL BDOS                 ;now

```



```

INR A ;a=ff says no
;directory space.

LXI D,NDSMSG
JZ EXIT ;print
;'no directory space' if so.

RDYLP: ;else look for incoming 'r'
CALL RDCOM
CPI 'R' ;wait here for it
JNZ RDYLP ;got it, send 's' as answer.
RDY1: MVI E,'S' ;output it
CALL WRCOM

GETGEE:
CALL RDCOM
CPI 'R' ;expect 'g' in return
;(handshaking).
JZ RDY1 ;maybe we missed it,
;try again.
CPI 'G' ;not an 'r', maybe a 'g'
JNZ GETGEE ;false alarm, wait some more
LXI H,WRKMSG ;contact,print
;downloading message

PRLP: MOV A,M ;display same on screen
ORA A ;print until 0 occurs
JZ TRYAGN ;end of message,
;loop some more.
PUSH H ; save h,l
MOV E,A
CALL CONOUT ;send byte to screen
POP H
INX H ;get h,l back and update
JMP PRLP ;do it again,sam

TRYAGN:
LXI H,BUFFER ;h,l point to dma address
MVI C,0 ;initialize checksum
MVI D,81H ;byte count,
;including checksum.

LOOP1:
CALL RDCOM
MOV M,A ;get byte, send it to buffer
MOV E,A ;save a copy in e
XRA C ;update checksum
MOV C,A ;saving same in c
INX H
DCR D ;update registers
JNZ LOOP1 ;repeating until all
;129 bytes received.
ORA A ;last byte=checksum,0=good
JZ GOODRD

```



```

BADRD:
    MVI E, 'B'           ;it wasn't good
    CALL CONOUT          ;show bad on screen
    MVI E, 'B'
    CALL WRCOM           ;also tell the other end
    JMP TRYAGN          ;and try once more

GOODRD:
    MVI E, '.'          ;success
    CALL CONOUT          ;show success on screen
    LXI D,FCB
    MVI C,21             ;write sequential to disc
    CALL BDOS
    MVI E, 'G'
    CALL WRCOM           ;ship out 'g'=good read
    JMP TRYAGN          ;and do iy again

DONE: STA APPKBD+10H
    LXI D,FCB
    MVI C,16             ;that's it, close the file
    CALL BDOS           ;and update the directory
    LXI D,DONMSG

EXIT: CALL PRMSG        ;then print the
                        ;'done' message.
                        ;and exit to cpm

    JMP BOOT

WRCOM:
    LDA COMSTS           ;read status port
    ANI TXRDY           ;to see if
                        ;transmit is ready.
                        ;wait for it

    JZ WRCOM
    MOV A,E
    STA COMDAT          ;ship out data
                        ;from 'E' register.

    RET

RDCOM:
    LDA COMSTS           ;read status port
    RAR
    JC READIT
    LDA APPKBD
    CPI 83H
    JZ DONE
    JMP RDCOM           ;not ^c, loop back

READIT:
    LDA COMDAT
    RET                 ;read input on rs232 line

PRMSG:
    MVI C,9
    JMP BDOS           ;cpm function to
                        ;print string.

```



```

CONOUT:
    MVI C,2
    JMP BDOS                                ;cpm function call
                                           ;for console out.

CRLF:  mvi c,2
        mvi e,0dh
        call bdos
        mvi c,2
        mvi e,0ah
        jmp bdos

INMSG:
    db 'apple Download Program'
    db 0AH,0DH, 'Type DOWNLOAD file.nam'
    db 0dh,0ah, 'where file=desired file'
    db 0dh,0ah, 'and nam=file type'
    db 0dh,0ah, 0dh,0ah
    db 'when transfer is finished,type ^C=control c'
    db 0dh,0ah,0dh,0ah
    db 'Enter <cr> when ready to start$'

CMDMSG:
    DB 'COMMAND ERROR$'

NDSMSG:
    DB 'NO DIRECTORY SPACES$'

DONMSG:
    DB 13,10,'DOWNLOAD COMPLETE$'

