To the Graduate Council:

I am submitting herewith a thesis written by Zafrul Ahsan entitled "Asynchronous File Transfer between the TI-990/12 Minicomputer and Other Computers." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Donald W. Bouldin

Donald W. Bouldin, Major Professor

We have read this thesis and
recommend its acceptance:

Robert E. Bodenheimer

Robert W. Rochelle

Accepted for the Council:

Lew Minkel

Vice Provost
and Dean of the Graduate School

ASYNCHRONOUS FILE TRANSFER BETWEEN THE TI-990/12 MINICOMPUTER AND OTHER COMPUTERS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Zafrul Ahsan

December 1985

ACKNOWLEDGMENTS

The author wishes to express his profound thanks and deep appreciation to his committee, Drs. D. W. Bouldin, R. E. Bodenheimer and R. W. Rochelle for their individual guidance, patience and helpful advice throughout the course of this work. Special thanks go to Dr. Bouldin who was constantly available to advise on the research and writing of the thesis. Appreciation is also extended to Mr. James Barnes, Mr. Jack Watson and Mr. Lendon Adkins of Technical Services of Electrical Engineering Department, for their technical assistance in doing the project.

Further appreciation is expressed to my parents for their support and encouragement throughout my academic career.

ABSTRACT

Asynchronous file transfers between microcomputers and mainframes are gaining popularity at most institutions in the United States. The interprocessor communication is done through a serial telecommunication line using an asynchronous protocol. This thesis presents software and hardware considerations for the development of such a protocol between the TI-990/12 minicomputer and other computers. Programs are written for the implementation of a file transfer mechanism between the TI-990 minicomputer and the DEC-10 mainframe. All operator instructions for receiving a file from, and transmitting a file to, the DEC-10 are presented. A receiving Kermit program for the TI-990 minicomputer is also included as an alternative approach for file transfer between the TI machine and the DEC-10 mainframe. A well documented software presentation is included for future work on this project. Certain program limitations are presented along with suggestions for future modifications and enhancements to software.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A great deal of attention has been focused recently on the developments in computer networking such as IBM's system network architecture (SNA), the IEEE 802 committee, the latest ethernet interfaces, fiber optics, satellite communications, and broadband versus baseband transmissions. But little attention has been focused on the single working mechanism, which may be the most widely used in the real world for direct interprocessor communication --- the asynchronous protocol. This form of protocol is operative in some form at most institutions which have a need to transfer files between microcomputers and central computers.

The University of Tennessee, Knoxville, has large time-sharing computers like the DEC-10 and the IBM 3081 at a central site. Computer facilities within the Department of Electrical Engineering include a VAX-11/780, a PDP-11/40, an HP-9000, a TI 990/12 with 10 workstations, several LSI-11 systems, and a remote computing center that links two DEC-system 10s, two VAX-11/780s, and an IBM 3081. Prior to the undertaking of this project, the TI 990/12 minicomputer within the department had no file-transfer mechanism with the DEC and IBM mainframes; nor did it have any with other computers within the department. To use the TI

990 more effectively in various research projects, the department needed to develop some kind of file-transferring mechanism for the TI 990 which would allow it to communicate with all our computers--large and small. With this objective in mind, the author started to work to develop an inexpensive and reliable means of communication, at least with the DEC-10 mainframe, if not with all other computers.

Since, commercial local-area-networking products are expensive, not widely available, and unsuitable for one-shot or long-haul applications, the author uses an inexpensive serial telecommunication line for point-to-point file transfer. At the beginning, the author had the intention of invoking an asynchronous file-transfer protocol, called "Kermit", for the TI 990 minicomputer. Although the author has written a compatible Kermit program for the TI-990 minicomputer to receive files, it is not yet in its full working condition. But a file-transferring mechanism has been developed for the TI to communicate with the DEC-10 mainframe and also the VAX-11 computer within the department. The method developed works well for a system which uses the asynchronous serial telecommunications line for connecting terminals to computers and follows the EIA RS-232C standard. Serial connections can be made in many ways: dedicated local cables ("null modem" cables), leased telephone circuits, and dial-up connections. Dial-up connections can be initiated manually from the home or office using

an inexpensive acoustic coupler or automatically from one computer to another using a programmable dial-out mechanism. The asynchronous serial line offers the ordinary user a high degree of convenience and control in establishing intersystem connections and at relatively low cost.

The only communication medium common to both the TI-990/12 minicomputer and most other computers (e.g. the DEC-10 main-frame) is the asynchronous serial telecommunication line. The line is connected to the terminal of the computer through a modem which follows an RS-232C standard. Several different ways that the TI-990 minicomputer can communicate with another computer are illustrated in Figure I-1.

(1) TI-990/12 minicomputer uses a Four Channel Communications Controller (FCCC) board, which is capable of interfacing up to four EIA RS-232C compatible communications devices to a TI-990 model computer. The FCCC provides control, error detection, data formatting and temporary storage of data for each of four independent communication channels. Data transfer between the TI-990 minicomputer and the other computer takes place in each channel through a TMS 9903, the Synchronous Communications Controller (SCC). On-board ROM of the FCCC supports various communication protocols like the IBM-3780/2780, Digital Data Communications Message Protocol (DDCMP), SDLC/HDLC/ADCCP bit-oriented protocols, TI-914A/Universal Polling Adapter (UPA) etc.

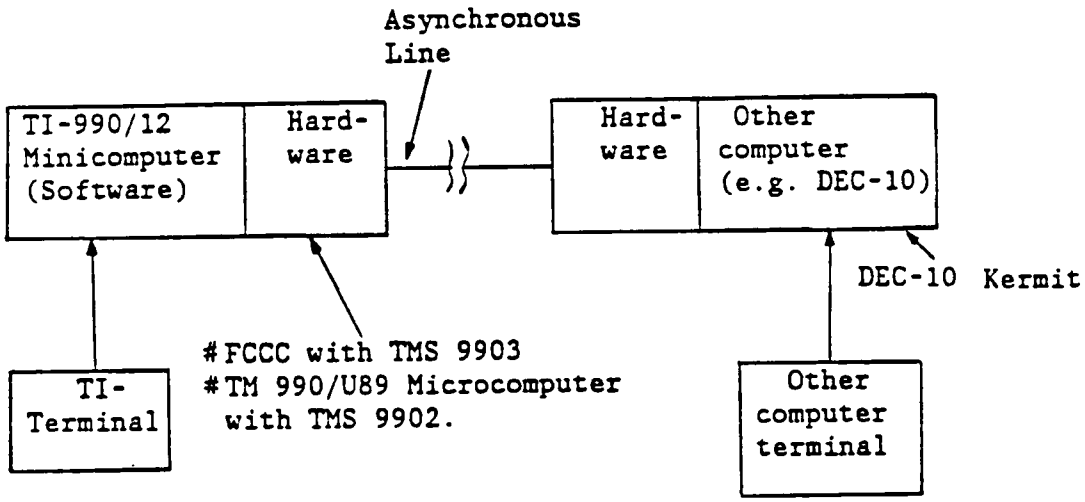


FIGURE I-1. Hardware and Software Considerations for Communication between TI-990/12 Minicomputer and Another Computer.

The TI-990 minicomputer has the capability to communicate with any computer which uses any of these protocols. Also, the FCCC allows the operator to download his own protocol in its user RAM.

(2) TI-990/12 minicomputer can communicate with a TM 990/U89 microcomputer using the AMPL (Advanced Microprocessor Prototyping Laboratory) utility of the microcomputer. The microcomputer kit uses an on-board TMS 9902, the Asynchronous Communications Controller (ACC). The ACC provides an interface between the TM 990/U89 microcomputer kit and a serial asynchronous communications channel, which follows the RS-232C standard.

(3) A software communication protocol called Kermit can also be used for communication between the TI-minicomputer and the other computer. Presently, implementations of Kermit exist for over 50 computer systems. Kermit is not a portable program but a portable protocol. Therefore, a compatible Kermit program for the TI-990 minicomputer should be able to communicate with any other computer system on which the Kermit has been implemented.

Chapter II contains software design considerations for developing a communication protocol called Kermit. Although Kermit was not implemented on the TI, an overview has been given in this chapter, which may be of some help to someone willing to work in that direction in the future. Chapter II also contains a description of a receiving Kermit program written by the author for the TI-990/12 minicomputer. Chapter III contains some hardware

design considerations of the TI-990 minicomputer for communications with other computers. This chapter is mainly concerned with using the Four Channel Communications Controller of TI-990 minicomputer. Chapter IV deals with operator instructions for file transfer. The chapter gives all the pieces of information necessary to transmit a file to, and receive a file from, the other computer (e.g. the DEC-10 mainframe). Chapter V contains the summary and conclusions.

CHAPTER II

SOFTWARE DESIGN CONSIDERATIONS

In order to develop software for a reliable file transfer between two computer systems, it is necessary to consider the various factors that affect its design. In this chapter the design considerations for developing such software are presented along with a full description of a file transferring protocol called Kermit which has been implemented on many computer systems. The chapter ends with a description of the actual receiving protocol developed for the TI- 990/12 minicomputer.

Communication Protocol

A communication protocol is a set of rules for handling packets of information. When two computers are connected to a serial line, information can be transferred from one machine to the other, provided one side can be instructed to send the information and the other side to receive it. A typical exchange between a terminal station and a computer on a point-to-point private line, as depicted in Figure II-1, will help to illustrate how a protocol works between computers.

Before such a protocol can be developed, the following design considerations should be taken into account:

1) NOISE-- It is rarely safe to assume that there will be no electrical interference on a line; any long or switched data

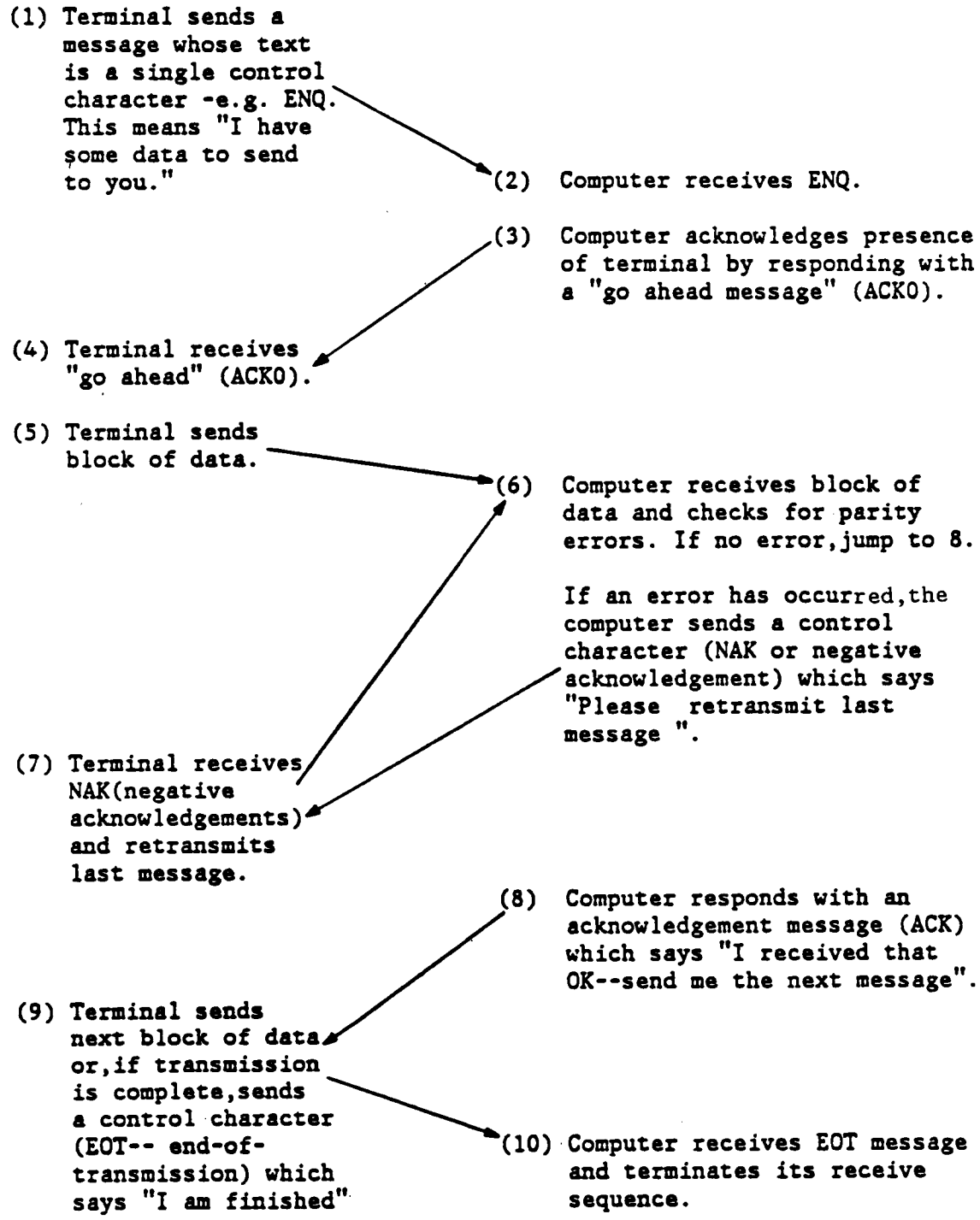


FIGURE II-1. Typical Data Exchange Using a Protocol.

communication line will have occasional interference, or noise, that typically results in garbled or extra characters. Noise can corrupt data which may remain unnoticed until it is too late to recover the actual data.