WRKMSG:
    DB 'DOWNLOADING',0

```


APPENDIX C. FLOWCHARTS

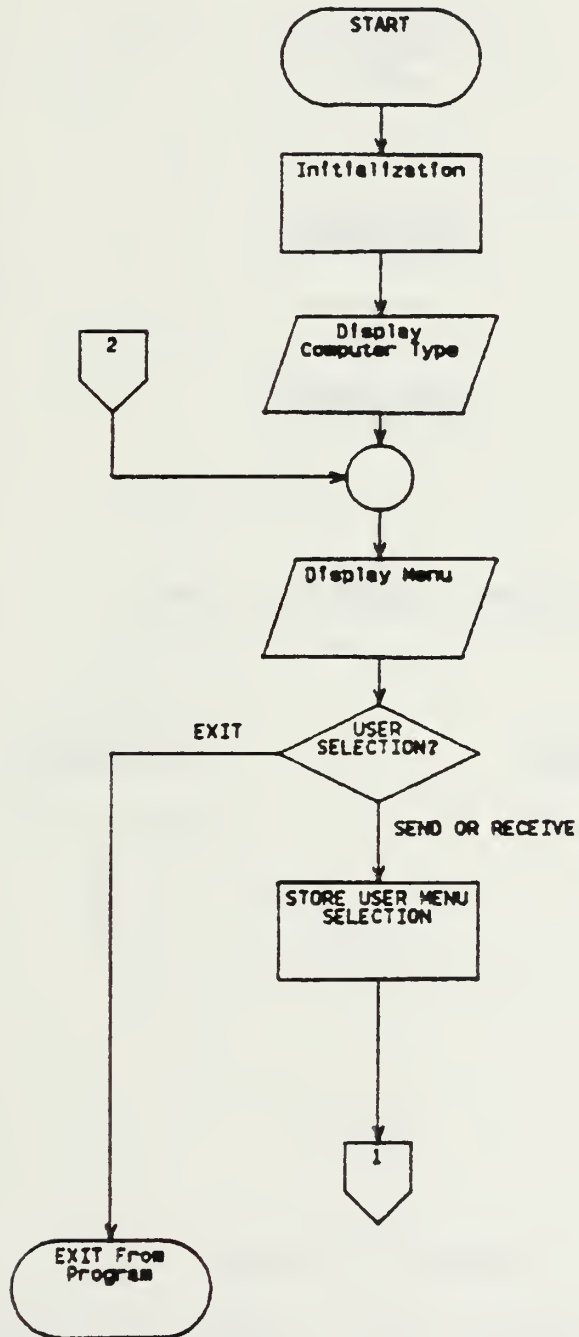
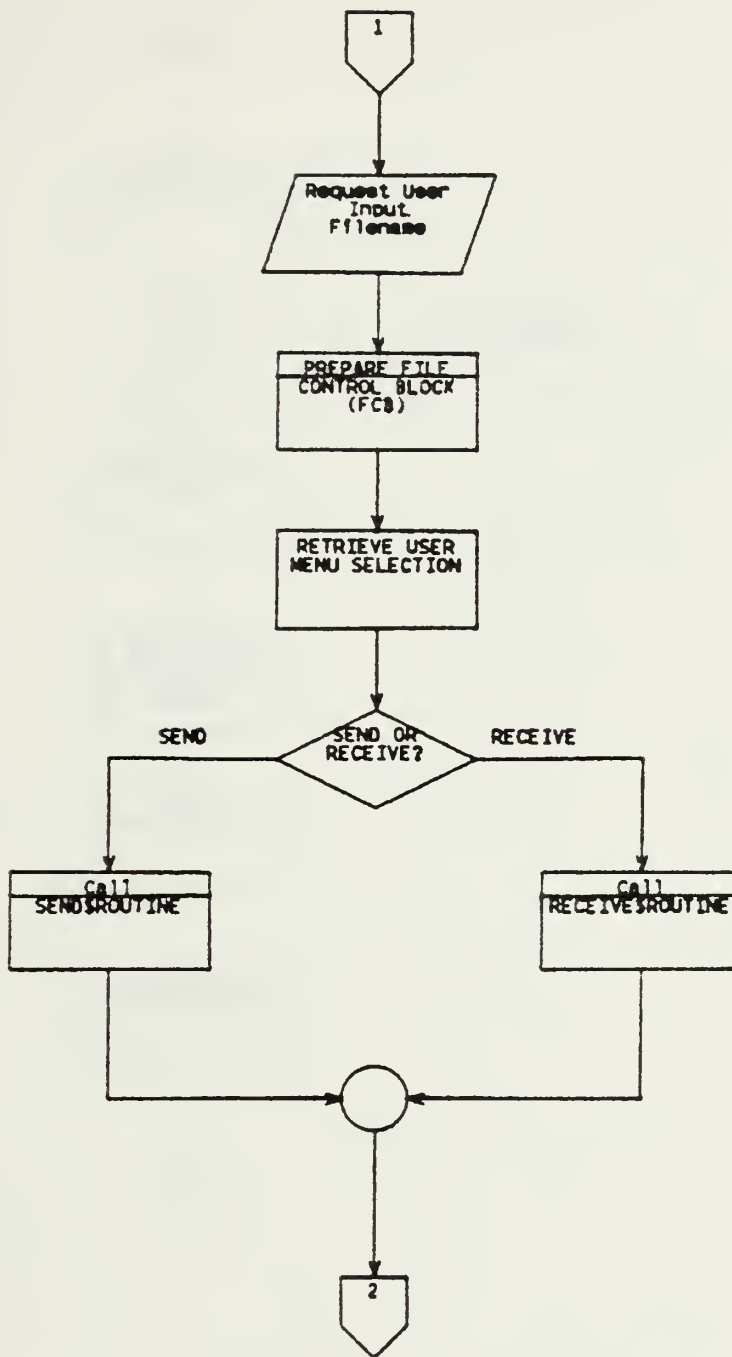


Figure C.1 CROSS.ASM Main Program



BRANCH BACK TO MENU DISPLAY

Figure C.2 CROSS.ASM Main Program (continued)

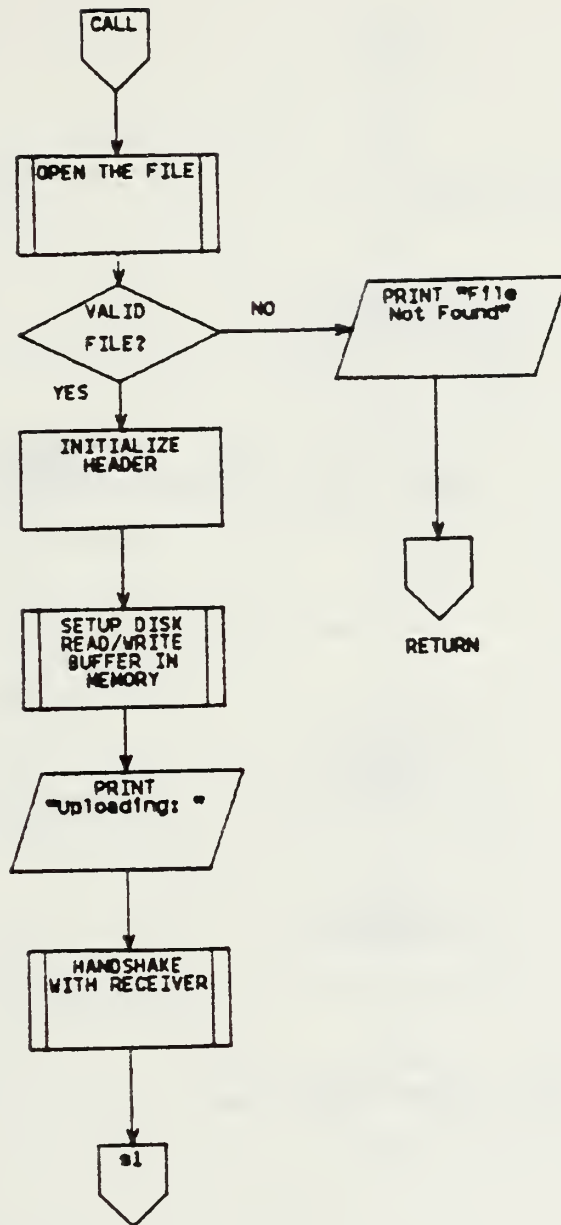


Figure C.3 CROSS.ASM SEND\$ROUTINE Program

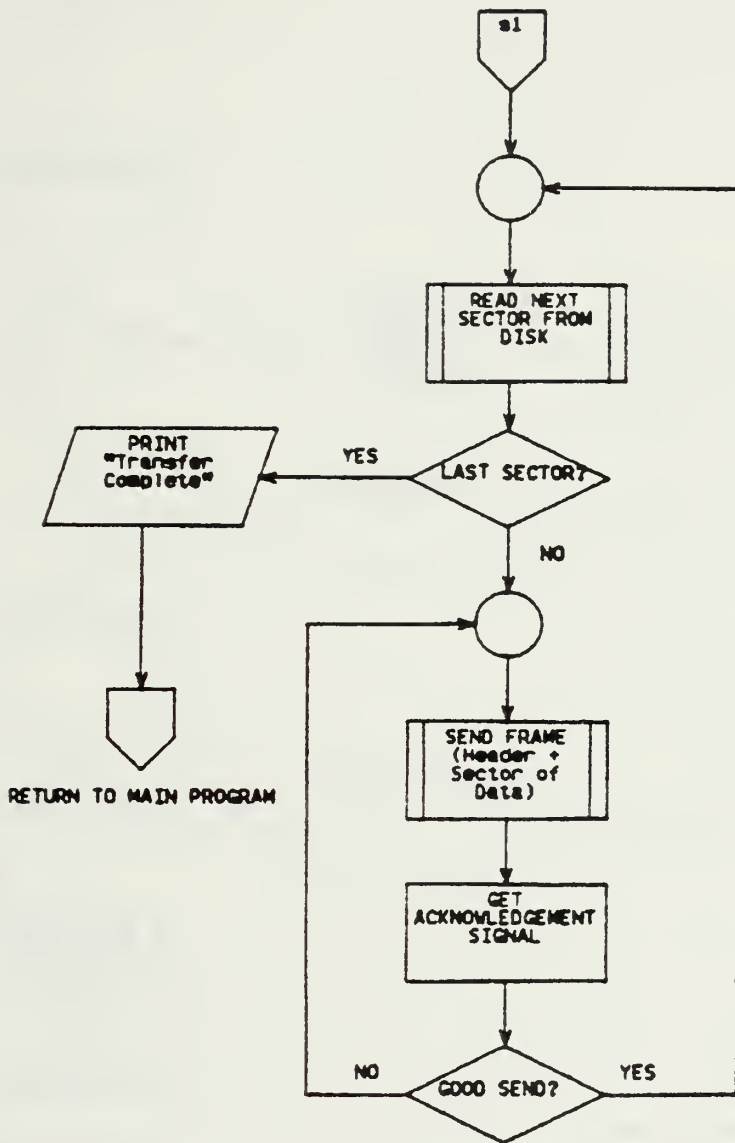


Figure C.4 SEND\$ROUTINE (continued)

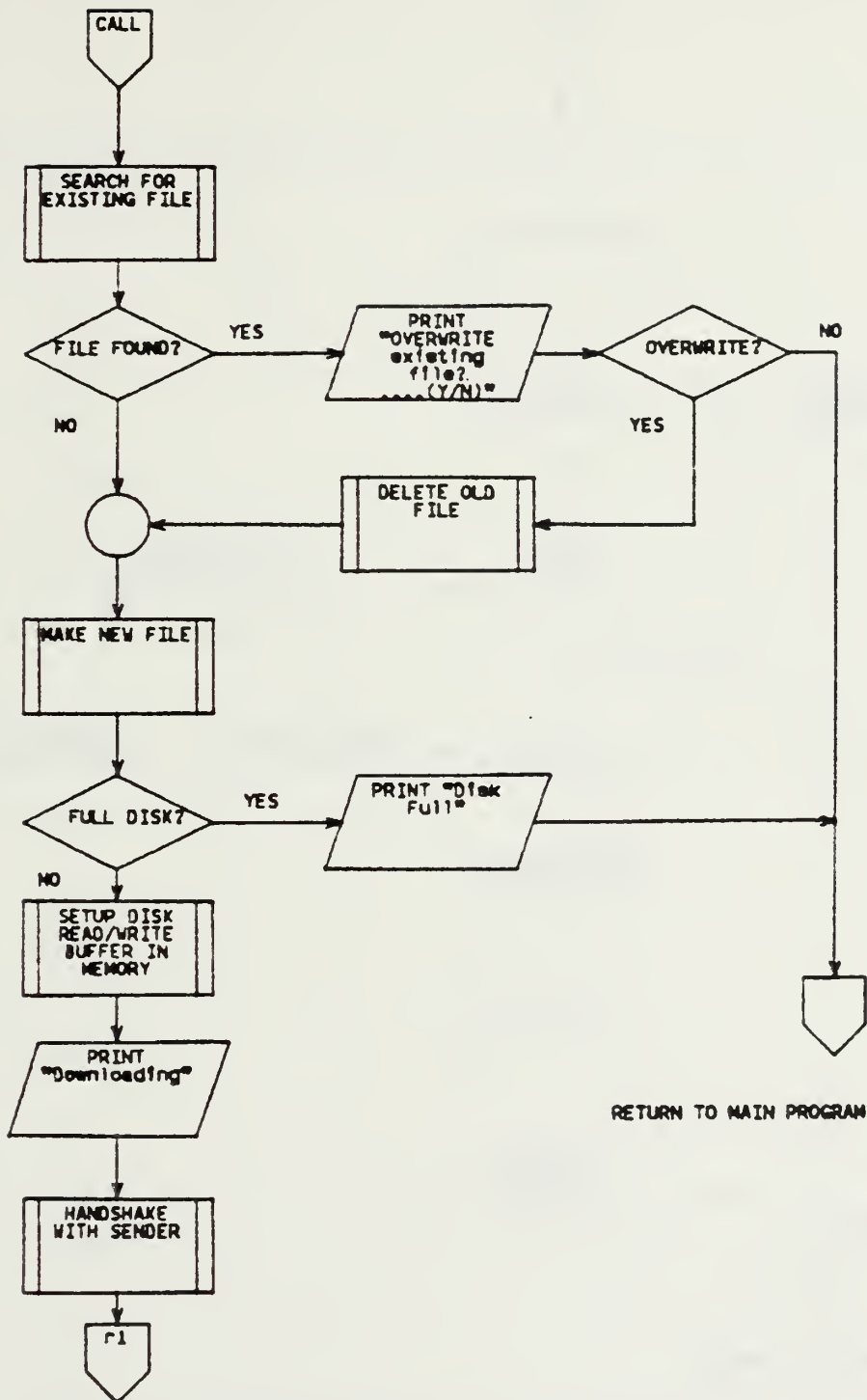


Figure C.5 CROSS.ASM RECEIVE\$ROUTINE Program

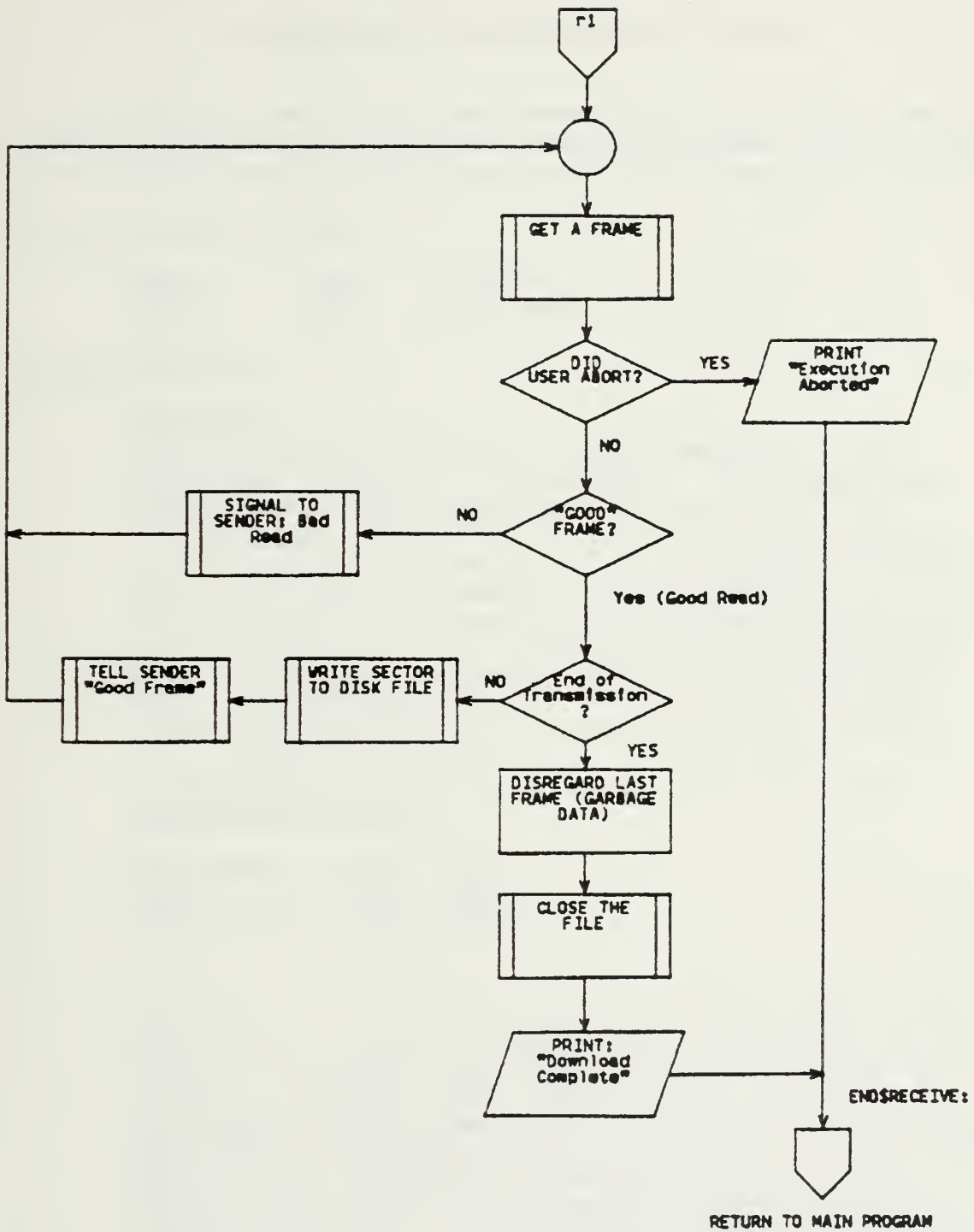


Figure C.6 RECEIVE\$ROUTINE (continued)

APPENDIX D: MAIN PROGRAM LISTING

```
;=====
; CROSS.ASM...crossloader for any CP/M based microcomputer
;=====
```

```
;Initial definitions:
```

```
TRUE      EQU      0FFFFH
FALSE     EQU      NOT TRUE
```

```
;Computer type:
```

```
;Example:
```

```
;If the program is for an Apple computer, the
;variable "APPLE" is set to TRUE. All others are
;set FALSE. This technique requires the file
; "APPLE.LIB" (an I/O library written specifically
;for an Apple computer) to be on the same disk as
;this one during assembly. Other computers can be
;added to this by creating I/O libraries for them
;in the same format as "APPLE.LIB" or "NSTAR.LIB" .
;The I/O libraries contain only macros that know
;how to perform specific I/O functions that are
;machine dependent (i.e. access the RS-232 port):
```

```
;This version of the program is set up for the
;NORTHSTAR HORIZON
```

```
NORTHSTAR EQU TRUE
APPLE     EQU FALSE
ANYCOMPUTER EQU FALSE
```

```
;The following flags are used by the macro
;libraries to avoid duplication of code when a
;macro is invoked more than once. Most macros used
;in the libraries call subroutines. During
;assembly when the macro is invoked the first time,
;the code is expanded, and its specific flag is set
;TRUE. On subsequent invocations, if the flag
;equals TRUE, most of the code is not expanded, but
;rather, calls the previously expanded subroutine.
;This technique exploits the advantages of having
;macros (ie. inline code and parameters) without all
;of the overhead of code duplication.
```


;Flags:

```
PRFLAG      SET  FALSE  ;
COFLAG      SET  FALSE  ;
OPFLAG      SET  FALSE  ;
RDFLAG      SET  FALSE  ;
OTFLAG      SET  FALSE  ;
MVFLAG      SET  FALSE  ;
MKFLAG      SET  FALSE  ;
WRFLAG      SET  FALSE  ;
UNFLAG      SET  FALSE  ;
DEFLAG      SET  FALSE  ;
CLFLAG      SET  FALSE  ;
DMFLAG      SET  FALSE  ;
RPFLAG      SET  FALSE  ;
CIFLAG      SET  FALSE  ;
RCFLAG      SET  FALSE  ;
RPKFLAG     SET  FALSE  ;
CL2FLAG     SET  FALSE  ;
SCHFLAG     SET  FALSE  ;
SYNCFLAG    SET  FALSE  ;
RDCOMFLAG   SET  FALSE  ;
SENDPFLAG   SET  FALSE  ;
```

;Invocation of the macro library follows:

```
IF          APPLE
MACLIB     APPLE
ENDIF
```

```
IF          NORTHSTAR
MACLIB     NSTAR
ENDIF
```

```
MACLIB     CPMMAC
```

```
;
;
;          *****
;          * LOCAL EQUATE STATEMENTS *
;          *****
```

```
BOOT        EQU  0      ;Warm boot address
FCB         EQU  5CH    ;Default file control block
BUFFER      EQU  80H    ;Default buffer address.
BELL        EQU  07H    ;ASCII for ring the bell.
DATA$LENGTH EQU  128D   ;Length of data field.
STOP$CODE   EQU  0FFH   ;"End of Transmission".
SEND$CODE   EQU  0FEH   ;Code for "SEND" mode.
RECEIVE$CODE EQU  0EFH  ;Code for "RECEIVE" mode.
```



```
;*****  
;*   M A I N       P R O G R A M *  
;*****
```

```
;Start the program at address 0100 hex:
```

```
ORG      0100H
```

```
;  
;Initialization  
;
```

```
;Move the stack pointer to the end of the program:
```

```
LXI      SP,END$OF$PROGRAM
```

```
;Initialize the computer for I/O operations  
;with "INITIALIZE" macro:
```

```
INITIALIZE
```

```
;  
;Display Computer Type  
;
```

```
;Tell the user which computer this program  
;was designed for:
```

```
PRINT <ESC,CLEAR,LF,LF,LF,CR>  
PRINT <'      '>  
PRINT <'CROSSLOADER for the '>  
COMPTYPE      ;Macro that prints computer name  
PRINT < CR,LF,LF,LF,LF,LF,LF>
```

```
;  
;Display Menu  
;
```

```
DISPLAY$MENU:
```

```
PRINT <'      '>  
PRINT <'SELECTION MENU:  ',LF,CR,CR>  
PRINT <'      '>  
PRINT <'S .... Send a file.',LF,CR>  
PRINT <'      '>  
PRINT <'R .... Receive a file.',LF,CR>  
PRINT <'      '>  
PRINT <'E .... Exit.',LF,LF,LF,CR>
```



```
PRINT <'What is your choice?..... '>
;Read char from console and convert to uppercase:
```

```
READCH
```

```
;
;Store User Menu Selection
;
```

```
;Check response to see if it is 'E', 'S', or 'R':
```

```
CHECK$E:
```

```
CPI      'E'
JZ       EXIT
```

```
CHECK$S:
```

```
CPI      'S'
JNZ     CHECK$R
```

```
;Inform user he has entered the 'SEND' mode,
;set the MODE$FIELD to SEND, and jump to next
;routine:
```

```
PRINT    <ESC,CLEAR,CR,'SEND:',CR,LF>
MVI      A,SEND$CODE
STA      MODE$FIELD
JMP      DETERMINE$FILE
```

```
CHECK$R:
```

```
;Inform user he has entered the 'RECEIVE' mode,
;set the MODE$FIELD to RECEIVE, and jump to next
;routine:
```

```
CPI      'R'
JNZ     COMMAND$ERR
PRINT    <ESC,CLEAR,CR,'RECEIVE:',CR,LF>
MVI      A,RECEIVE$CODE
STA      MODE$FIELD
JMP      DETERMINE$FILE
```

```
COMMAND$ERR:
```

```
;Inform the user he has made an error and return
;him to the menu:
```



```
PRINT <ESC,CLEAR,CR,'Command error: '>
PRINT <'try again....',LF,LF,LF,LF,CR>
JMP DISPLAYMENU
```

```
;  
; Request User Input Filename  
;
```

```
DETERMINE$FILE:
```

```
;Find out what file he wants to transfer:  
  
PRINT 'What file do you wish to transfer? .....

```
;Read in a string from the console with READB macro:

READB

;Determine if any entries were typed in.
;The global variable RBUFF$COUNT contains the
;number of characters that were typed in
;(RBUFF$COUNT is contained in READB macro).

LDA RBUFF$COUNT ;Find out how many chars.
CPI 0H ;Check for zero.
JNZ GOOD$ENTRY ;If RBUFF$COUNT not 0, jump.
```


```

```
BAD$ENTRY:
```

```
;Tell user he didn't enter anything:  
  
PRINT <BELL,ESC,CLEAR,CR>  
PRINT <'ERROR: Nothing entered ',CR,LF>  
PRINT <' press RETURN to continue .... '>  
READCH  
PRINT <ESC,CLEAR>  
JMP DETERMINE$FILE ;Go ask again.
```

```
;  
; Prepare File Control Block (FCB)  
;
```

```
GOOD$ENTRY:
```

```
;Take the filename from the console buffer that  
;was created by READB macro,and put it into the  
;File Control Block (FCB) for CP/M operations to  
;be performed later:  
  
CALL PREPARE$FCB
```



```
;  
; Retrieve User Menu Selection  
;
```

```
    ;Retrieve and check the mode-type:
```

```
LDA     MODE$FIELD    ;Recall mode semaphore.  
CPI     RECEIVE$CODE ;If RECEIVE was specified,  
JZ      RECEIVE$MODE ;jump to RECEIVE$MODE,  
                        ;otherwise to SEND$MODE.
```

```
SEND$MODE:
```

```
    ;Branch to the send routine:
```

```
CALL    SEND$ROUTINE  
JMP     DISPLAY$MENU
```

```
RECEIVE$MODE:
```

```
    ;Branch to the receive routine:
```

```
CALL    RECEIVE$ROUTINE  
JMP     DISPLAY$MENU
```

Note: At this point, execution branches back up to PRINT\$MENU. If the user selects "E", then execution jumps to the following label (EXIT):

```
EXIT:                                     ;Universal program exit.
```

```
PRINT <ESC,CLEAR>           ;Clear the screen  
JMP  BOOT                   ;and exit gracefully.  
;End of main program.
```



```
;  
;  
; *****  
; * S U B R O U T I N E S *  
; *****  
  
;  
PREPARE$FCB: ;Subroutine to fill in File Control Block  
; (FCB) from a console buffer created by  
; READB macro.  
;
```

```
;Initialize filename with 14 ASCII zeroes.
```

```
MOVE      '           ',FILE$NAME
```

```
;Prepare the FCB to receive the filename.
```

```
MOVE      0,FCB+32D      ;Zero FCB+32 (reqd. by CPM).  
MOVE      <0,'          ',0>,FCB  
;Put '0h' in FCB+0 and FCB+12  
;Put '20h' in FCB+1 thru FCB+11  
;(required by CP/M for disk  
;operations.)
```

```
LXI      H,FCB+1        ;Point to first filename  
;location in the FCB.
```

```
LXI      D,FILE$NAME    ;Also point to the block that  
;will hold the file name for  
;screen display.
```

```
MOVCHARLOOP:
```

```
CALL     GETCH          ;A global subroutine contained  
;in READB macro that returns  
;one char from the console  
;buffer each time it is called.
```


;When GETCH has returned all characters in the
;buffer, it set the CARRY flag. When CARRY is
;set, we want to exit this loop:

```
JC      ENDFILLFCB    ;Carry flag set if no more  
                        ;chars to read, ;so return.  
CPI     '.'           ;Check to see if char is  
                        ;decimal point. If so, next  
                        ;char is part of "filetype",  
JZ      DECIMALPT    ;so we branch.
```

NOTDECIMALPT:

;Convert char in A-reg to uppercase.

UCASE

;Put file name in FCB+1 thru FCB+8
;This is the "Filename" field of the FCB.

```
MOV     M,A  
STAX   D      ;Also update the variable FILE$NAME.  
INX    D      ;Move pointers to next space.  
INX    H  
JMP    MOVCHARLOOP
```

DECIMALPT:

;Decimal point found so we skip ahead to FCB+9
;and fill in the "Filetype" field of the FCB:

```
LXI    H,FCB+9D    ;Point to "Filetype" field  
                        ;of the FCB.  
STAX   D  
INX    D  
JMP    MOVCHARLOOP ;Get next character.
```

;The only RETURN for this subroutine:

ENDFILLFCB:

RET


```
;  
SEND$ROUTINE: ;Subroutine to conduct a "SEND" operation.  
;
```

```
;  
; Open The File  
;
```

```
;Invoke OPEN macro from CPMMAC.LIB. OPEN consults  
;the disk directory and identifies all sectors  
;containing data belonging to the file whose name  
;is in the FCB. If the file is not found, it  
;branches to the label appearing in its second  
;parameter (NOT$FOUND is the label in this case):
```

```
OPEN FCB,NOT$FOUND  
JMP OPEN$OK ;File found so jump past this  
;next part.
```

```
;  
; Print "File Not Found"  
;
```

```
NOT$FOUND:
```

```
PRINT <ESC,CLEAR,'ERROR: your file '>  
PRINT FILE$NAME,14D  
PRINT <' cannot be found ... Try again',CR,LF,BELL>  
JMP SEND$END
```

```
;  
; Initialize Header  
;
```

```
OPEN$OK:
```

```
;The file is good so we initialize any fields in  
;header that require it:
```

```
XRA A  
STA FRAME$NUM  
STA FRAME$NUM+1  
STA STOP$FIELD
```



```

;
; Setup Disk READ/WRITE Buffer In Memory
;

;Invoke SETDMA macro. Tell CP/M that for all
;subsequent "disk reads", read data from the disk
;and place it in memory at the location specified
;by the address "DATA$FIELD":

SETDMA DATA$FIELD

;
; PRINT "Uploading:"
;

PRINT <ESC,CLEAR>
PRINT <'Uploading: '>
PRINT FILE$NAME,14D ;Print the file name to the CRT.
PRINT <' '>

;
; Handshake With Receiver
;

;Invoke HANDSHAKE macro. This macro synchronizes
;the two computers. The parameter TRANSMIT or
;RECEIVE tells the macro which mode to use:

HANDSHAKE TRANSMIT ;(SEND or "TRANSMIT" mode)

;
; Read Next Sector From Disk
;

READ$NEXT$SECTOR:

;Invoke READSECTOR macro to get a sector of data
;from the file opened by OPEN macro. Store it in
;memory at the location set by SETDMA macro. If
;the operation is good, return a zero in the
;A-register and display a '+' on the screen.
;If not successful, return a non-zero value in the
;A-register:

READSECTOR FCB,'+'

```



```

;
; Last Sector?
;

        JNZ   ENDOFSEND           ;If A-reg not equal to 0,
                                   ;all sectors of the file have
                                   ;been read, so, we're done.