2) SYNCHRONIZATION-- Although line speeds (baud rate) at the two ends of the connection may match, a problem is encountered if the data are coming faster than the receiving machine can handle them. This may cause the receiving machine not to process a steady stream of input at that speed. Its central processor may be too slow or too heavily loaded or its buffer full. The typical symptom of a sychronization problem is lost data; most operating systems will discard incoming data they are not prepared to receive.

3) LINE OUTAGES-- A line may be malfunctioning for a short period because of a faulty connector, loss of power or a similar reason. On dial-up or switched connections, such intermittent failures will cause the carrier signal to be dropped and the connection to be closed. But for any connection in which the carrier signal is not used, the symptom will be lost data.

Other communication media, such as a parallel data bus, have safeguards built in to prevent or minimize these effects. For instance, distances may be strictly limited, the environment controlled, special signals may be available for synchro-

nization, and so forth. The serial telecommunication line does not provide safeguards of this kind; hence it is considered to be an intrinsically unreliable medium.

To determine correct and complete transmission of data between two machines, the machines can compare data before and after transmission. A scheme commonly used for file transfer employs cooperating programs running simultaneously on each machine, communicating in a well-defined concise language. The sending program divides outbound data into discrete pieces, adding special information to each piece describing the data for the receiving program. The result is called a "packet". The receiver separates the description from the data and determines whether they still match. If so, the packet is acknowledged, and the transfer proceeds. If not the packet is negatively acknowledged and the sender retransmits it. This procedure is repeated for each packet until it is received correctly. This is known as a communication protocol-- a set of rules for forming and transmitting packets, carried out by programs that embody those rules.

Accommodating Diverse Systems

Computer systems vary in their capabilities over a wide range. Consider the characteristics of a microcomputer versus a mainframe. Generally, the microcomputer is a single user system in which the serial communication port is strictly an external device. A mainframe is often the host to multiple, simultaneous users at terminals. Some mainframe systems allow users to assign another terminal line on the same machine as an external I/O device. The operating system may reformat control characters into printable equivalents; or translate lowercase letters to uppercase (or vice versa) or tabs into spaces. Also, based on the terminal's declared width and length, long lines may be wrapped around or truncated, formfeeds translated to a series of linefeeds, and pauses inserted at the end of each screen full of output. Input from a job's controlling terminal may also be handled specially: lowercase letters may be converted to uppercase, a linefeed may be supplied when a carriage return is typed, or control characters may invoke special functions, such as line editing or program interruption. The DECSYSTEM-20 is an example of a computer where any of these might happen.

The point to be considered here is that care must be taken to disable special handling of a controlling terminal when it is to be a vehicle for interprocessor communication. But some systems

simply do not allow certain of these features to be disabled; so, file transfer protocols must be designed around them.

LINE ACCESS-- Line access is either full or half-duplex. On mainframes, the host echoes characters typed at the terminal in full-duplex but not in half-duplex. Full-duplex systems can usually accommodate half-duplex communication but not vice versa. IBM mainframes are the most prevalent half-duplex systems.

BUFFERING AND FLOW CONTROL-- Some systems cannot handle sustained bursts of input on a telecommunication line since the input buffer can fill up faster than it can be emptied, especially at high line speeds. Some systems attempt to buffer "typeahead" (unrequested input); others discard it. Those that buffer typeahead may or may not provide a mechanism to test or clear the buffer.

Systems may try to regulate how fast characters come in using a flow control mechanism, either in the data stream or parallel to it (modem control signals), but no two systems can be assumed to honor the same conventions for flow control. Even when flow control is being done control signals themselves are subject to noise corruption. A burst of more than a line's worth of characters (60 to 100) into a terminal port at moderate speed could result in some loss of data-- or for some hosts it may be worse. An example is the communications front-end of the DEC system-2060 which

has been designed on the statistical assumption that all terminal inputs come from human fingers, and it cannot allocate buffers fast enough when this assumption is violated by sending continuous data simultaneously from several microcomputers attached to terminal ports.

CHARACTER INTERPRETATION-- Interpretation of characters that arrive at the terminal port differ from system to system. A host can receive certain characters as sent, translate some characters, take special action on others etc. Communications front-ends might swallow certain characters like DC1, DC2, etc. for flow control, padding (NUL, DEL), or transfer of control (escape) characters. The characters that typically trigger special behavior are the ASCII control characters, including the delete character. For instance, out of 33 control characters the TI-990/12 minicomputer uses about 30 of them to invoke special functions.

Some operating systems allow an application to input a character at a time; others delay passing the characters to the program until a logical record has been detected, usually a sequence of characters terminated by a carriage return or linefeed. Most DEC operating systems keep the carriage return but also add a linefeed.

TIMING-OUT-- Hosts may or may not have the ability to time out. While communicating with another computer, it is desirable to be able to give some input command without waiting forever should the incoming data be lost. A lost message could result in the deadlock of a protocol in which one side is waiting forever for the message to come while the other side is waiting for a response. Some systems can set timer interrupts to allow escape from potential blocking operations. However, there are many microcomputers which do not have this facility. When time-outs are not possible, they may be simulated by sleep-and-test or loop-and-test operations or deadlock systems may be awakened by manual intervention.

FILE ORGANIZATION-- Some computers store all files in a uniform way, such as the linear stream of bytes that is a UNIX file. Other computers have more complicated or diverse file organizations and access methods. Even simple microcomputers can present complications when files are treated as uniform data to be transferred; for instance, under CP/M the ends of binary and text files are determined differently. The major question which arises for any operating system is whether a file is specified sufficiently by its contents and its names or if additional information is required to make the file valid. A simple generalized file-transfer facility may be able to transmit the file's name and its contents but not every conceivable attribute a file might possess. Designers

of expensive networks have gone to great lengths to pass file attributes along when transferring files between different systems.

Invertibility is the sending of a copy of a file to another system, receiving a copy of that file back from the other system, and having all the attributes of this second copy of the file match the original file's characteristics. DECnet is able to provide invertibility for operating systems like VMS or RSX, which can store the necessary file attributes along with the file. But simpler file systems like those of TOPS-10 or TOPS-20 can lose important information about the incoming files. Invertibility is a major problem with no simple solution. Fortunately, file transfer between unlike systems usually involves source program, data, textual informations etc. which are sequential in organization, and for which any required transformations are simple and not dependent on any special file attributes. To allow the necessary transformations to take place on textual files between systems, there must be a standard way of representing these files during transmission.

SOFTWARE-- Systems have different application software. In particular, it cannot be assumed that a system will have a particular programming language. Common languages like FORTRAN, BASIC and PASCAL may not be available in some computers. Even when two different systems support the same

language, it is unrealistic to expect that the two implementations will be perfectly compatible with each other. Therefore, a general-purpose file transfer protocol should not be written in or geared toward the features of any particular computer language.

Kermit-- A File Transfer Protocol

Kermit is a protocol for point-to-point file transfer over serial telecommunication lines. It has the following characteristic features:

* Communication takes place over ordinary terminal connections.

* Communication is half-duplex. This allows both full- and half-duplex systems to participate, and it eliminates the echoing that would otherwise occur for characters at a terminal.

* The packet length is variable, but the maximum is 96 characters so that most hosts can take packets in without any buffering problems.

* Packets are sent in alternate directions; a reply is required for each packet. This allows half-duplex systems to participate and prevent buffer overruns that would occur on some systems if packets were sent back-to-back.

* A time-out facility when available, allows transmission to resume after a packet is lost.

* All transmission is in ASCII. Any non-ASCII host should

convert into ASCII before transmitting over the serial line. ASCII control characters are prefixed with a special character and then converted to printable characters to ensure that they arrive as sent. A single ASCII control character (normally SOH--start of header) is used to mark the beginning of a packet.

* Binary files can be transmitted by a similar prefix scheme or by use of the parity bit when both sides have control of it.

* Logical records (lines) in textual files are terminated during transmission with prefixed carriage return/line feed sequences, which are transparent to the protocol and may appear anywhere in a packet.

* Only a file's name and contents are transmitted -- no attributes. The user should make certain that the file has been received correctly on the target system. Within this framework, invertible transfer of text files can be assured; but in the case of non-text files, invertibility depends on the capabilities of the particular implementations of Kermit and the host operating systems.

* Kermit need not be written in any particular language. It is not a portable program but is a portable protocol.

Thus, Kermit accommodates itself to many different systems by conforming to a common subset of their features. The resulting simplicity and generality allows Kermit on any machine to

communicate with Kermit on any other machine. The communication may be from microcomputer to mainframe, microcomputer to microcomputer, or mainframe to mainframe. Packets are exchanged back-and-forth, which keeps both sides synchronized; but the protocol can be called asynchronous only because the communication hardware itself operates asynchronously. Kermit does all the operations by itself. To transfer files between a microcomputer and a mainframe, Kermit is executed on the microcomputer, in terminal emulation mode to permit connection to the mainframe. The user then logs in and executes Kermit on the mainframe and then escapes back to the microcomputer and issues commands to the microcomputer's Kermit to send or fetch the desired files. Any inconvenience implicit in this procedure is a consequence of the power it gives the ordinary user to establish reliable connections between computers that could not otherwize be connected.

Kermit Packets

To ensure reliable transmission, Kermit packets contain some additional information in addition to the actual data to be transferred. Several considerations are to be taken into account when designing a packet layout-- how to represent data, how to delimit fields within the packet, how to delimit the packet itself, and how to arrange the fields within the packet. Since the

transmission medium itself is character-oriented, bit-oriented protocols like HDLC (high-level data-link control) and SDLC (synchronous data-link control) are not feasible here. Therefore, the smallest unit of information in a packet must be the ASCII character.

CONTROL FIELDS-- A packet may be altered as it passes through a hierarchy of protocol levels, each level adding its own information at the ends of an outbound packet or stripping its information from the ends of an incoming packet. The fields for each layer must be arranged so that they can be found, identified and interpretted correctly at the appropriate level. Since the Kermit packets are short, the control information should be made as small as possible per packet. It would be desirable to limit the control fields to one character each. Since there are 95 printable characters out of a total of 128 ASCII characters less the delete character [DEL], a single character can be used to represent values from 0 to 94.

The packet sequence number is used to detect duplicate or missing packets. It is not likely that a large number of packets could be lost, especially when packet (n) is acknowledged before packet (n+1) is sent. The sequence number can thus be a small quantity, which wraps around to its minimum value when it exceeds a specified maximum value.

A small packet length can be ensured by specifying the

packet length, using one character. Since, there are 95 printable ASCII characters that can be transmitted, the maximum packet length is 95.

Kermit uses a checksum. The checksum can be of fixed length. The actual length will depend on the desired balance between efficiency and error detection.

Both checksum and packet length are used to detect corrupted, missing or extra characters. These are the essential fields to ensure reliable transmission. So far, only packets which carry the actual file data have been considered but some special packets contain control information. For instance, to tell the file pathname to the remote host and also, to send it the information that the transmission is complete, special packets are used. This can be accomplished with a packet type field. The number of functions to be specified in this field can also be done with one character.

PACKET FRAMING -- SOH (control A) is often chosen as a mark for the beginning of a packet. This character cannot appear anywhere inside the packet. SOH is chosen because, unlike most other control characters, it is generally accepted upon input at a terminal as a data character rather than as an interrupt or break character on most mainframes. If it is not possible for a system to send or receive SOH, the start-of-packet character can be redefined

to be any other control character; it is not necessary for the two sides to use the same character.

To recognize the end of a packet, there are three principal options: fixed length, distinguishing packet-end character and length field. Arguments can be made in favor or against each depending on what happens when a terminator or length character is lost or corrupted. Kermit uses a length field.

In Kermit a data packet starts with an SOH. To take a packet in, Kermit gets characters from the line until it encounters the SOH. After the SOH, the next character is the length. Kermit reads and decodes the length and then reads that number of subsequent characters to complete the packet. If another SOH is encountered before the count is exhausted, the current packet is forgotten and a new one started automatically. This strategy allows arbitrary amounts of noise to be generated spontaneously between packets without interfering with the protocol.

ENCODING-- Kermit terminates logical records with carriage return/linefeed combinations when transmitting textual data. On record-oriented systems, trailing blanks or length fields are removed and a CR/LF appended to outbound records, with the inverse record performed on incoming records.

Kermit makes each character in the packet printable. Any unprintable character is transformed into a printable one by using a special prefix character, normally , which precedes the printable

character. The transformation is done by complementing the seventh bit (adding or subtracting 64 modulo 64). Thus, control-A is written as #A and control-B as #B. The prefix character is also used to prefix itself: ##. Upon input, the reverse transformation is performed. Printable characters are not transformed. The assumption is that most files to be transferred are printable, and they have a relatively small number of control characters. Under this assumption the packet is not significantly lengthened by quoting. For binary files the average quoting overhead will be 26.6 percent more characters if all bit patterns are equally likely, since the characters that must be prefixed (the control characters, plus DEL and # itself) comprise 26.6 percent of the ASCII alphabet. The special prefix character (~) indicates that the next character is a repeat count (a single character encoded printably) and that the character after that (which may also have control or eight bit prefixes) is repeated so many times. As for example, ~} A indicates a series of 93 letter A's.

In order to keep the protocol simple, no other transformations are done.

Standard, base-level Kermit employs a single-character arithmetic checksum, which is simple to program, low in overhead and has proven quite adequate in practice. The sum is obtained by adding together the ASCII values of all the characters in the packet, excluding the SOH and the checksum itself, and including any

prefixing characters. Even non-ASCII hosts must do this calculation in ASCII. The result can approach 12,000 in the worst case, the binary representation of which is 10111011100000 (binary) -- a 14-bit number. This is more than one character's worth of bits. Since every character in the checksum will contribute to the low-order 7-bits of the sum, some high-order bits can be discarded and a viable validity check still be made.

The Kermit protocol also allows other block-check options, including a two-character checksum and a three-character 16-bit Cyclic Redundancy Code. The two-character checksum is simply the low order 12 bits of the arithmetic sum broken into two printable characters. Care must be taken in the formation of the single-character block check. Since it must be expressed as a single printable character, values of the high-order data bits may be lost, which could result in undetected errors, especially when transferring binary files. So, the seventh and eighth bits of the sum are extracted and added back to the low-order bits; if the arithmetic sum of all the characters is S, the value of the single-character checksum is given by

(S + ((S AND 300)/100)) AND 77

The numbers are represented in octal. Here the checksum is formed with the contribution of every bit from every character in the packet.

PACKET LAYOUT-- Kermit packets have the format, as shown in

Figure II-2 where all fields consist of ASCII characters. The char function converts a number in the range of 0-94 to a printable ASCII character by adding 32. In terms of the seven layer ISO (International Organization for Standardization) network reference model, 8-bit bytes are presented to Kermit by the hardware and operating-system software comprising the physical-link layer. Correct transmission is ensured by the packet-level routines that implement the data-link layer using the outer "skin" of the packet --the MARK, LEN and CHECK fields. The network and transport layers are moot, since Kermit is a point-to-point affair in which the user personally makes all the necessary connections. The session layer is responsible for requesting retransmission of the missing packets or ignoring the reduntant ones based on the SEQ field: the presentation layer is responsible for any data conversions (EBCDIC/ASCII, insertion or stripping of CR/LFs etc.). Finally, the TYPE and DATA fields belong to the application layer.

The six fields of a Kermit information packet are shown in Appendix E. The packet may be followed by any line terminator required by the host, a carriage return by default. Line terminators are not part of the packet and are not included in the count or checksum. Terminators are not necessary to the protocol and are invisible to it, as are any characters that may appear

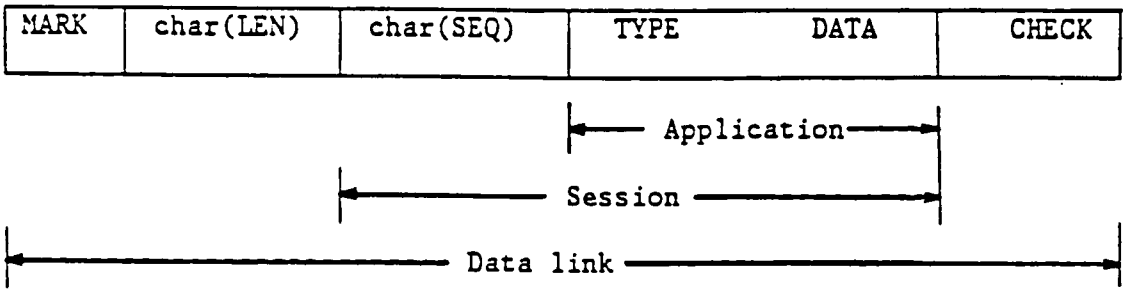


FIGURE II-2. Format of an Information Packet in Kermit Protocol.

between packets. If a host cannot do single character input from a terminal, a terminator will be required for that host.

Some sample Kermit data packets are shown in Figure II-3. The ~A represents the unprintable SOH (or control-A) character. In the last packet shown, E is the length, the ASCII value of which is 69, less 32 (the unchar transformation, which is the opposite of char) gives a length of 37. The next character,"&", tells the packet sequence number, in this case 6. Next is the packet type D for data. The next characters "of# M#Jconstructing a theory conta", form the actual data. The final character 5 is the checksum, which represents the number 21.

Finally, the asynchronous serial communications used by the Kermit protocol can accommodate a variety of diverse computer systems and their different ways of handling information and files. Kermit sets minimum transmission standards by providing a common subset of the machines' features. These features include transfer of the filename and contents of both textual and binary files, different error detection methods, and time-out facilities if either end of the communication link experiences delays or difficulties.

```
~AE"D  Nocelestial body has required J
~AE#Das much labor for the study of its#
~AE$D#M#Jmotion as the moon. Since ClaA
~AE%Dirault (1747). who indicated a way7
~AED of#M#Jconstructing a theory conta5
```

FIGURE II-3. Listing of Some Sample Packets of Information in the Kermit Protocol.

States and Transitions, Heuristic Rules and Examples of Kermit

Kermit file-transfer protocol was first developed at Columbia University for downloading files from mainframes to microcomputers, but it has evolved into a comprehensive communication system for transferring data between numerous types of computers. Kermit is inexpensive and reliable. It uses an RS-232C asynchronous serial communications to exchange packets of information, and has error checking for reliable file transfer.

By defining a common subset of different computer's communication capabilities, Kermit sets up the minimum standards for communication between two systems. Numerous advanced features and options make Kermit powerful and flexible; many parameters can be changed, and it can even be set up as a simple network file server. Kermit is a portable protocol that is compatible and has already been implemented on many operating systems and brands of computers.

STATES AND TRANSITIONS OF THE KERMIT PROTOCOL : The Kermit protocol can be described as a set of states and a set of transitions that define, for a given event, what action to take and what new state to change to. The fact that each packet must be acknowledged, reduces the number of states and actions and the amount of state information that must be maintained to a minimum. The state diagram for Kermit receiving a file is shown in Figure II-4.

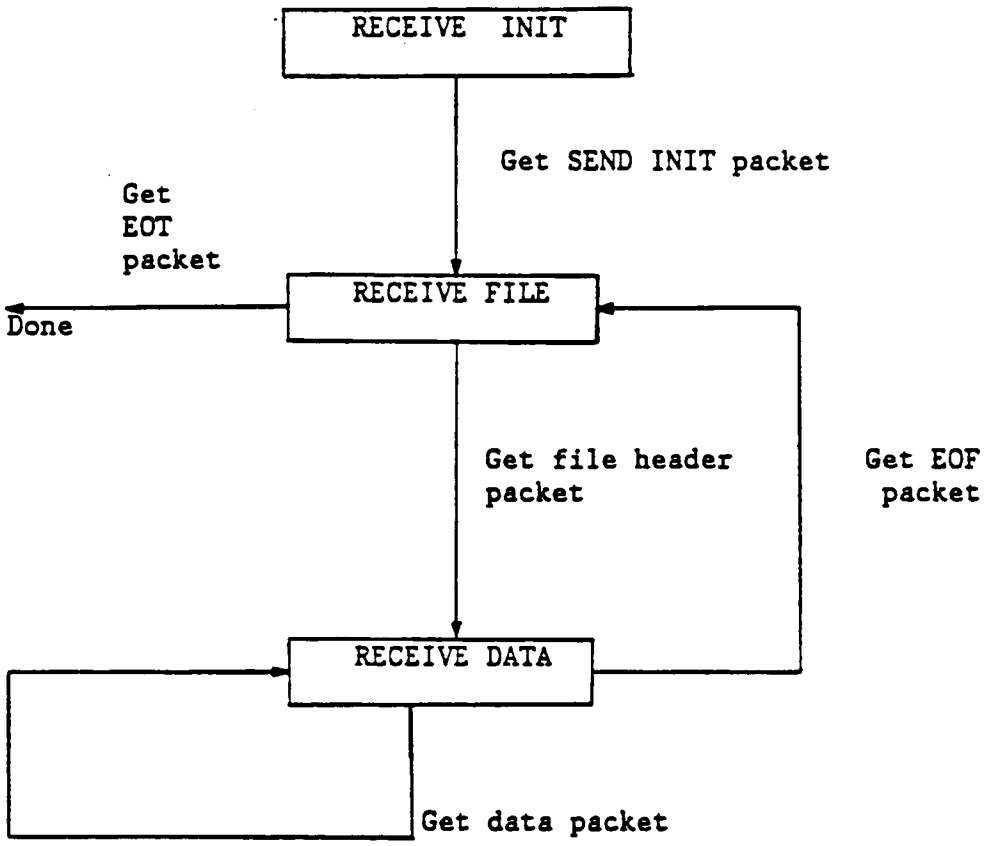


FIGURE II-4. A Simple State Diagram for Kermit Receiving Files.

The receiver is initially in the RECEIVE INIT state and waits for the other side to send a SEND INIT packet. If any other packet is received or the SEND INIT packet is not received on time, a NAK (negative acknowledge) is sent. Time-outs or NAKs can occur up to a threshold, which when exceeded causes the protocol to assume that the connection has become unusable. When the SEND INIT arrives, the state changes to RECEIVE FILE. Now, Kermit waits for the File Header packet containing the name of the file which is to come. When the File Header arrives, Kermit opens a new file using the name provided. The filename may be transformed to suit local naming conventions or to avoid a name collision. The state is now changed to the RECEIVE DATA state.

Kermit then receives the contents of the file until an EOF (end-of-file) packet arrives. When an EOF packet is received, Kermit switches back to the RECEIVE FILE state. If another file is to be sent, another File Header packet will follow, otherwise an EOT (end of transmission) packet will terminate the transfer. The distinction between EOF, EOT and the File Header itself, allows files to be sent in groups. EOF marks the end of a file; EOT indicates the end of a group. This distinction also allows the two sides to disconnect cleanly: the EOF must be acknowledged before the sender will believe the file has been transmitted correctly; the EOT will follow, but if the ACK (acknowledge) sent in response to the EOT is lost, no harm will

be done, since both sides are terminating anyway. For simplicity, some transitions are not shown in Figure II-4.

*If a bad packet or time-out occurs in any state, a null transition back to the same state occurs with a NAK for the expected packet.

*In any state, an error may occur that can cause the transfer to terminate. For example, the target disk might fill up. The side that encountered the error sends an error packet containing an informative error message and quits. Upon receipt of the error packet, the other side displays the message on the screen (if it is in control of the screen) and also quits.

*Actions that are taken on each transition, such as opening a file when a File Header packet is received, are not shown. Each packet successfully received is acknowledged with an ACK.

The state transitions for a sending Kermit are similar to that of Figure II-5. In each state, instead of waiting for particular packet types, Kermit sends the appropriate packet and waits for an ACK. If the ACK does not arrive within the allotted time, or a NAK appears instead of an ACK, the same packet is retransmitted. The send operation starts with a SEND INIT packet; it includes one or more files starting with a File Header, followed by one or more data packets and then an EOF. When all the specified files have been sent, an EOT packet closes the connection and the operation is terminated.