;
; Send Frame (Header and Sector of Data)
;

SEND$IT:

        ;Prepare to send the frame of data starting at label
        ;"START$OF$FRAME" by invoking the SENDFRAME macro.
        ;SENDFRAME expects to find the number of bytes to
        ;send in the A-register.   The variable SIZE$FRAME
        ;has this value:

        LDA   SIZE$FRAME

        ;Transmit the frame located at the address
        ;"START$OF$FRAME".

        SENDFRAME START$OF$FRAME

;
; Get Acknowledgement Signal
;

VERIFYTRANSMIT:

        ;Evaluate the return from the receiving computer.
        ;If it is a 'B' (BAD), resend the frame.   If it is
        ;a 'G' (GOOD), go get the next sector in sequence.
        ;READ$PORT macro checks the RS-232 port until it
        ;receives a character, then transfers it to the
        ;A-register:

        READ$PORT           ;Read the returned character
                               ;from port.

        CPI   'B'           ;Bad...resend.
        JZ    SEND$IT

        CPI   'G'           ;Good...get another if it exists.
        JZ    READ$NEXT$SECTOR
        JMP   VERIFYTRANSMIT ;Ignore everything else.

```


ENDOFSEND:

;Transfer here when all sectors of the file have
;been sent. Put a code in the header of the next
;frame telling the receiving computer we're all
;done and its OK to close the file:

MVI A,STOP\$CODE ;End of file (EOF) signal.
STA STOP\$FIELD ;Mark frame header with EOF.

;Then send the frame:

LDA SIZE\$FRAME ;Load A-reg with frame size
SENDFRAME START\$OF\$FRAME ;Send EOF to receiver.

;Tell the user the transfer is complete:

PRINT <CR,LF,LF,'UPLOAD COMPLETE',BELL>

SEND\$END:

PRINT <' press RETURN to continue '>
READCH
PRINT <ESC,CLEAR>
RET


```

;
; RECEIVE$ROUTINE: ;Subroutine for receiving a file.
;
;
; Search for Existing File
;
;Invoke SEARCH macro. SEARCH looks in the disk
;directory for the filename specified in the FCB.
;If found, it continues; if not found, it branches
;to the label specified by its specified in the FCB.
;If found, it continues; if not found, it branches
;to the label specified by its specified in the FCB.
;If found, it continues; if not found, it branches
;to the label specified by its second parameter
;("NO$MATCH" in this case):

SEARCH FCB,NO$MATCH

;
; PRINT "OVERWRITE existing file? ...(Y/N)"
;
;The file to be received already exists and would
;be overwritten if the transfer occurs. Notify the
;user:

PRINT <'OVERWRITE existing file? .....(Y/N) '>

READCH ;Read a char from the console with
;READCH macro.

CPI 'Y'
JNZ DISPLAY$MENU ;User wants to save the old
;file. Skip it and start over

;
; Delete Old File
;
;Invoke DELETE macro to delete the old file
;(a prerequisite of CP/M). Must be performed
;before one can write the new file to disk. The
;file to delete is indicated by the filename in the
;FCB. Then continue as if not matching filename
;was found:

DELETE FCB

```


NO\$MATCH:

```
;  
; Make New File  
;
```

```
;Once no other file by that name exists we can set  
;up to write the new file to disk. This is done  
;with MAKE macro. If the new file is successfully  
;created, execution jumps to the label specified by  
;the macro's second parameter ("DISK$NOT$FULL" in  
;this case). If the disk is full, the user is  
;notified and execution branches back to the main  
;menu:
```

```
MAKE FCB, DISK$NOT$FULL
```

```
;  
; PRINT "Disk Full"  
;
```

```
;Disk full, could not make new file:
```

```
PRINT <ESC,CLEAR>  
PRINT <'ERROR: Disk full .... ',BELL>  
JMP END$RECEIVE
```

DISK\$NOT\$FULL:

```
;Initialize the frame header for transfer:
```

```
XRA A  
STA FRAME$NUM  
STA FRAME$NUM+1  
STA STOP$FIELD
```

```
;  
; Setup Disk READ/WRITE Buffer in Memory  
;
```

```
;Invoke SETDMA macro. Tell CP/M that for all  
;subsequent "disk writes", find the data in memory  
;at the location specified by the address  
;"DATA$FIELD" and write it onto disk:
```

```
SETDMA DATA$FIELD
```



```

;
; PRINT "Downloading"
;
;Notify the user the transfer has begun:
PRINT <ESC,CLEAR> ;Clear the screen.
PRINT 'Downloading: '
PRINT FILE$NAME,14D
PRINT <' '>

;
; Handshake With Sender
;

HANDSHAKE RECEIVE ;(RECEIVE mode)

;
; Get a Frame
;

GET$A$FRAME:

;Set up to receive the inbound frame. Put the
;frame size in the D-register as required by
;GETFRAME macro:

LDA SIZE$FRAME ;Lookup the size of the frame
; (including the checksum).
MOV D,A ;Store the size in D-reg for
;GET$FRAME macro.

;Invoke GETFRAME macro. It reads in the inbound
;frame a character at a time from the RS-232 port.
;The frame is stored in memory starting at location
;"START$OF$FRAME". It also returns the following
;codes in the A and B registers:

; B=0, A=0: valid data
; B=0, A not 0: invalid data. Request "retransmit".
; B=0FFh : user on receiving computer aborted
; the operation with control-C

GETFRAME START$OF$FRAME

;We now test the A and B registers:

PUSH PSW ;Save contents of A-register.
MOV A,B
CPI 0FFh
JNZ CHECKA ;If not aborted by user, branch and
;look at A-reg.

```



```

;
; PRINT "Execution Aborted"
;

;Confirm for the user that he aborted the transfer:
PRINT <ESC,CLEAR,BELL>
PRINT <'Program ABORTED by user ... '>
JMP      END$RECEIVE ;Return to the main program.

CHECKA:
;Not aborted by user. Check to see if the
;A-register is zero (meaning the frame was not
;damaged in transfer):

POP      PSW                ;Get contents of A-register
                        ;back.

;
; "Good" Frame?
;
JZ       GOODREAD          ;A=0: good frame transfer.

;
; Signal to Sender: Bad Read
;

BADREAD:
;Invoke the OUTPUT macro to send the 'B' character
;to the RS-232 port and out onto the bus. Also,
;display the letter 'B' on the console screen:

OUTPUT 'B',,'B'           ;Send 'B' back to sender,
JMP      GET$A$FRAME      ;and try once more.

GOODREAD:
;Branch here if a good frame was received. Check
;STOP$FIELD in the header to see if the sending
;computer is done transferring:

LDA      STOP$FIELD
CPI      0FFH              ;0FFH means "all done".

;
; End of Transmission?
;
JZ       CLOSE$THE$FILE

```



```

;
; Write Sector to Disk File
;

;Invoke WRITESECTOR macro to read the data from the
;frame (disregard the header and checksum).
;Perform a "disk write" to the file on disk. For
;each good write, display a '+' on the screen:

WRITESECTOR FCB, '+'

;
; Tell Sender "Good Frame"
;

;Once the data is safely on disk, signal the sender
;to transmit another frame by invoking OUTPUT macro
;to send the character 'G' to the sender to confirm
;a "good send":

OUTPUT 'G' ;Send 'g' = "good send".
JMP GET$A$FRAME ;set up to receive another.

;
; Close the File
;

CLOSE$THE$FILE:
;Branch here at the end of transfer and invoke
;CLOSEFILE macro to update the disk directory:

CLOSEFILE FCB

;
; PRINT "Download Complete"
;

;Tell the user:
PRINT <ESC,CLEAR,'DOWNLOAD COMPLETE '>

END$RECEIVE:

PRINT <'press RETURN to continue '>
READCH ;macro to read one char from console.
PRINT <ESC,CLEAR>

;The only exit from this subroutine:

RET

```



```

; *****
; * B U F F E R   A R E A *
; *****

MODE$FIELD:      DB   00H           ;Either "SEND" or "RECEIVE".
FILE$NAME:       DB   '           ' ;Holds the file name.
SIZE$FRAME:      DB   END$OF$FRAME-START$OF$FRAME ;Frame size calculation.

;+++++ FRAME BOUNDARY ++++++
START$OF$FRAME:  ;This label is required to designate the beginning
                  ;of the frame to be sent. It is used by the
                  ;assembler to calculate frame size ... therefore,
                  ;it must be present and must precede the frame.

START$OF$HEADER: DB   01H           ;01h is Start of
                  ;Header (SOH).
FRAME$NUM:       DW   0000H         ;Frame number.
DESTINATION$ADDR: DB   00H         ;not currently used.
SOURCE$ADDR:     DB   00H         ;Not currently used.
OP$CODE:         DB   00H         ;Not currently used.
STOP$FIELD:      DB   00H         ;Normally 00h.
                  ;0FFh = end of file.

DATA$FIELD$LENGTH: DB DATA$LENGTH
                  ;Allows for variable
                  ;length data fields of
                  ;up to size 0FFh.
SPARE$BLOCK:     DB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                  ;Fields for expansion.
;..... WRITE SECTOR IN FOLLOWING DATA FIELD .....
DATA$FIELD:      DS DATA$LENGTH
                  ;Data is read from
                  ;disk to here and
                  ;vice-versa.
;.....
CHECK$SUM:       DB   00H

END$OF$FRAME:    ;A label used for SIZE$FRAME
                  ;computation. Must immediately
                  ;follow CHECK$SUM.
;+++++ FRAME BOUNDARY ++++++

EXPANSION:       DB   80H           ;Overflow area.
                  DS   36D         ;Stack pointer storage.
END$OF$PROGRAM: ;Label used by stack.
                END   0100H

```


APPENDIX E: MAIN MACRO LIBRARY

```

;=====
;CPM MACRO LIBRARY
;=====

```

;List of macros contained:

NAME	PARAMETERS	FLAGS
; sysf	macro func,ae	<none>
; pchar	macro par	COFLAG
; print	macro text, bytes	PRFLAG
; search	macro ptr,where	SCHFLAG
; open	macro ptr,where	SCHFLAG
; readsector	macro ptr,char	RDFLAG
; writesector	macro ptr,star	WRFLAG
; move	macro from,to,bytes	MVFLAG
; make	macro ptr	MKFLAG
; delete	macro ptr,where	DEFLAG
; closefile	macro ptr	CLFLAG
; handshake	macro transmit?	SYNCFLAG
; getframe	macro where,size	RECPFLAG
; sendframe	macro start,size	SENDPFLAG
; setdma	macro ptr	DMFLAG
; ucase	macro reg	<none>
; readb	macro	RCFLAG
; readch	macro noupper?, reg	CIFLAG

```

;-----
; EQUATE STATEMENTS
;-----

```

;CPM equates:

```

BDOS EQU 0005H ;bdos address

```

;NETWORK CONTROL EQU'S:

;MACRO EQU'S:

```

TRANSMIT EQU TRUE ;used by HANDSHAKE macro.
RECEIVE EQU FALSE;used by HANDSHAKE macro.

```

;Z80 EQU'S:


```

;*****
SYSF      MACRO   FUNC, AE
;*****
    ;;Description:
    ;;      A macro to generate BDOS calls.  FUNC is the
    ;; BDOS function number that is put in C-reg.
    ;; The contents of A are stored in E if there is a
    ;; second parameter.  NOTE: not an 'inline macro'.