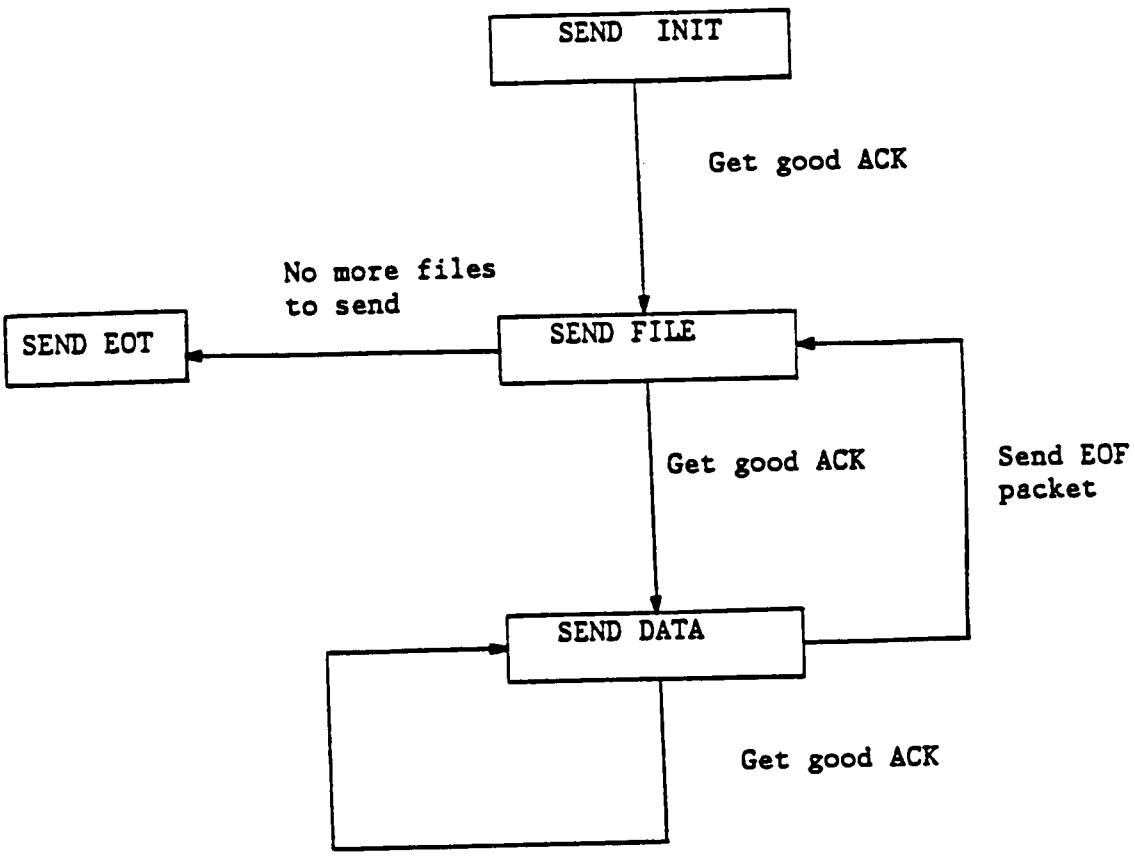


FIGURE II-5. A Simple State Diagram for Kermit Sending Files.

Base-level Kermit provides that during any particular transaction the sender is the master and the receiver is the slave. These roles may be reversed in the next transaction; any Kermit implementation is capable of acting as either the master or the slave.

INITIAL CONNECTION: To allow a diverse group of computers to communicate with one another an exchange takes place during the initial phase of connection, when the two sides configure with each other. The sending Kermit contains setup parameters in its SEND INIT packet and the receiving Kermit responds to it with an ACK packet. Figure II-6 shows the DATA field of the SEND INIT packet. "I" and "you" are used to distinguish between the two sides. Fields are encoded printably using the char function unless indicated otherwise.

*MAXL-- The maximum-length packet I want to receive. This is a number, the maximum value of which is 94. You respond with the maximum you want me to send. This allows systems to adjust to each other's buffer sizes or to the condition of the transmission medium.

*TIME-- The number of seconds after which I want you to time me out while waiting for a ecket from me. You respond with the amount of time I should wait for receiving packets from you. This allows the two sides to accommodate different line speeds or other factors which may cause some timing problems.

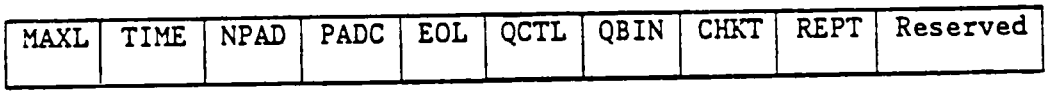


FIGURE II-6. The Set-up Parameters in the DATA Field of the SEND INIT Packet of the Kermit Protocol.

*NPAD-- The number of padding characters I want to precede each incoming packet; you respond in kind. Padding may be necessary for a half-duplex system that requires some time to change the direction of transmission.

*PADC-- The control character I need for padding, if any, modified by XOR with 64 to make it printable. You respond in kind. Normally the null character (NUL) is used; some systems use the delete character (DEL). This field is ignored if the value NPAD is zero.

*EOL-- The character I need to terminate an incoming packet, if any. You respond accordingly. Most systems use a carriage return as a line terminator.

*QCTL-- The printable ASCII character I will use to quote control characters or prefix characters. Normally '#' is used for this purpose. You respond with the one you will use.

*QBIN-- The printable ASCII character I will use to quote (prefix) characters that have the eighth bit set, for transmitting binary files when one or both systems cannot use the parity bit for data. Since this kind of quoting increases both processor and transmission overhead, it is normally to be avoided.

*CHKT-- Check type, the method for detecting errors. 1 for single-character checksum (the normal method), 2 for two-character checksum, 3 for three-character CRC-CCITT. If your response

agrees, the designated method will be used; otherwise, the single-character checksum will be used. Other check types may also be added.

*REPT-- The prefix character I will use to indicate a repeated character This can be any printable character other than blank (which denotes no repeat count prefix), but ~ recommended. If you do not respond identically, repeat counts will not be done. Groups of four or more identical characters may be transmitted more efficiently using a repeat count, although an individual implementation may wish to set a higher threshold.

*CAPAS-- An extendable bit mask encoded printably to indicate whether certain advanced capabilities, such as file-attribute packets, are supported.

*Reserved fields-- The next four fields are reserved for future use. Those wishing to add their own parameters to the initial connection exchange should start at the fifth field after the capability mask in order to remain compatible with other Kermit programs.

If the trailing fields in the DATA field are ommitted, they will assume some appropriate defaults. Defaults for intermediate fields can be selected by setting those fields to blanks. Every parameter has an appropriate default. In fact, the entire DATA field of the SEND INIT packet or its ACK may be left empty to accept all the default values. The SEND INIT mechanism preserves

compatibility from the earliest Kermit to the most recent. There is no protracted negotiation. Everything must be settled in a single exchange of the SEND INIT packet. Some parameters are outside the scope of this exchange, and must be set even before the first packet is sent. For example, if the receiving side can read characters only with odd parity while the sending computer sends them with even parity, the SEND INIT packet will never arrive successfully. In such cases, some initial commands may have to be given to inform one or both the Kermits about the vagaries of the other system. Another example is the packet terminator (EOL). If the receiving Kermit requires one that the sending Kermit does not know about the SEND INIT will never be realized. For these reasons, most implementations of Kermit provide SET commands for all the parameters listed above and some others as well.

RULES AND HEURISTICS: While transferring a file, one Kermit sends information packets--file headers, data and so forth-- and the other side sends only ACKs or NAKs in response. The most important rule in the Kermit protocol is WAIT FOR A RESPONSE BEFORE SENDING THE NEXT PACKET. This prevents buffer overruns and allows participation by half-duplex systems. This does not mean that the Kermit should wait forever; a time-out should occur after a few seconds if the desired packet has not arrived. Upon time-out a sending packet retransmits the current packet; a receiving Kermit

reacknowledges the current packet or negatively acknowledges the expected one.

Some interesting heuristics are used in Kermit to boost efficiency and improve error recovery. A number of important rules take care when packets are lost during transmission.

*A NAK for the next packet implies an ACK for the current packet. A NAK with packet number (n+1) implies that the receiving Kermit got packet n, sent an ACK that was never received, and, not knowing that the ACK didn't get through, is now waiting for packet (n+1), which was never sent. In this case, you simply need to send packet (n+1). An important exception is when the missing ACK is for a SEND INIT packet. The next rule, ACKNOWLEDGE AND DISCARD REDUNDANT PACKETS, handles the situation where the same packet arrives again. The sending Kermit timed out waiting for an ACK that was sent but lost and retransmitted the packet. The receiver must discard the redundant data and acknowledge the packet again; to do otherwise could have undesirable effects, such as adding the same data to a file twice or opening the same file multiple times. The situation resulting from a lost ACK depends upon which side times out first.

Kermit must handle another situation arising from possible lost packets: NEGATIVELY ACKNOWLEDGE THE EXPECTED COMMAND. The potential problem occurs in either the RECEIVE INIT state or when a Kermit server is waiting for a command. In either case, Kermit

won't know whether communication has begun if the other side's initial packet was lost. Kermit can't assume the other side will time out and retransmit; so, it must check periodically by sending a NAK for packet zero. If the Kermit does not get any response, then it assumes that nothing has happened and goes back to sleep for a while. But sending periodic NAKs opens the door to the buffering problem. Some systems buffer input for a device; when a program isn't looking for input some or all the input is saved by the computer for future requests. This can cause problems when talking to a Kermit server or sending a file to a Kermit in receive state. If some time has elasped since activating the remote Kermit sending the file and escaping back and starting up the local Kermit, a number of NAKs may have accumulated in the local Kermit's input buffer, so CLEAR THE INPUT BUFFER AT THE BEGINNING OF A TRANSFER.

If the input buffer is not cleared, the local Kermit will think the remote side is having trouble receiving the first packet. In an effort to get the packet through, it will be sent again; this repeats for every NAK waiting in the input buffer. By the time the first ACK is finally encountered in the buffer, a number of duplicates of the first packet will have been sent out. If this number exceeds the NAK threshold, the connection will be broken. If not, however, the second packet will be retransmitted once for each of the extra ACKs the remote Kermit correctly sent for

the duplicate first packets. This can continue throughout the file transfer, causing each packet to be sent many times. So, CLEAR THE INPUT BUFFER AFTER READING EACH PACKET. Any computer that buffers its input clear its input buffer before the transfer and after each packet that arrives successfully. But since not all systems provide a clear-buffer function, the last rule may be added: SCARD REDUNDANT ACKs.

EXAMPLE OF A FILE TRANSFERRING TECHNIQUE: A sequence of packets from a real file transfer is shown in Figure II-7. Each packet starts with control-A, shown as ~A. In the first packet control-A is followed by the packet length ")" (41 less 32, or 9), the packet type, S (SEND INIT), and then by the appropriate parameter declarations: maximum packet length is H (72 -32 =40), time-out is " (" (40-32 =8), number of padding characters is 0 (space =32-32 =0), the padding character is 0, end-of-line is "-" (45 -32= 13, the ASCII value of a carriage return), the control quote character is "#", and the remaining fields are ommitted, defaulting to appropriate values. The final character ("^") is the single-character checksum, computed as follows (all numbers and computations are in octal, and "sp" represents a space):

```
) sp S H  ( sp @ -                    #
51 + 40 +123 + 110 + 50 + 40 + 100 + 55 + 43 =674
```

```
~A) SH( @-#~                                    SEND INIT
~A) YH( @-#%                                    ACK for SEND INIT
~A)+|FMOON.DOC2                                 File header
~A#|Y?                                          ACK for file header
~AE"D No celestial body has required J          First data packet
~A "Y@                                          ACK for first data
                                                packet
~AE Das m%%%uch labor for the study of          Second data
 its                                            packet,bad
~A##N8                                          NAK for second data
                                                packet
~AE#Das much labor for the study of its           Second data packet
                                                  again
~A##YA                                          ACK for second data
                                                packet
~AE$D#M#Jmotion as the moon. Since ClaA           ETC...


(many packets omitted here)


~AD"Dout 300 terms are sufficient.#M#JU           Last data packet
~A#"Y@                                          ACK for last data packet
~A##ZB                                          EOF
~A##YA                                          ACK for EOF
~A#$B+                                          EOT
~A#$YB                                          ACK for  EOT
```

FIGURE II-7. Listing of a Sequence of Packets from an Actual File Transfer (Comments are on the right side).

674 + (674/300) = 676

676 AND 77 =76;

char (76) =76+40 =136 = " "

The receiver acknowledges with its own parameters, which are the same. Then comes the file header, the file, EOF and EOT. One data packet was corrupted by a burst of "%" characters, negatively acknowledged, and retransmitted.