    ;;Registers saved:  B,D,H
    ;;Flags used:      none
    ;;Usage:           OPEN:  SYSF 15
    ;;                  PCHAR: SYSF 2,AE

    PUSH H ! PUSH D ! PUSH B
    MVI     C,FUNC

    IF NOT NUL AE
    MOV     E,A
    PUSH   PSW
    CALL   BDOS
    POP    PSW

    ELSE

    CALL   BDOS
    ENDIF

    POP B ! POP D ! POP H
    RET
    ENDM

;*****
PCHAR     MACRO   PAR
;*****
    LOCAL  AROUND

    ;;Description:
    ;;      An inline macro to print on character to
    ;; the console.  PAR, if present, is loaded into
    ;; the A-reg.  Registers saved:  none
    ;; Flags used:  COFLAG
    ;; Usage:      PCHAR
    ;;            PCHAR '*'
    ;
    IF NOT NUL PAR
    MVI     A,PAR
    ENDIF

```



```

CALL      PCH2?

IF NOT COFLAG
JMP      AROUND

PCH2?:
SYSF      2,AE

COFLAG    SET      TRUE
          ENDIF

AROUND:

ENDM

```

```

;*****
PRINT     MACRO   TEXT, BYTES, REG
;*****
          LOCAL   AROUND,MSG

          ;;Description:
          ;;      An inline macro to print a string on the
          ;; CRT. TEXT is the address of the string, BYTES
          ;; is the length. TEXT may be in quotes if
          ;; BYTES is omitted.
          ;;      If the third parameter, REG, is not nul,
          ;; then the number of chars to print is expected
          ;; in B-reg. In this case, the second parameter
          ;; must not be nul, but its value does not matter.
          ;;
          ;;Registers saved: HL,BC
          ;;Flags used: PRFLAG
          ;;Macros needed: PCHAR
          ;;Usage: PRINT FCB+1,1
          ;; PRINT 'end of file'
          ;; PRINT <CR,LF,'message'>
          ;; PRINT ,12
          ;; PRINT TEXT,1,1 ;Prints text
          ;; ;located at
          ;; ;TEXT addr.
          ;; ;The number of
          ;; ;bytes is ex-
          ;; ;pected in B-
          ;; ;reg.

PUSH H ! PUSH B

IF      NUL BYTES
        LXI      H,MSG
        MVI      B,AROUND-MSG
ELSE

```



```

                IF      NOT NUL TEXT
                LXI    H,TEXT
                ENDIF

                IF NUL REG
                MVI    B,BYTES
                ENDIF
ENDIF

CALL    PBUF?

POP B ! POP H

IF NOT PRFLAG OR NUL BYTES
JMP     AROUND
ENDIF

IF NOT PRFLAG

PBUF?:
    MOV     A,M
    PCHAR
    INX    H
    DCR    B
    JNZ    PBUF?
    RET

PRFLAG    SET     TRUE
          ENDIF

          IF  NUL BYTES

MSG:
    DB     TEXT
    ENDIF

AROUND:                                     ;; PRINT
          ENDM
;*****
OPEN     MACRO  POINTR,WHERE
;*****
          LOCAL AROUND

          ;;Description:      An inline macro to open
          ;;                  an existing disk file.  POINTER refers to
          ;;                  the file control block (FCB).  Extent and
          ;;                  current record number are zeroed.  Branch to
          ;;                  location WHERE if file not found, or print
          ;;                  error message and branch to DONE otherwise.

```



```

;;Registers saved:  none
;;Flags used:      OPFLAG
;;Macros used:    SYSF, ERRORM
;;Useage:        OPEN  FCB
;;                OPEN  FCB,BOOT

LXI      D,POINTR      ;zero
XRA      A
STA      POINTR+12     ;extent
STA      POINTR+32     ;current record
CALL     OPEN2?
INR      A              ;0=ok,FF means error.
JNZ      AROUND

IF       NUL WHERE
ERRORM  'No source file. ',AROUND;0FFh in A-reg
        in unable.

ELSE
JMP     WHERE
ENDIF

IF       NOT OPFLAG

OPEN2?:  SYSF      15          ; Open the disk file.
OPFLAG  SET       TRUE       ; Only one copy.
        ENDIF

AROUND:  ;;OPEN
        ENDM
;*****
SEARCH   MACRO   POINTR,WHERE
;*****
        LOCAL  AROUND

;;Description:      An inline macro to search
;;                  an existing disk file.  POINTER refers to
;;                  the file control block (FCB).  Extent and
;;                  current record number are zeroed.  Branch to
;;                  location WHERE if file not found, or print
;;                  error message and branch to DONE otherwise.

;;Registers saved:  none
;;Flags used:      SCHFLAG
;;Macros used:    SYSF, ERRORM
;;Useage:        SEARCH  FCB
;;                SEARCH  FCB,BOOT

LXI      D,POINTR      ;zero
XRA      A

```



```

STA     POINTR+12      ;extent
STA     POINTR+32      ;current record
CALL    SEARCH2?
INR     A               ;0=ok,FF means error.
JNZ     AROUND

IF      NUL WHERE
ERRORM 'No source file. ',AROUND;0FFh in A-reg
if unable

ELSE
JMP     WHERE
ENDIF

IF      NOT SCHFLAG
SEARCH2?:
SYSF    17              ; SEARCH the disk file.
SCHFLAG SET    TRUE      ; Only one copy.
ENDIF

AROUND:                               ;;SEARCH
ENDM

;*****
ERRORM  MACRO  TEXT,WHERE
;*****
;;Description:      Macro to print error
;;      message on the CRT. Message must be enclosed
;;      in apostrophes. Optional second parameter
;;      has branch address. If no second parameter,
;;      go to BOOT.

;;Registers saved:  none
;;Flags used:      none
;;Macros used:      PRINT
;;Useage:          ERRORM 'message'

PRINT  <TEXT>

IF      NUL WHERE
JMP     BOOT

ELSE
JMP     WHERE
ENDIF

ENDM

```



```

;*****
READSECTOR      MACRO    POINTR, CHAR
;*****

```

```

LOCAL    AROUND

```

```

;;Description:      Inline macro to read a disk
;;      sector. POINTR refers to the file control
;;      block (FCB). Optional second parameter is
;;      symbol to be printed after sector is read.
;;      Zero flag is reset if end of file.

```

```

;;Registers saved: none
;;Flags used:      RDFLAG
;;Macros used:    SYSF, PCHAR
;;Useage:        READSECTOR  FCB1
;;              READSECTOR  FCBS, '*'

```

```

IF NOT NUL CHAR
PCHAR    CHAR          ;;to console.
ENDIF

```

```

IF NOT NUL POINTR
LXI     D,POINTR
ENDIF

```

```

CALL    READ2?
ORA     A          ;; set flags.
IF NOT RDFLAG
JMP     AROUND

```

```

READ2?:
SYSF    20          ;; Read disk sector.
RDFLAG  SET        TRUE      ;; Only one copy.
ENDIF

```

```

AROUND:          ;;READSECTOR
ENDM

```

```

;*****
MOVE     MACRO    FROM, TO, BYTES
;*****

```

```

LOCAL    AROUND, MESH

```

```

;;Description:      An inline macro to move text.
;;      If parameter 1 is a literal and parameter 3
;;      is nul, it puts a literal string in the
;;      location beginning at the TO address.
;;      If parameter 3 is included,
;;      it expects to have FROM=the "source address"

```



```

;;      (or find the source address in HL-reg if
;;      parameter 1 is nul), and to have TO = the
;;      "destination address" (or find the destina-
;;      tion address in DE-reg if parameter 2 is
;;      null).

```

```

;;Registers saved: HL,BC,DE
;;Flags used:      MVFLAG
;;Macros used:   none
;;Useage:
MOVE   'Input Text',FCB
;;
MOVE   0200H,BUFFER,36d
;;
MOVE   <02h,53h,'Text'>,FCB
;;
MOVE   FROM,,20d
;;
MOVE   ,,20d

```

```

PUSH H ! PUSH D ! PUSH B

```

```

IF NOT NUL TO
LXI   D,TO
ENDIF

```

```

IF NUL BYTES      ;;String move.
LXI   H,MESG     ;;Test.
LXI   B,AROUND-MESG

```

```

ELSE      ;;Not string move.

```

```

IF NOT NUL FROM
LXI   H,FROM
ENDIF

```

```

LXI   B,BYTES
ENDIF      ;;String/not string.

```

```

CALL   MOVE2?

```

```

POP B ! POP D ! POP H

```

```

IF NOT MVFLAG OR NUL BYTES
JMP   AROUND
ENDIF

```

```

;

```

```

MOVE2?:
IF NOT MVFLAG

```

```

MOV   A,M      ;;Get byte.
STAX  D        ;;New place.
INX   H        ;;From.
INX   D        ;;To.
DCX   B        ;;Byte count.
MOV   A,C

```



```

ORA      B
JNZ      MOVE2?      ;;Not done.
RET

;
MVFLAG   SET        TRUE      ;;One copy.
ENDIF    ;;not MVFLAG.

IF NUL BYTES
MMSG:
DB       FROM        ;;Text.
ENDIF
AROUND:  ;;Move.
ENDM

```

```

;*****
MAKE     MACRO    POINTR, WHERE
;*****

```

```

LOCAL   AROUND, ERROR
;;Description:      Inline macro to create a
;;                new disk file. POINTR refers to the file
;;                control block address. Extent and record
;;                number are zeroed. WHERE is an optional
;;                parameter that, if not nul, designates
;;                the address to branch to if CP/M can
;;                make the new file. If it cannot make the
;;                file (in case of "full disk"), execution
;;                continues after the macro.

```

```

;;Registers saved: none
;;Flags used:      MKFLAG
;;Macros used:    SYSF, ERRORM
;;Useage:        MAKE FCB

```

```

LXI     D,POINTR
XRA     A      ;;Zero A-register.
STA     POINTR+12 ;extent
STA     POINTR+32 ;current record
CALL    MAKE2?
CPI     0FFH   ;0=ok,FF means error.
JZ      ERROR  ;Jump to error routine if
; 0FFh.

```

```

IF WHERE
JMP     WHERE
ELSE
JMP     AROUND

```

ERROR:

```

ERRORM 'Disk full. ',AROUND; LEAVES 0FFH IN A-reg.