The User Interface

Kermit was designed from a mainframe perspective. Like many mainframe programs, Kermit issues a prompt, the user types a command, Kermit executes the command and issues another prompt, and so on until the user exits from the program. Much care is given to the command perser, even on microcomputer versions. The goal is to provide English-like commands composed of sequences of keywords or operands, with abbreviations possible for any keyword in any field down to minimum unit length and with "?" help available at any point in a command.

The basic commands are SEND and RECEIVE. These allow most Kermits to exchange files. Operands can be the name of a single file or a file-group designator to transmit multiple files in a single operation.

The CONNECT command provides the mechanism for logging in and typing commands at the remote host, which is necessary in order to start the Kermit on that side.

When two systems are dissimilar, a SET command is provided to allow them to accommodate each other's peculiarities, for instance, SET PARITY ODD to add odd parity to all outbound characters or SET LOCAL ECHO to do local echoing when connected as a terminal to a half-duplex system. The SET command must sometimes be used to supply information to the target system on how to store an incoming file with respect to block size, byte size, record format, and record length.

Most Kermit implementations take special care to reassure the user during file transfer. The names of the files being transferred are shown, and a dynamic display is made of the packet traffic, showing successful transmission of packets as well as time-outs and retransmissions. Messages are issued when the user connects to the remote system or escapes back from it, Kermit prompts identify the implementation. Helpful error messages are displayed when necessary; these may emanate from either the local or the remote system. The final disposition of the transfer is clearly stated: complete or failed.

The actions required of the Kermit user depend upon thr degree to which the Kermit programs involved have implemented the specification. Minimal implementations require that the user

connect to the remote host, start Kermit there, issue a SEND (or RECEIVE) command, escape back to the local machine, and issue the complementary RECEIVE (or SEND) command. All this must be done for each transfer. More adnanced implementations allow the remote side to run as a server and to take all its instructions in special command packets from the local Kermit; all the user has to do on the remote end is make the initial connection to start the server. The server will even log itself out upon command from the local Kermit. A minimal server can process commands to send files, receive files and shut down.

The need for a cheap, convenient file transfer capability among diverse systems is pressing, and many efforts are under way at many places. Kermit is a much simpler protocol compared to the more complicated ones used in real networks or when integration of microcomputer and mainframe is a major goal. The Kermit protocol has proven successful and continues to grow in popularity. As of today, implementations exist for over 50 computer systems. No single implementation necessarily includes all features mentioned in this article, but all are able to communicate at least at base level. Requests for more information about Kermit can be addressed to:

Kermit Distribution

Columbia University Center

for Computing Activities

7th Floor. Watson Laboratory

612 West 115th St.

New York, NY 10025

The Receiving Kermit for the TI-990/12 minicomputer

Kermit is a file-transferring program allowing movement of files among variety of computer systems including micro-mainframe and mainframe-mainframe. It is not a portable program but a portable protocol. So, to invoke Kermit to a system, a compatible Kermit program has to written for that system. The TI-990/12 minicomputer installed in the Department of Electrical Engineering of UTK, is a multi-user system with 10 workstations. Although a file-transferring mechanism (other than the Kermit) has been developed by the author between the TI-990 minicomputer and the DEC-10 mainframe, a considerable time of the research was spent in writing a compatible Kermit program for the TI machine. More work is necessary in order to make the receiving Kermit for the TI totally compatible with the DEC-10 Kermit. Even though the Kermit program written for the TI machine is presently unable to transfer files, the following information may be valuable for future research towards attaining that goal.

The receiving Kermit program is written in the AMPL (Advanced Microprocessor Prototyping Laboratory) language of TI-990/12 minicomputer. The program is well documented. All procedures and functions within the program have been tested to work fine, with the exception of procedure named "SPACK". SPACK is responsible for sending a packet to the other computer. SPACK works fine in all modes of the Kermit other than its packet mode. A solution to this problem may render the TI Kermit working. In order to invoke the receiving Kermit on the TI minicomputer, the following sequences of operation are performed:

(1) Go to the AMPL utility of TI-990/12 minicomputer by typing AMPLDX on the VDT-911 (TI Terminal) and hitting <ret) twice.

(2) Type on the TI terminal:

RSTR('AHS.AMPL.KERMIT') <ret>

INIT <ret>

(3) Log on to the DEC-10 terminal using the modem and the serial asynchronous telecommunication line. Any terminal which is compatible with RS-232C standard, can be considered to be a DEC-10 terminal.

(4) Type on the DEC terminal:

R Kermit <ret>

"Kermit-10>" will appear as the prompt for Kermit on the DEC terminal.

(5) Type on the DEC terminal:

SEND <filename> <ret>

(6) Type on the TI terminal:

KERMIT <ret>

(7) Wait on the TI terminal for the message "RECEIVE OVER".

Hit <cmd> when you are through. You can terminate the Kermit running on the TI-990 minicomputer any time by hitting <cmd>. Next time when you want to invoke Kermit on the TI machine (You are still inside the AMPL utility of TI), type "RECSW" instead of "Kermit" and wait for the message "RECEIVE OVER". But, if you terminate the Kermit program running on the TI, and then come back to the same by typing "RECSW", you will lose all the characters that has been received from the DEC-10 before RECSW is typed. The file AHS.AMPL.STORE contains the decimal values of all the ASCII characters received from the DEC-10, including the control characters.

(8) Go out of TI-990 AMPL utility by typing "EXIT".

(9) Check the contents of the file AHS.AMPL.STORE [use SF (Show File) command] so that it does not contain anything other than decimal data. If it does, you need to delete those characters before you can use the file. Run a PASCAL program on the TI to reconstruct the original file sent by the DEC from the datas

available in the file AHS.AMPL.STORE. Proceed as follows: Type on the TI terminal--

```
[]XPT <ret>
EXECUTE TI PASCAL TASK
  PROGRAM FILE :  AHS.PASCAL.RECVLINK <ret>
  TASK NAME :  RECEIVE <ret>
  INPUT :  AHS.AMPL.STORE <ret>
  OUTPUT :  <ret>
  MESSAGES :  ME <ret>
  MODE(F,B,D) :  FOREGROUND <ret>
  MEMORY :  1,1 <ret>
```

"OUTFILE" is the actual file received from the DEC-10.

LIMITATIONS OF THE RECEIVING KERMIT PROGRAM AND THE POSSIBLE REMEDIES: (A) The AMPL utility of the TI machine has about 9K bytes of user memory So, it allows about 8000 characters to be received at one time from the DEC. But the operations (1), (2), and (6) through (10) can be repeated as many times as necessary without terminating the Kermit running on the DEC. The reason Kermit is not terminated on the DEC is because the DEC-10 continues to send the same data packet to the TI until the DEC has received an ACK packet from the TI or that the time-out, as agreed upon by both the systems, has elasped.

(B) The AMPL utility is not always fast enough to work at a baud rate of 300 bits/second (b.p.s). In other words, while reading it may lose some characters sent by the DEC. After the file AHS.AMPL.STORE is opened by the Kermit program running on the TI machine, everything that appears on the TI screen is stored into the file AHS.AMPL.STORE. So, storing a character sent by the DEC-10 into the TI memory involves two operations--reading the character from the asynchronous serial telecommunication line and then displaying the character on the TI screen in order to save it into the file. If reading and displaying a character are done simultaneously then it has been found that in every data packet the DEC-10 sends, one or two characters are always lost from that packet. As a result, the TI calculates a faulty checksum, and when this is repeated several times, the Kermit programs running on both machines are automatically terminated. This problem has been overcome by following different techniques of storing characters into the file. Every time DEC sends a data packet, all the characters in the packet are first read without displaying them simultaneously. When all the characters in the packet have been read, they are then saved into the file by displaying them on the screen. Following this technique, none of the characters are lost while reading from the serial line, and the Kermit proceeds accordingly.

The Kermit program written in AMPL language of TI-990/12 minicomputer is shown in the Appendix A.

Actual Receiving Protocol on TI-990/12 Minicomputer

The receiving protocol between the TI-990 minicomputer and the DEC-10 mainframe is shown by the flowcharts of Figures II-8 and II-9. If a data packet is not received correctly by the TI-990 model computer, the DEC sends that packet again, and the process continues until the packet has been received correctly by the TI-computer. A checksum character and an end-of-packet character (ASCII character with decimal value 3) follow the actual data packet. Whenever an end-of-line is read, it is replaced by an ASCII character of value 2. The following flowcharts will help illustrate the various phases of the routines working interactively between the two diverse computer systems during reception.

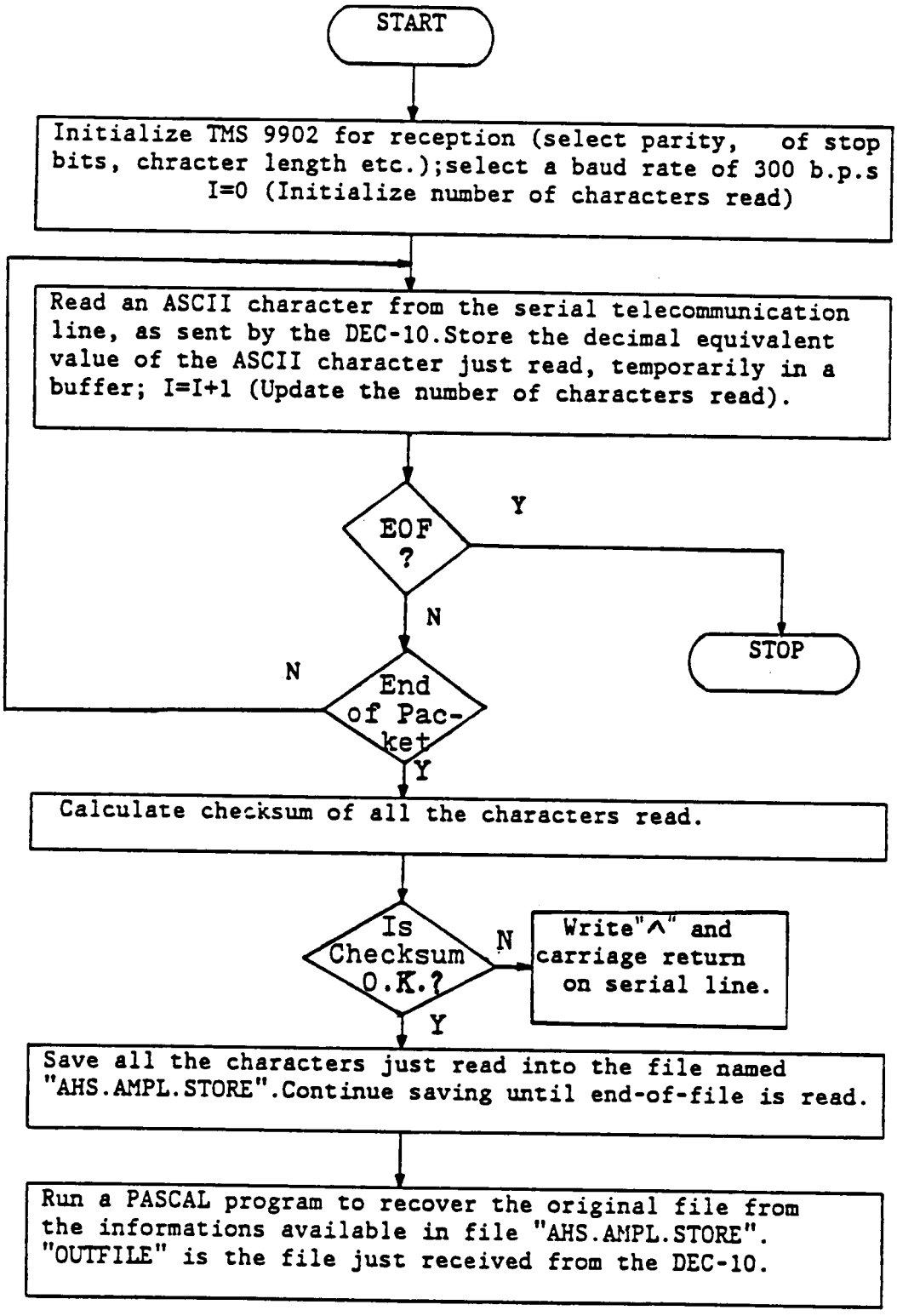


FIGURE II-8. Flowchart for the Actual Receiving Protocol on TI-990.

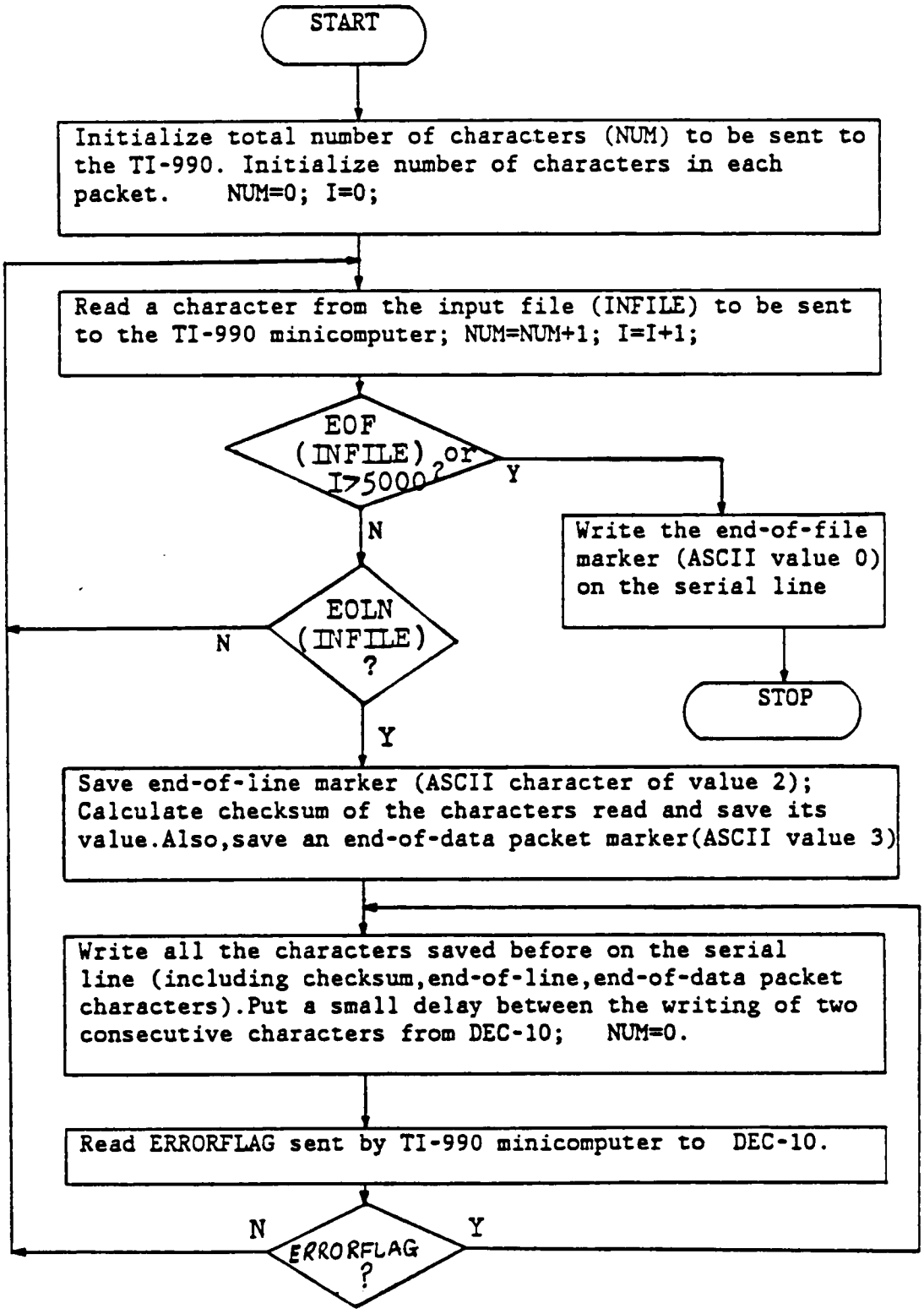


FIGURE II-9. Flowchart of the Actual Receiving Protocol on DEC-10 .

CHAPTER III

HARDWARE DESIGN CONSIDERATIONS

The chapter begins with a description of the hardware interfaces necessary for the TI-990/12 minicomputer to communicate with other computers. The rest of the chapter mainly deals with the information necessary to program the Four Channel Communications Controller (FCCC), using one of the asynchronous protocols supported by the controller. The program is given in Appendix B.

Hardware Interfaces

Following are the various hardware options available to the TI-990 minicomputer for communication with other computers:

(1) USING AMPL UTILITY OF THE TI-990/12 MINICOMPUTER: The only communication medium common to the TI-990 minicomputer and the other computer is the asynchronous serial telecommunication line. Standards for this medium are almost universally followed -- connectors, voltages and signals (EIA RS232-C); character encoding (ASCII ANSI X3.4-1977). Dial-up connection is initiated manually from the laboratory of the Electrical Engineering Department using an inexpensive acoustic coupler. The baud rate selected for the line is 300 b.p.s., and the communication is full duplex. However, there is one difficulty with the VDT-911 (Video Display Terminal), used by the TI-990/12 minicomputer. The TI

terminal is not compatible with the RS-232C standard. But the TI-990 minicomputer can communicate with a TM 990/U89 microcomputer, which has an on-board TMS 9902 -- the asynchronous communication controller (ACC). The TMS 9902 is TTL-compatible on all inputs and outputs, including the power supply (+5V) and single phase clock. The TMS-9902 ACC provides an interface between the TM990/U89 microcomputer board and a serial asynchronous communacations channel; it accepts EIA standard RS-232C protocol.

The DEC-10 terminal is interfaced with the VDT-911, as shown in the Figure III-1. Since the line access is full-duplex, any character typed on the DEC-10 terminal (or whatever character appears on the DEC screen), is echoed back to the RS-232C port connected to the TM-990/U89 kit, which is readily available for reading by the TI-990/12 minicomputer. Also, any character sent by the TI to the DEC-10 appears on the DEC screen.

(2) USING FCCC: The FCCC of TI-990/12 minicomputer is able to interface up to four EIA RS-232C/423 compatible communications devices to a model 990 computer with TILINE interface. Figure III-2 shows the interconnection between the TI-990/12 minicomputer and the communications devices that can be supported. The pin connections between any channel (either P3, P4, P5 or P6 connector) of the FCCC and the DEC-10 terminal are the same as those of Figure III-1.

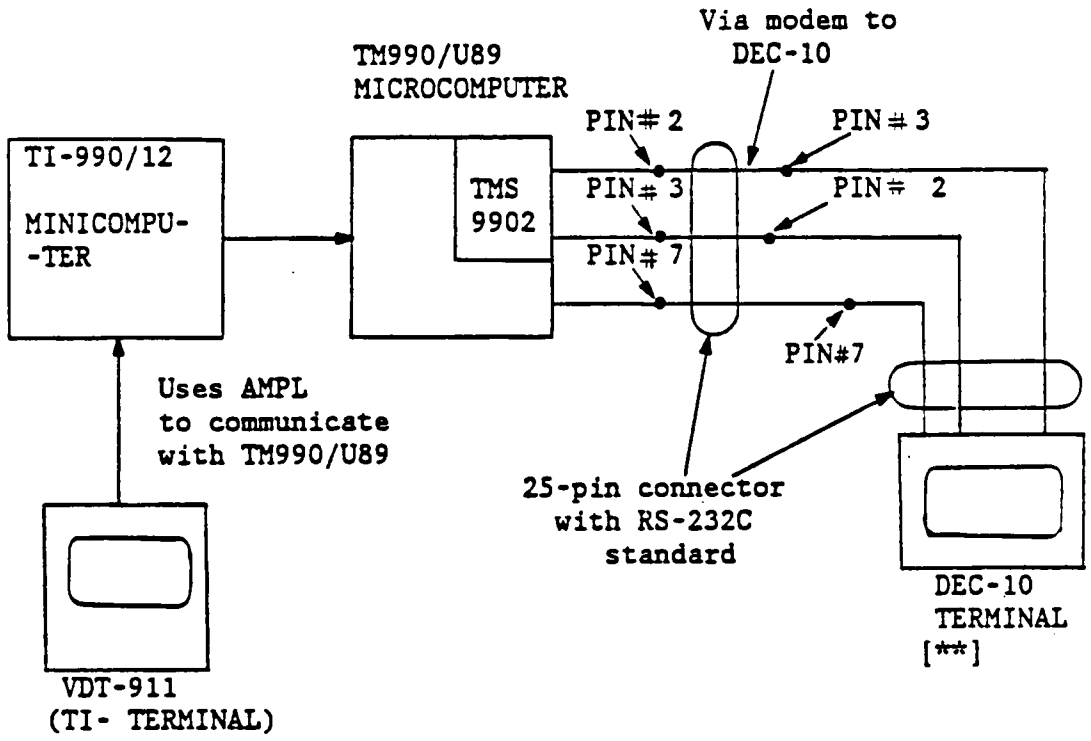


FIGURE III-1. Hardware Interface between VDT-911 of TI-990/12 Minicomputer and the DEC-10 Terminal.

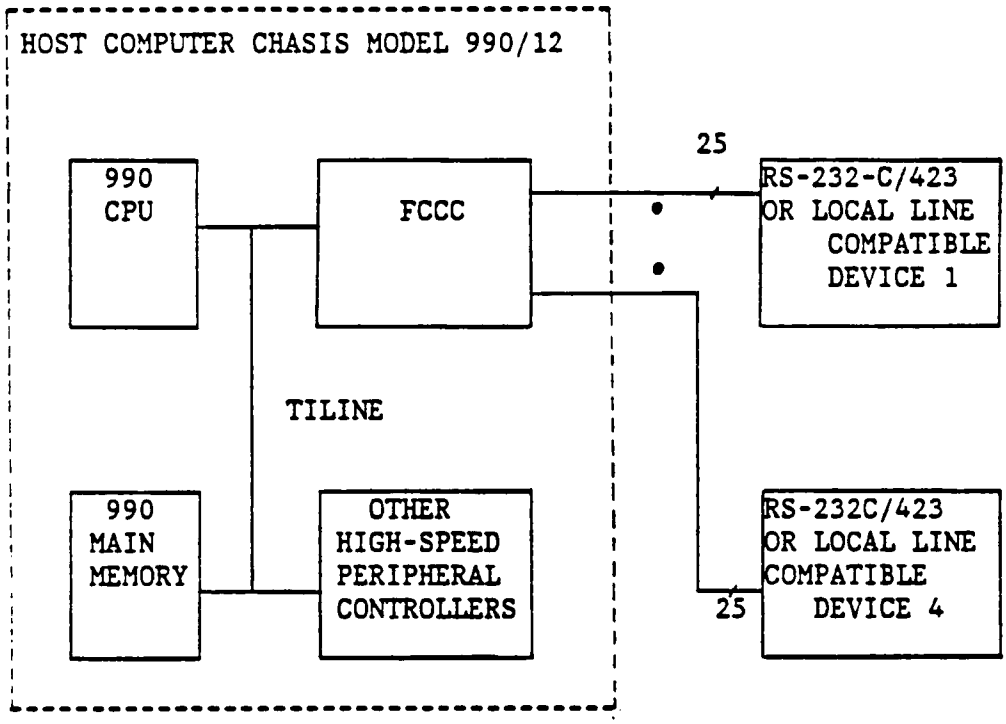


FIGURE III-2. FCCC System Block Diagram.

Hardware Functional Description of FCCC

Figure III-3 is a simplified block diagram of the FCCC, which shows the interconnections among the major components of the FCCC. The host computer (TI-990/12 minicomputer) passes its commands and requests through the TILINE bus to the FCCC, where they are processed by the slave processor (TMS 9900 microprocessor) on the FCCC board. Routines coded into the on-board ROM of the FCCC handle all the commands and requests made by the host computer to the FCCC. Routines contained in the FCCC board include:

* Control programs for the TILINE interface.

* Control programs for the TMS 9903 Synchronous Communications Controller.

* Self-test programs and diagnostics to check operation of the hardware on the FCCC board.

* Message blocking, error detection and CRC generation routines for supporting the various communications protocols.