IF     NOT MKFLAG

```



```

MAKE2?:
    SYSF      22                ; Make new disk file.
MKFLAG      SET      TRUE      ; Only one copy.
    ENDIF

AROUND:
    ENDM                ;;MAKE

;*****
WRITESECTOR  MACRO  PTR, STAR
;*****
    LOCAL AROUND

    ;;Description:  Inline macro to write a disk
    ;;              sector.  PTR refers to file control
    ;;              block.  STAR is the symbol to print on
    ;;              the CRT for each successful sector "write".

    ;;Registers saved: none
    ;;Flags used:      WRFLAG
    ;;Macros used:    SYSF, PCHAR, ERRORM
    ;;Useage:         WRITESECTOR FCB, '*'
    ;;                WRITESECTOR FCB

    IF NOT NUL STAR
    PCHAR    STAR
    ENDIF

    IF NOT NUL PTR
    LXI     D, PTR
    ENDIF

    CALL    WRIT2?
    ORA    A                ;set flag

    IF     WRFLAG
    JNZ    NROOM?

    ELSE
    ;first time

    JZ     AROUND          ;ok

NROOM?:
    MVI    A, 0FFH        ; A= FFh  no more room.
    ERRORM 'ERROR: Disk full...ran out of room. ',

```



```

;
WRIT2?:
    SYSF      21                ;write disk sector
WRFLAG      SET      TRUE      ;only one copy.
    ENDIF    ;;WRFLAG
AROUND:     ;;WRITESECTOR
    ENDM

;*****
UNPROT      MACRO    POINTR
;*****
    LOCAL    AROUND

    ;;Description:      Inline macro to convert
    ;;      R/O file to R/W. POINTR refers to file
    ;;      control block address.

    ;;Registers saved:  none
    ;;Flags used:      UNFLAG
    ;;Macros used:     SYSF
    ;;Useage:          UNPROT FCB

    LXI      D,POINTR
    LDA      POINTR+9          ;load from file type.
    ANI      7FH              ;set for R/W.
    STA      POINTR+9          ;store at beginning of file
                                ;type.

    CALL     UNPR2?

    IF NOT UNFLAG
    JMP      AROUND

UNPR2?:
    SYSF      30                ;set file attributes.
UNFLAG      SET      TRUE      ;only one copy.
    ENDIF
AROUND:     ;;UNPROT
    ENDM

;*****
DELETE      MACRO    POINTR,WHERE
;*****
    LOCAL    AROUND, DEL3?

    ;;Description:      inline macro to delete an
    ;;      existing disk file. POINTR refers to file
    ;;      control block. If file is protected,
    ;;      branch to WHERE or DONE.

```



```

;;Registers saved: none
;;Flags used:          DEFLAG
;;Macros used:   SYSF, UNPROT, READCH,
;;              UCASE, CRLF, PFNAME, PRINT
;;Useage:        DELETE FCB, EXIT

```

```

LXI      D,POINTR
LDA      POINTR+9
ANI      80H          ;PROTECTED?
JZ       DEL3?       ;NO

```

```

;CRLF
;PFNAME POINTR
PRINT    ' is READ ONLY.  Delete? '
READCH
;UCASE
CPI      'Y'

```

```

IF NOT NUL WHERE
JNZ      WHERE

```

```

ELSE

```

```

JNZ      AROUND          ;quit.
ENDIF

```

```

UNPROT   POINTR
DEL3?:   CALL    DEL2?
         IF NOT DEFLAG
         JMP     AROUND

```

```

DEL2?:   SYSF    19          ;delete disk file.
DEFLAG   SET     TRUE       ;only one copy.
         ENDIF

```

```

AROUND:  ;;DELETE
         ENDM

```

```

;*****
CLOSEFILE MACRO POINTR
;*****
LOCAL AROUND, CLOSE3

```

```

;;Description:      An inline macro to "close"
;;                  a file to a disk. POINTR is the address of
;;                  the file control block.
;; NOTE: NOT FULLY IMPLEMENTED....MISSING SOME
;; REQD MACROS.

```



```
;;Registers saved: none
;;Flags used:          CL2FLAG
;;Macros used:  = SYSF
;;Useage:              CLOSE  FCB
```

```
IF NOT NUL POINTR
LXI    D,POINTR
ENDIF
```

```
CALL    CLOS2?
```

```
IF NOT CL2FLAG          ;one copy.
JMP    AROUND
```

```
CLOS2?:
SYSF    16              ;close disk file.
CL2FLAG SET    TRUE    ;only one copy.
ENDIF   ;CL2FLAG
AROUND: ;CLOSE
        ENDM
```

```
*****
GETFRAME MACRO  WHERE, SIZE
*****
```

```
LOCAL  AROUND, GETBYTE
```

```
;;Description:
```

```
;; WHERE is the address of where you want frame
;; written in memory.  SIZE is the number of
;; bytes contained in the frame.
```

```
;;
```

```
;; If SIZE is nul, then this macro expects it
;; to be already stored in D-reg.
```

```
;;
```

```
;; RETURNS:
```

```
;; B = 0FFh, (ABORT)
```

```
;; B = 0, A = 0 (validchecksum)
```

```
;; B = 0, A not 0 (bad checksum)
```

```
;;Registers saved: none
```

```
;;Flags used:  RPKFLAG
```

```
;;Macros used:  RDCOM
```

```
;;Useage:          GETFRAME BUFFER, FRAMESIZE
```

```
LXI    H,WHERE          ;h,1 point to dma address
```

```
IF NOT NUL SIZE
```

```
    MVI    D,SIZE          ;byte count, including
                          ;checksum
```

```
ENDIF
```



```

CALL      RPACK2?

IF NOT    RPKFLAG
JMP       AROUND

RPACK2?:

MVI      C,0                ;initialize checksum
GETBYTE:  RDCOM              ;A= 0FFh, ABORT.
                                ;A= 0, data waiting in
                                ;TSTORAGE.
CPI      0FFh              ;Check for ABORT.
JNZ      CONTIN
MOV      B,A
RET

CONTIN:

LDA      TSTORAGE          ;Otherwise, get the byte from
                                ;TSTORAGE and put in A-reg.

MOV      M,A                ;get byte, send it to buffer
XRA     C                    ;update checksum, leave
                                ;results in A-reg.
MOV      C,A                ;saving same in c
INX     H
DCR     D                    ;count down to zero then exit
                                ;subroutine.
JNZ     GETBYTE            ;repeating until all 129 bytes
                                ;in

                                ;last byte left in A-reg =
                                ;checksum
                                ;Still must be checked for
                                ;validity
                                ; (A=0: valid, A not 0:
                                ;not valid)
MVI     B,0                ;B=0: No ABORT, normal exit.
RET

RPKFLAG  SET      TRUE
ENDIF

AROUND:

ENDM

```

```

;*****
SETDMA   MACRO   POINTR
;*****

```


LOCAL AROUND

```
;;Description:      Inline macro to set the
;;      DMA address where the next sector will
;;      be read or written.
```

```
;;Registers saved:  none
;;Flags used:      DMFLAG
;;Macros used:     SYSF
;;Useage:         SETDMA  0080H
```

```
IF NOT NUL POINTR
LXI    D,POINTR
ENDIF
```

```
CALL   DMA2?
```

```
IF     NOT DMFLAG
JMP    AROUND
```

DMA2?:

```
SYSF   26          ;set DMA address.
DMFLAG SET   TRUE  ;only one copy.
ENDIF
```

AROUND:

```
;;SETDMA
ENDM
```

```
*****
UCASE  MACRO  REG
*****
```

```
LOCAL  NOTUP?
```

```
;;Description:      Inline macro to convert a
;;      a character in any register to uppercase.
;;      Omit parameter for A-reg.
```

```
;;Registers saved:
;;Flags used: none
;;Macros used:     none
;;Useage:         UCASE C
;;                UCASE
```

```
IF NOT NUL REG
PUSH   PSW          ;save
MOV    A,REG        ;get value
ENDIF
```

```
CPI    'Z'+7        ;uppercase?
JC     NOTUP?       ;no.
ANI    5FH          ;make uppercase.
```


NOTUP?:

```
IF NOT NUL REG
MOV     REG,A           ;put back
POP     PSW            ;restore
ENDIF

                                ;;UCASE

ENDM
```

READB MACRO

```
LOCAL AROUND,RBUFM,RBUF,CHARS$REMAINING,RBUFE
```

```
;;Description:      Inline macro to input a line
;;                 from console. Buffer is located at end of
;;                 macro. Get characters from buffer by calling
;;                 global subroutine GETCH in this macro.
;;                 Buffer pointer RBUFP is also global.
;;
;;                 Global address, "RBUFF$COUNT", contains the
;;                 actual number of chars read in.
```

```
;;Registers saved: HL,DE,BC
;;Flags used:      RCFLAG
;;Macros used:    none
;;Useage:         READB           ;This macro.
;;
;;                 CALL GETCH    ;Global sub-
;;                 ;routine.
```

```
CALL RDB2?
IF NOT RCFLAG
JMP AROUND
```

RDB2?:

```
PUSH H ! PUSH D ! PUSH B
LXI D,RBUFM
MVI C,10
CALL BDOS
LXI H,RBUFM+2
SHLD RBUFM-2
LDA CHARS$REMAINING
STA RBUFF$COUNT ;Save the actual number of
                 ;chars read.

POP B ! POP D ! POP H
RET
```


;global routine to get char from buffer.

GETCH:

```
LDA    CHARS$REMAINING ;get count
SUI    1                ;decr with carry.
RC     ;no more chars.
```

```
STA    CHARS$REMAINING
PUSH  H
LHLD   RBUF
MOV    A,M              ;get char.
INX    H                ;next.
SHLD   RBUF
POP    H
RET
```

```
;
RCFLAG SET TRUE ;only one copy.
```

```
;
RBUF$COUNT: DB 0 ;GLOBAL: actual number of
chars read.
```

```
RBUF: DW RBUF ;buffer pointer.
;console buffer address
```

```
RBUF: DB RBUF-RBUF ;max length
CHARS$REMAINING: DS 1 ;num of chars remaining
;after call to GETCH.
```

```
RBUF: DS 40D ;buffer start
RBUF: ;buffer end
```

```
ENDIF
```

```
AROUND: ;;READB
```

```
ENDM
```

```
;*****
```

```
READCH MACRO NOUPPER?,REG
```

```
;*****
```

```
LOCAL AROUND
```

```
;Description: Inline macro to read one char from
;console.
```

```
;If the second parameter (NOUPPER?) is blank,
;the char is converted to uppercase. This only
;works for alphabetic characters. If not, there is
;no conversion.
```

```
;;
```

```
;If the third parameter is blank, the char is left
;in the A-reg. If the third parameter is specified,
;then it is copied from the A-reg and placed in the
;specified register.
```

```
;;
```

```
;Registers saved:
```

```
;Flags used: CIFLAG
```



```

;;Macros used: UCASE
;;Useage:      READCH      - converted to upper,
;;              left in A-reg.
;;              READCH ,D  - converted to upper,
;;              put in D-reg.
;;              READCH 1   - not converted,
;;              left in A-reg.
;;              READCH 4,E - not converted,
;;              put in E-reg.

```

```
CALL RDCH?
```

```
IF NUL NOUPPER?
UCASE
ENDIF
```

```
IF NOT NUL REG
MOV REG,A
ENDIF
```

```
IF NOT CIFLAG
JMP AROUND
```

```
RDCH?:
```

```

SYSF      1
CIFLAG    SET      TRUE          ;only one copy.
ENDIF
```

```

AROUND:                                     ;;READCH
ENDM
```

```

;*****
HANDSHAKE MACRO TRANSMIT?
;*****
LOCAL AROUND, RDY, GETGEE
;;Description:
;;Registers saved:
;;Flags used:   SYNCFLAG
;;Macros used:   OUTPUT, READ$PORT
;;
;;Useage:       HANDSHAKE TRANSMIT (where TRANSMIT =
;;              TRUE)
;;              HANDSHAKE RECEIVE (where RECEIVE =
;;              FALSE)

IF TRANSMIT?
CALL TRANSMITSYNC?
ELSE
CALL RECEIVESYNC?

```



```

ENDIF

IF NOT SYNCFLAG
JMP     AROUND

TRANSMITSYNC2?:                ;; Handshake by the trans-
                                ;; mitter.
    OUTPUT  'R'
    READ$PORT
    CPI     'S'
    JNZ     TRANSMITSYNC2?
    OUTPUT  'G'
    RET

RECEIVESYNC2?:                ;; Handshake by the receiver.
    READ$PORT
    CPI     'R'
    JNZ     RECEIVESYNC2?