Processing of the host computer request may include transferring data to or from host memory (TI-990 memory), transmitting data to or receiving data from the remote communications devices through the TMS 9903 synchronous communications controller (SCC), setting up a communications channel to support a certain protocol, or downloading code from the

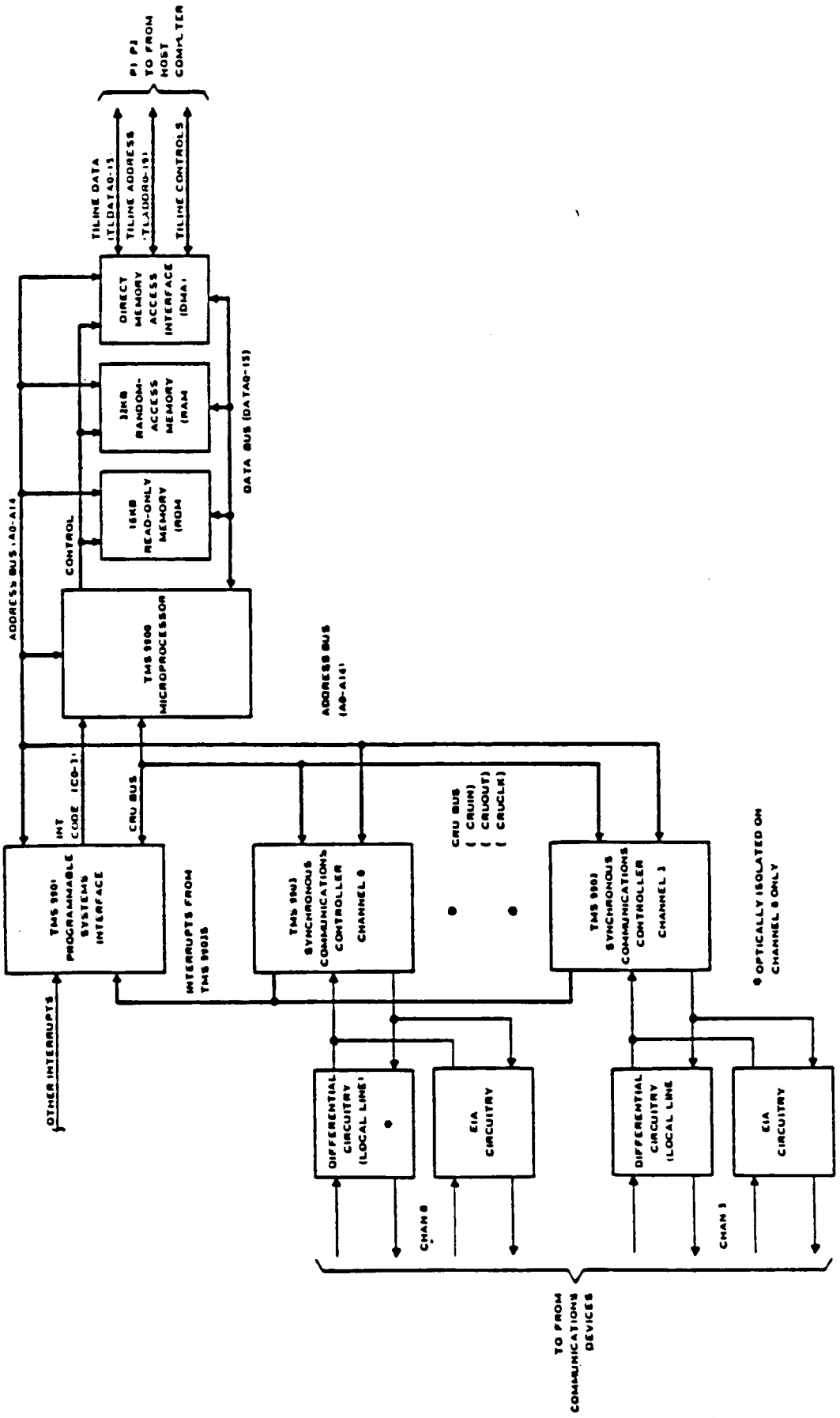


FIGURE III-3. FCCC Functional Block Diagram.

host to the FCCC memory. The slave processor transfers data from host memory to FCCC memory or vice versa by using the DMA interface logic.

Software Description for a Typical Request by the Host

In order to pass a request to the TMS 9900 microprocessor of the FCCC, the host computer (TI-990/12 minicomputer) writes the four write-only- slave words (WOSW0 through WOSW3) to the FCCC. The host computer writes information that the TMS 9900 will need to process the request in the four WOSWs.

WOSW0 is implemented in the hardware on the FCCC, and WOSW1 through WOSW3 are written into the TILINE RAM, a dedicated 16-word static RAM on the FCCC board. Figure III-4 shows the organization of the 16-word space of the TILINE RAM. As soon as the host computer finishes writing into the WOSW3, it creates an interrupt to the TMS 9900 microprocessor and causes the microprocessor to read the WOSWs. Each time a request is made by the host computer through the WOSWs (which specifies the conditions of the request), the FCCC stores the request information in a data structure called a communications request block (CRB) in FCCC memory.

When all the WOSWs have been written, it initiates the microprocessor to branch to the proper routine (stored in ROM) to process the request. If a write request is initiated for

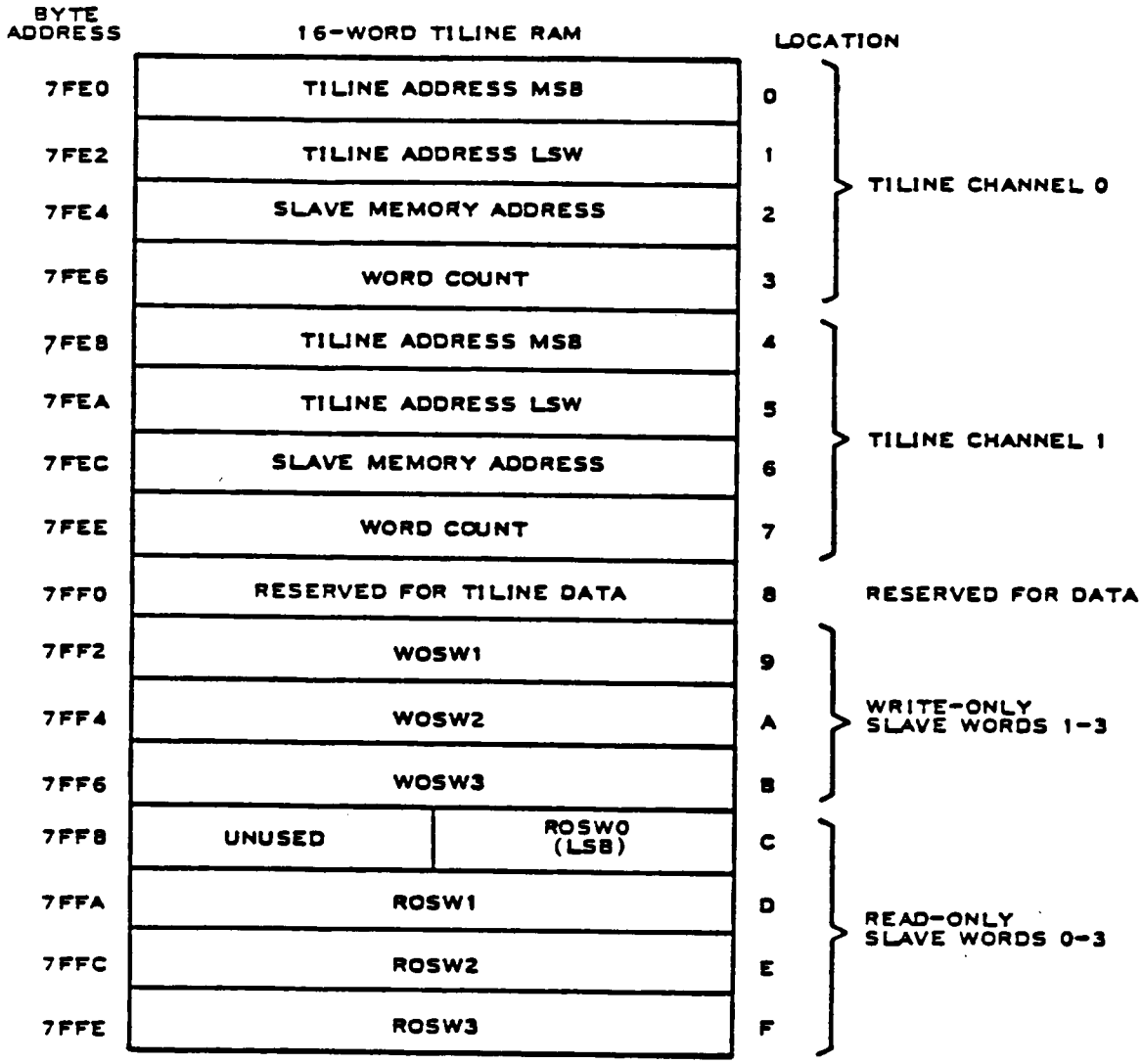


FIGURE III-4. TILINE RAM Utilization.

transmitting data to one of the external communications devices on communications channels 0 through 3, then the information like the location of the data to be transmitted along with the channel, etc. are sent to the microprocessor in the WOSWs. For receiving (reading) data from one of the communication devices, the information regarding the location of the data to be stored in the host memory (TI-990 memory), has to be given in the WOSWs.

The microprocessor initiates a DMA transfer to transfer the data from host computer TILINE memory to FCCC memory using the TILINE master channel 0 or 1. Assume that the TILINE master channel 0 is used for a write request. To initiate the write request, the information needed are the location in host computer memory of the data to be transferred, the location in FCCC memory where the data is to be placed and the count of data words to be transferred. The microprocessor writes all these information into the TILINE master channel 0 location of the TILINE RAM. Data words are then transferred from data buffers in host computer memory to a location in the TILINE RAM, and then to slave data buffers in FCCC dynamic RAM one 16-bit word at a time until the transfer is complete.

There are level interface tables in the ROM on the FCCC to guide the processing of host requests. The line control status table (LST) controls the processing at the highest level; processing at the lowest level is controlled by the interrupt service table (IST). Figure III-5 shows a software functional

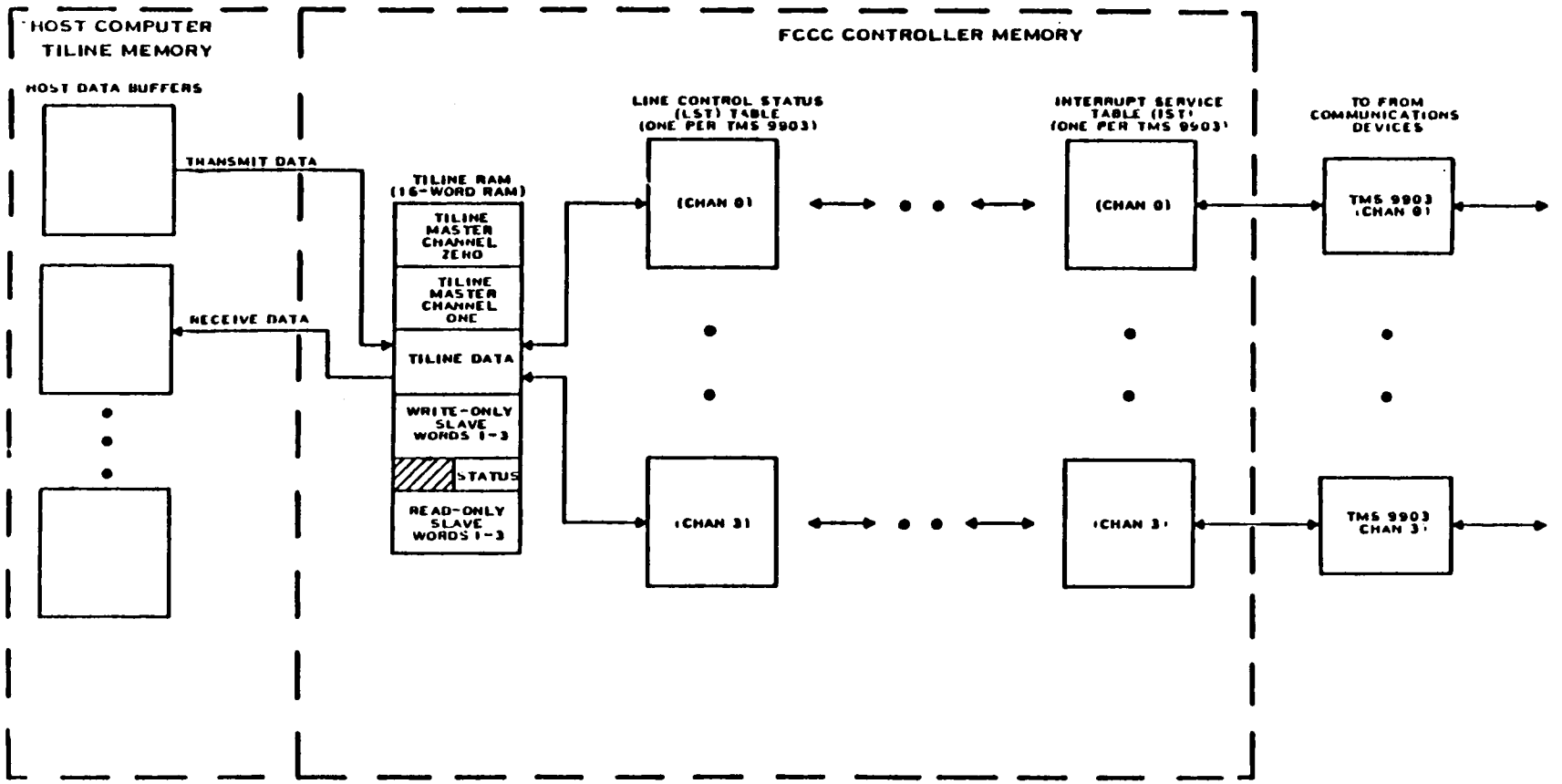


FIGURE III-5. Software Functional Block Diagram.

block diagram of the FCCC. The user may add other level interface tables and levels of processing if he wants. They are normally not present. These tables point to routines that process the data and contain the address of the TMS 9903 associated with the communications channel being used. After all data has been transmitted and a check for errors has been made, the completion status and other information relevant to the request are written by the microprocessor into the ROSWs (read-only slave words) in the TILINE RAM. Once the information is in the ROSWs, the host computer (TI-990/12 minicomputer) is interrupted, causing it to read the ROSWs in order to determine the status of the FCCC. This clears the interrupt so that another request can be initiated.