RDY:
    OUTPUT  'S'

GETGEE:
    READ$PORT
    CPI     'R'
    JZ      RDY

    CPI     'G'
    JNZ     GETGEE
    RET

    SYNCFLAG SET TRUE
ENDIF

AROUND:
    ENDM

```

```

;*****
SENDFRAME  MACRO  START, SIZE
;*****
    LOCAL AROUND, LOOP
    ;;Description:
    ;;    Packetsize = header + data + checksum
    ;;
    ;; Packetsize is to be given in the parameter
    ;; "SIZE", otherwise, it is expected in the A-reg.
    ;;
    ;; The address of the first byte of the header field
    ;; is to be specified by the parameter "START",
    ;; otherwise, it is expected in the HL-reg.

```



```

;;
;;Registers saved:
;;Flags used: SENDPFLAG
;;Macros used:      OUTPUT
;;Useage:

IF NOT NUL START      ;If given, load the frame's
LXI      H,START      ;starting address into HL.
ENDIF                ;Otherwise, expect it
                    ;in HL-reg directly.
IF NOT NUL SIZE      ;If size is given
      MVI      D,SIZE ;load it into D-reg.
ELSE                ;Else, expect it in A-reg
      MOV      D,A   ;and load it into D-reg.
ENDIF

CALL      SENDFRAME2?
IF NOT SENDPFLAG
JMP      AROUND

SENDFRAME2?:
      DCR      D      ;Subtract 1 for the checksum
                    ;(add on later).
      MVI      C,0

LOOP:
      OUTPUT   M,1
      XRA      C
      MOV      C,A
      INX      H
      DCR      D
      JNZ      LOOP

      OUTPUT   C,1
      RET
      SENDPFLAG SET TRUE

AROUND:
      ENDM

```


APPENDIX F: APPLE I/O MACRO LIBRARY

```

;=====
;APPLE.LIB .... macro library of I/O routines for the Apple
;computer
;=====

```

;List of macros contained:

```

;NAME                PARAMETERS

; output             macro  source,reg?,cchar
; rdcom              macro  <none>
; read$port          macro  regstatus,recmask,dataddr
; initialize          macro  <none>
; comptype           macro  <none>

```

```

; *****
; * E Q U A T E          S T A T E M E N T S *
; *****

```

;Terminal specific equates:

```

ESC          EQU    1BH
CLEAR        EQU    '*'
LO$LIGHT     EQU    ')'
HI$LIGHT     EQU    '('
CR           EQU    0DH
LF           EQU    0AH
BELL         EQU    07H

```

;I/O specific equates:

```

;A number of equate statements are dependent on the
;slot in which the RS-232 port resides. During
;assembly (only), the following statement allows
;one to change only the number following the EQU in
;order to change the slot used:
;===== SLOT SELECTION =====
IO$SLOT EQU 7 ;Use slots 1 thru 7 ONLY.
;=====

```



```

IO$BASE    EQU 0E000H    ;Base number for Microsoft
                    ;Z80 card.

IO$OFFSET EQU (IO$SLOT*10H)+80H

TXRDY     EQU    02H    ;"Transmit ready" mask for
                    ;serial I/O (slot _).
                    ;Used by: OUTPUT macro.

STATUS    EQU    IO$BASE+IO$OFFSET+0EH
                    ;Status register for
                    ;serial I/O.
                    ;Used by: OUTPUT,READ$PORT

DATA      EQU    IO$BASE+IO$OFFSET+0FH
                    ;Data register for
                    ;serial I/O.
                    ;Used by: OUTPUT

RXRDY     EQU    01H    ;"Receive ready" mask for
                    ;serial I/O .

KEYBOARD  EQU    IO$BASE

```

```

;*****
OUTPUT    MACRO    SOURCE, REG?, CCHAR
;*****
;;Description:      A macro that tests the output
;;                  port until it is clear, takes a data byte
;;                  and sends it out.  If REG? is nul, it expects to
;;                  send a "literal" 8 bit byte from the A-reg.
;;                  In this case, SOURCE = <data byte>.
;;                  If REG? is not nul, it moves an 8-bit data
;;                  byte from the 'source' register into the A-reg
;;                  and sends it.  In this case,
;;                  SOURCE = < source register>.
;;
;;                  If CCHAR is not nul, it is displayed on the crt
;;                  screen just before the data byte is sent out.
;;                  CCHAR is a confirmation character.
;;Registers saved:  none
;;Flags used:      OTFLAG
;;Macros used:     none
;;Example:        OUTPUT 63h    ; a literal.
;;                OUTPUT 'Q'    ; a literal.
;;                OUTPUT E,1    ; The data byte is
;;                               ; contained in E-reg.
;;                OUTPUT M,1    ; The data byte is in

```



```

;;                                     ; memory location
;;                                     ; pointed to by HL-reg.
;;         OUTPUT      'K', '*'
;;                                     ; The char 'K' is sent
;;                                     ; out and '*' is
;;                                     ; displayed at the
;;                                     ; screen.
;;         OUTPUT      E,1, '.'

```

```

LOCAL      WRCOM, AROUND

```

```

IF NUL REG?
MVI        A, SOURCE
ELSE
MOV         A, SOURCE
ENDIF

```

```

IF NOT NUL CCHAR
PCHAR      CCHAR
ENDIF

```

```

CALL       OUTPUT2?

```

```

IF NOT OTFLAG
JMP        AROUND

```

```

OUTPUT2?:
PUSH       PSW

```

```

WRCOM:

```

```

LDA        STATUS
ANI        TXRDY
JZ         WRCOM
POP        PSW
STA        DATA
RET

```

```

OTFLAG     SET    TRUE
ENDIF

```

```

AROUND:
ENDM

```

```

;*****
RDCOM      MACRO
;*****
;;Description:      An inline macro used ONLY by
;;      GETFRAME MACRO (see CPMMAC.LIB) .
;;      Samples the port.  If data is waiting,
;;      reads it, pushes it onto the stack, clears
;;      the A-reg, and returns.

```



```

;;
;;      If there is no data waiting, it enters a loop.
;;      Exit from the loop is accomplished only by
;;      data appearing on the port or by Control-C
;;      coming from the console.
;;
;;      When the Control-C is received, a 0FFh is
;;      loaded into A-reg and it returns.
;;
;; SUMMARY:
;;      data in port reg.:      return 0 in A-reg
;;                               and data in TSTORAGE.
;;
;;      no data in port
;;      and control-C from
;;      console:                return 0FFh in A-reg
;;                               (ABORT)
;;
;;      no data in port
;;      and no control-C
;;      from console:          remain in loop checking
;;                               port and console.
;;

;;Registers saved:  none
;;Flags used:      RDCOMFLAG
;;Macros used:    none
;;Example:        RDCOM

        LOCAL    AROUND, READ$IT
        CALL    RDCOM2?

        IF NOT RDCOMFLAG
        JMP     AROUND

RDCOM2?:
        LDA     STATUS          ;Slot 1: 0E09Eh
        ANI    RXRDY           ;Slot 1: 01h
        JNZ    READIT          ;Data at port: read it,
                               ;store it, and zero
                               ;the A-reg.

        LDA    KEYBOARD        ;Slot 1: 0E000h
        CPI    83H             ;Check to see if Control C.
        JNZ    RDCOM2?        ;Not Control-C: keep looping.

        MVI    A,0FFh         ;Control-C(signal for ABORT):
        RET                    ;Load 0FFh in A-reg and
                               ;return.

```



```

READIT:
    LDA     DATA           ;Slot 1: 0E09Fh
    STA     TSTORAGE       ;Store the data.
    XRA     A               ;Zero A-reg: Signal
                                ;for 'data waiting'.

    RET

```

```

TSTORAGE:

    DB     0H               ;Store data byte here.

    RDCOMFLAG SET TRUE
    ENDIF

```

```

AROUND:
    ENDM

```

```

;*****
READ$PORT MACRO REGSTATUS, RECMASK, DATADDR
;*****
;;Description:      An inline macro that reads the
;;      status of a port; if full, sends the data byte
;;      to the A-reg; if empty, zeroes the A-reg and sets
;;      the
;;      zero flag.
;;
;;      REGSTATUS is the address of the port's status
;;      register. RECMASK is a "bit mask" to check a specific
;;      "ready" bit in the port. DATADDR is the address of
;;      the data register for the port.
;;
;;      If REGSTATUS is nul, the macro uses the values
;;      specified in the EQU section of this file (above) for
;;      the values of REGSTATUS, RECMASK, and DATADDR.

;;Registers saved: none
;;Flags used:      RPFLAG
;;Macros used: none
;;Example:
;;      READ$PORT
;;      READ$PORT 08H,02H,05H

```

```

LOCAL AROUND
IF NOT NUL REGSTATUS
LDA     REGSTATUS
ANI     RECMASK
JZ      AROUND

LDA     DATADDR
JMP     AROUND
ENDIF

```



```
CALL    RP2?
IF NOT  RPFLAG
JMP     AROUND
```

RP2?:

```
LDA     STATUS
ANI     RXRDY
RZ                                     ;No input, zero A-reg &
                                       ;set zero flag.
```

```
LDA     DATA
RET
```

```
RPFLAG SET TRUE
ENDIF
```

AROUND:

```
ENDM
```

```
*****
```

```
INITIALIZE MACRO
```

```
*****
```

```
;;Description:      Used to initialize the
;;                  I/O routines of a specific computer.
;;                  Usually a dummy operation if not used
;;                  by the particular computer.
```

```
;;Registers saved: none
```

```
;;Flags used: none
```

```
;;Macros used: none
```

```
;;Example:          INITIALIZE
```

```
    NOP
    ENDM
```

```
*****
```

```
COMPTYPE MACRO
```

```
*****
```

```
;;Description:      Used to print out the name ;;of the
computer on the screen.
```

```
;;Registers saved: none
```

```
;;Flags used: none
```

```
;;Macros used: PRINT
```

```
;;Example:          COMPTYPE
```

```
    PRINT <'Apple II'>
    ENDM
```


APPENDIX G: NORTHSTAR I/O MACRO LIBRARY

```

;=====
;NSTAR.LIB .... macro library of I/O routines for the
;                               NORTHSTAR computer.
;=====

```

;List of macros contained:

```

;NAME                PARAMETERS

; output            macro source,reg?,cchar
; rdcom             macro
; read$port         macro regstatus,recmask,dataddr
; initialize         macro <none>
;                   c o m p t y p e                m a c r o                < n o n e >

```

```

;-----
;                   E Q U A T E           S T A T E M E N T S
;-----

```

;TERMINAL equates:

```

;CLEAR EQU 1B2AH
CR EQU 0DH
LF EQU 0AH
BELL EQU 07H
ESC EQU 1BH

```

;All of the following can be used by the macro
;PRINT to control screen attributes. The format
;used is illustrated in the following example:

```

;                   PRINT <ESC, (one of the following) >

```

```

CLEAR            EQU '+'
ST$H$INT        EQU ')'
END$H$INT        EQU '('
ST$INVERSE       EQU 'j'
END$INVERSE      EQU 'k'
START$BLINK      EQU '^'
END$BLINK        EQU 'q'
ST$UNDERLINE     EQU 'l'
END$UNDERLINE    EQU 'm'
PROTECT$ON       EQU '&'
PROTECT$OFF      EQU 27H
MOVE$CURS        EQU '=' ;Used for X-Y addressing.

```

```

;       Example:
;                   PRINT <ESC,MOVE$CURS,ROW,COL>

```



```

;I/O      specific EQU's:

TXRDY     EQU      01H      ;"Transmit ready" mask for
                                ;serial I/O.
                                ;Used by: OUTPUT
STATUS    EQU      05H      ;Status register for serial I/O
                                ;(slot _).
                                ;Used by: OUTPUT, INPUT$STS
DATA      EQU      04H      ;Data register for serial I/O
                                ;(slot _).
                                ;Used by: OUTPUT, INPUT$STS
RXRDY     EQU      02H      ;"Receive ready" mask for serial
                                ;I/O (slot _).
                                ;Used by: INPUT$STS
CONSOLE$STATUS EQU 03H      ;NORTHSTAR console status
CONSOLE$DATA  EQU 02H      ;NORTHSTAR console data

;*****
OUTPUT     MACRO   SOURCE, REG?, CCHAR
;*****
        LOCAL   WRCOM, AROUND

;;Description:          A macro that tests the output
;;                    port until it is clear, takes a data byte
;;                    and sends it out. If REG? is nul, it expects
;;                    to send a "literal" 8 bit byte from the
;;                    A-reg. In this case, SOURCE = <data byte>.

;;                    If REG? is not nul, it moves an 8-bit data
;;                    byte from the 'source' register into the
;;                    A-reg and sends it. In this case, SOURCE =
;;                    < source register>.

;;                    If CCHAR is not nul, it is displayed on the
;;                    crt screeb just before the data byte is sent
;;                    out. CCHAR is a confirmation character.

;;Registers saved:    none
;;Flags used:         OTFLAG
;;Macros used:        none
;;Useage:             OUTPUT 63h      ;a literal.
;;                   OUTPUT 'Q'      ;a literal.
;;                   OUTPUT E,1      ;The data
;;                                   ;byte is
;;                                   ;contained
;;                                   ;in E-reg.

;;                   OUTPUT M,1      ;The data
;;                                   ;byte is in
;;                                   ;memory
;;                                   ;location

```



```

;;
;;
;;          OUTPUT    'K',, '*' ;pointed to
;;          ;by HL-reg.
;;          ;The char
;;          ;'K' is sent
;;          ;out and '*'
;;          ;is display-
;;          ;ed at the
;;          ;screen.
;;          OUTPUT    E,1, '.'
IF NUL REG?
MVI     A,SOURCE
ELSE
MOV     A,SOURCE          ; contained in a register or
                           ; memory.
ENDIF

IF NOT NUL CCHAR
PCHAR  CCHAR
ENDIF

CALL   OUTPUT2?

IF NOT OTFLAG
JMP    AROUND
OUTPUT2?:
PUSH   PSW

WRCOM:
IN     STATUS
ANI    TXRDY
JZ     WRCOM
POP    PSW
OUT    DATA
RET

OTFLAG SET TRUE
ENDIF

AROUND:
ENDM

```

```

;*****
RDCOM    MACRO
;*****

```

```

LOCAL   AROUND, READ$IT

```

```

;;Description:          An inline macro used ONLY by
;;          GETFRAME MACRO (see CPMMAC.LIB) .
;;          Samples the port.  If data is waiting,
;;          reads it, pushes it onto the stack, clears
;;          the A-reg, and returns.

```



```

;;
;;      If there is no data waiting, it enters a
;;      loop.  Exit from the loop is accomplished
;;      only by data appearing on the port or by
;;      Control-C coming from the console.
;;
;;      When the Control-C is received, a 0FFh is
;;      loaded into A-reg and it returns.
;;
;; SUMMARY:
;;      data in port reg.:      return 0 in A-reg
;;                               and data in TSTORAGE.
;;
;;      no data in port
;;      and control-C from
;;      console:                 return 0FFh in A-reg
;;                               (ABORT)
;;
;;      no data in port
;;      and no control-C
;;      from console:           remain in loop checking
;;                               port and console.
;;

```

```

;;Registers saved:  none
;;Flags used:      RDCOMFLAG
;;Macros used:    none
;;Useage:         RDCOM

```

```

CALL    RDCOM2?
IF NOT RDCOMFLAG

```

```

JMP     AROUND

```

```

RDCOM2?:

```

```

IN      STATUS          ;(05h)
ANI     RXRDY           ;(02h)
JNZ     READIT         ;Data at port: read it, store
                       ;it, and zero the A-reg.

```

```

IN      CONSOLE$STATUS ;(03h)
ANI     RXRDY           ;(02h)
JZ      RDCOM2?        ;Nothing on console or at
                       ;port: loop

```

```

IN      CONSOLE$DATA    ;(02h)
ANI     7FH             ;Somthing from console.
CPI     03H            ;Check to see if Control C.
JNZ     RDCOM2?        ;Not Control-C: keep looping.

```



```

MVI      A,0FFh      ;Control-C (signal for ABORT):
RET      ;Load 0FFh in A-reg and
          ;return.

```

```

;-----

```

READIT:

```

IN      DATA      ;(04h)
STA     TSTORAGE   ;Store the data.
XRA     A          ;Zero A-reg: Signal data
          ;waiting.
RET

```

```

TSTORAGE:  DB      0H      ;Store data byte here.

```

```

RDCOMFLAG SET     TRUE
ENDIF

```

AROUND:

```

ENDM

```

```

;*****
READ$PORT MACRO  REGSTATUS, RECMASK, DATADDR
;*****
LOCAL AROUND

```

```

;;Description:      An inline macro that reads
;;                  the status of a port; if full, sends the
;;                  data byte to the A-reg; if empty, zeroes
;;                  the A-reg and sets the zero flag.
;;
;;                  REGSTATUS is the address of the port's
;;                  status register. RECMASK is a "bit mask"
;;                  to check a specific "ready" bit in the port.
;;                  DATADDR is the address of the data register
;;                  for the port.
;;
;;                  If REGSTATUS is nul, the macro uses the
;;                  values specified in the EQU section of
;;                  this file (above) for the values of
;;                  REGSTATUS, RECMASK, and DATADDR.

```

```

;;Registers saved: none
;;Flags used:      RPFLAG
;;Macros used: none
;;Useage:          READ$PORT
;;                  READ$PORT 08H,02H,05H

```



```
IF NOT NUL REGSTATUS
IN      REGSTATUS
ANI     RECMASK
JZ      AROUND
```

```
IN      DATADDR
JMP     AROUND
ENDIF
```

```
CALL    RP2?
IF NOT  RPFLAG
JMP     AROUND
```

RP2?:

```
IN      STATUS      ;
ANI     RXRDY       ;
RZ      ; No input, zero A-reg & set
;zero flag.
```

```
IN      DATA      ;
RET     ;
```

```
RPFLAG  SET      TRUE
ENDIF
```

AROUND:

```
ENDM
```

```
*****
```

```
INITIALIZE MACRO
```

```
*****
```

```
;;Description:      Used to initialize the
;;                  I/O routines of a specific computer.
;;                  Usually a dummy operation if not used
;;                  by the particular computer.
;;Registers saved: none
;;Flags used: none
;;Macros used: none
;;Useage:           INITIALIZE
```

```
NOB
ENDM
```

```
*****
```

```
COMPTYPE  MACRO
```

```
*****
```

```
;;Description:      Used to print out the name
;;of the computer on the screen.
;;Registers saved: none
;;Flags used: none
```



```
;;Macros used: PRINT
;;Useage:          COMPTYPE

PRINT <'North Star Horizon'>
ENDM
```


LIST OF REFERENCES

1. Metcalfe, R.M. and Boggs, D.R., "ETHERNET: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, Vol. 19, No.7, July 1976, pp. 395-404.
2. Tanenbaum, A. S., Computer Networks, Prentice-Hall, 1981.
3. Digital Research Inc., CP/M Operating System Manual, 1982.
4. Digital Research Inc., CP/M MAC Macro Assembler: Language Manual and Applications Guide, 1980.
5. Hogan, T., Osborne CP/M User Guide, Osborne/McGraw-Hill, 1982.
6. Electronic Industries Association, Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange, 1969.
7. California Computer Systems, Asynchronous Serial Interface Model 7710 Owner's Manual, 1980.
8. Brooks, F. P., The Mythical Man-Month, Addison-Wesley, 1982.
9. Ross, D.T., Goodenough, J.B., and Irvine, C.A., "Software Engineering: Process, Principles, and Goals," Computer, May 1975, pp. 54-64.
10. Shooman, Martin L., Software Engineering: Design, Reliability, and Management, McGraw-Hill, 1983.
11. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, December 1972, pp. 1053-1058.
12. Kernighan, B.W., and Plauger, P.J., Software Tools, Addison-Wesley, 1976.
13. Smith, L.B., "The Use of Interactive Graphics To Solve Numerical Problems", Communications of the ACM, Vol. 13, No. 10, October 1970, pp. 625-634.

14. Mozeico, H., "A Human/Computer Interface to Accomodate User Learning Stages", Communications of the ACM, Vol. 25, No. 2, February 1982, pp. 100-104.
15. Miller, R.B., "Response Time in Man-Computer Conversational Transactions", a report for the International Business Machine Corporation, 1968.
16. Miller, A.R., Mastering CP/M, Sybex, 1983.
17. Intel Corporation, 8080/8085 Assembly Language Programming Manual, 1982.
18. Zaks, R., Programming the Z80, Sybex, 1982.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
2. Library, Code 0142 Naval Postgraduate School Monterey, CA 93940	2
3. Professor Gordon E. Latta, Code 53Lz Department of Mathematics Naval Postgraduate School Monterey, CA 93940	1
4. Professor Norman R. Lyons, Code 54Lb Department of Administrative Sciences Naval Postgraduate School Monterey, CA 93940	1
5. Department Chairman Department of Administrative Sciences, Code 54Ea Naval Postgraduate School Monterey, CA 93940	1
6. LT John F. Hall, II, USN SMC 1824 Naval Postgraduate School Monterey, CA 93940	1
7. Salinas-Monterey User Group P.O. Box 3207 Carmel, CA 93921	1
8. LCDR Thomas C. Carnahan, USN Helicopter Squadron Light Three Sioux Naval Station Mayport, FL 32228	2
9. LT Michael K. Waters, USN 33249 S.W. Watts Scapoosse, OR 97056	2

10. Computer Technology Curricular Office
Code 37
Naval Postgraduate School
Monterey, CA 93940

1

202535

Thesis

C261365 Carnahan

c.1 Microcomputer net-
working: a CP/M-
based application.

24 JUN 81

51060

0

202535

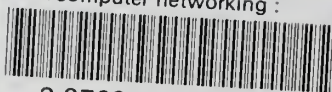
Thesis

C261365 Carnahan

c.1 Microcomputer net-
working: a CP/M-
based application.

thesC261365

Microcomputer networking :



3 2768 002 08555 7
DUDLEY KNOX LIBRARY