If the host computer had to read (receive) data from one of the communication devices instead of writing, then the data transfer would have been from FCCC memory to host computer TILINE memory and the address where the TILINE master controller stored the received data in host memory would appear in the ROSWs.

Host Data Structures

The host data buffer format is shown in Figure III-6. The buffer is basically divided into two parts -- the header and the data. The header consists of words 0 through 3, and the data consists of words n through m. Additional control or protocol dependent information can be passed to the slave processor by data

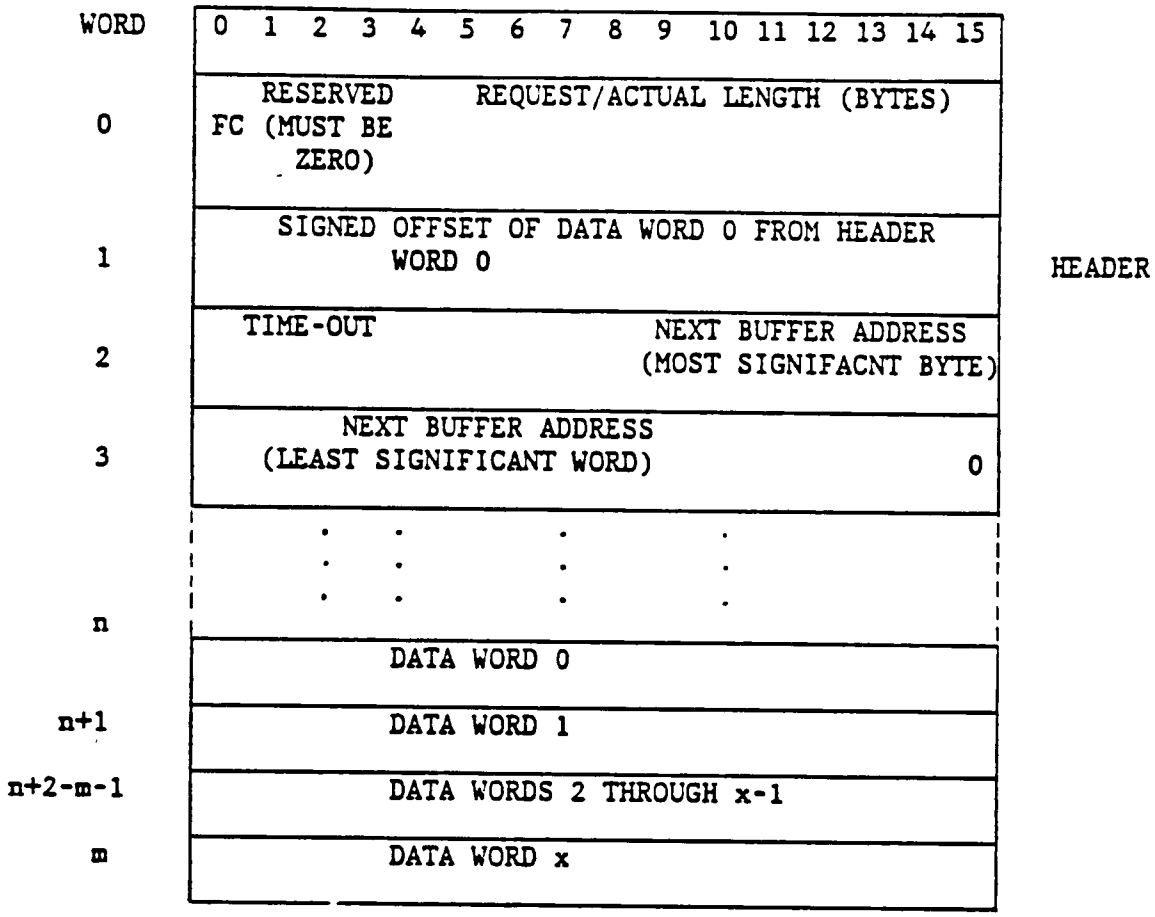


FIGURE III-6. Host Data Buffer Format.

words n through m. The data words n through m does not have to be located contiguously with the header words, but they must be located within + 16K bytes of the header. To provide enough data space for the transmission of an entire message in the selected protocol, buffers can be chained together. When a complete message or frame has been received and transferred to the host memory, the host is informed about this through the ROSWs.

As shown in Figure III-6, bit 0 of word 0 of the host data buffer contains the frame complete flag (FC). Bits 4 through 15 of word 0 indicates the request/actual length (byte count). Bits 1 to 3 of the same word are reserved for future use and must be set to zero. In host buffer chaining for write requests, bit 0 set to a one by the host indicates the end of the message and also the end of the buffer chain. When the request for reading is initiated, bit 0 is set to one by the host to indicate that the end of the host buffer chain has occured. When reading is completed, the FCCC interrupt service routine sets bit 0 to 1 in the buffer containing the last byte of the received data frame. As each buffer is transferred, bits 4 through 15 of word 0 is updated to indicate the actual number of data bytes transferred. If bits 4 through 15 contain all zeroes, then the request will be terminated with an error.

Word 1 of the host data buffer contains the signed offset of data word 0 from header word 0. In other words, this indicates the

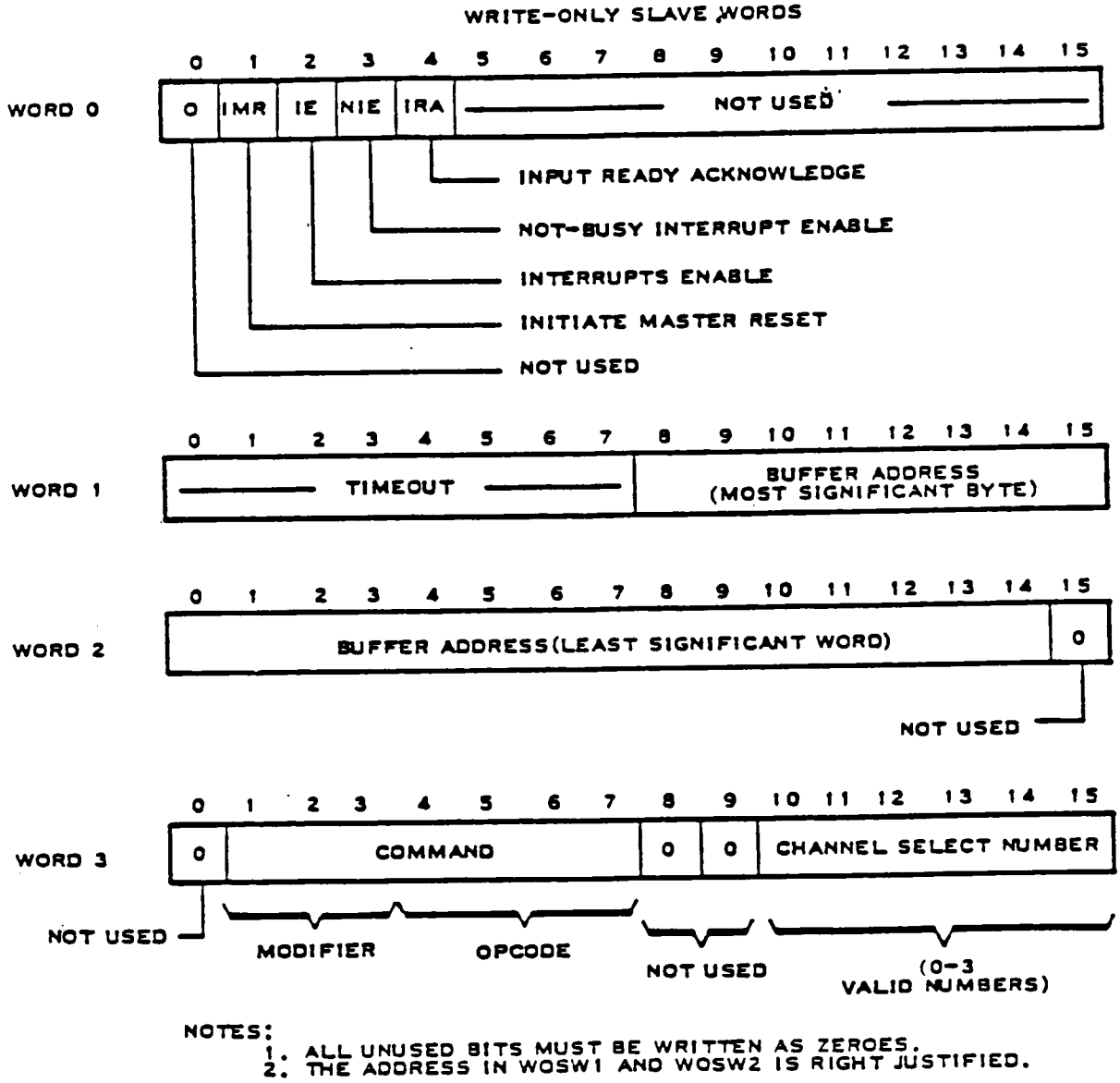
beginning of the data byte 0 in the host buffer. As the data bytes are not always located contiguously with the buffer header words, this information is necessary. Word 2 of the host data buffer contains the time-out information and the most significant byte of the 21-bit byte address of the next buffer in the chain. Time-out is specified by the word only for the first read buffer in a chained write (write and then read) command. In all other cases, the time-out is given by WOSW1. When the time-out information is valid, bits 0 through 8 indicates the number of time intervals that can be allowed for the completion of the command. The time intervals are selected by the protocol and are either 250 milliseconds or 1 minute. Bits 8 through 15 of word 2 contains the most sinificant byte of a 21- bit address of the next buffer header in the chain. If the next buffer header does not exist, then this information will be ignored.

Word 3 contains the least significant word of the 21-bit address of the next header word 0, which is concatenated with the LSB value of word 2 to form the next header address. Bit 15 of header word 3 is set to zero. Each host buffer header must start on a word boundary. The data words in the buffer are used to contain data characters unless they are needed to convey additional information about the command or protocol dependent information.

FCCC Commands

The host computer (TI-990/12 minicomputer) passes a command or request to the FCCC by writing into the WOSWs on the FCCC and then controlling the FCCC's response to these commands by reading the ROSWs written by the controller. Each command from the host to the FCCC define a specific task or tasks that the FCCC has to perform. As shown in the format of WOSW3 of Figure III-7, the commands are made up of a 3-bit modifier and a 4-bit command code, also called the opcode. These bits are written by the host into the WOSW3, bits 1 to 7. The opcode is written in bits 4-7, while the modifier is written in bits 1-3. The modifiers can be used to pass additional information about the command. They also allow one command, such as the write command with opcode 3, to be modified into several different sub-commands, such as a write parameters command (modifier 7, opcode 3). Certain commands issued by the host require data buffers to be generated in the host memory. These data buffers follow the same format as discussed in the previous section. For passing additional command parameters to the downloaded chracter detection routines on the FCCC, any number of data words in the host buffer can be used; but, in that case, these data words should also be includded in the byte count of the header word 0. Figure III-8 shows format of ROS Words.

Some of the commands may need the memory address to be



NOTES:
1. ALL UNUSED BITS MUST BE WRITTEN AS ZEROES.
2. THE ADDRESS IN WOSW1 AND WOSW2 IS RIGHT JUSTIFIED.

FIGURE III-7. Format of Write-only-slave Words.

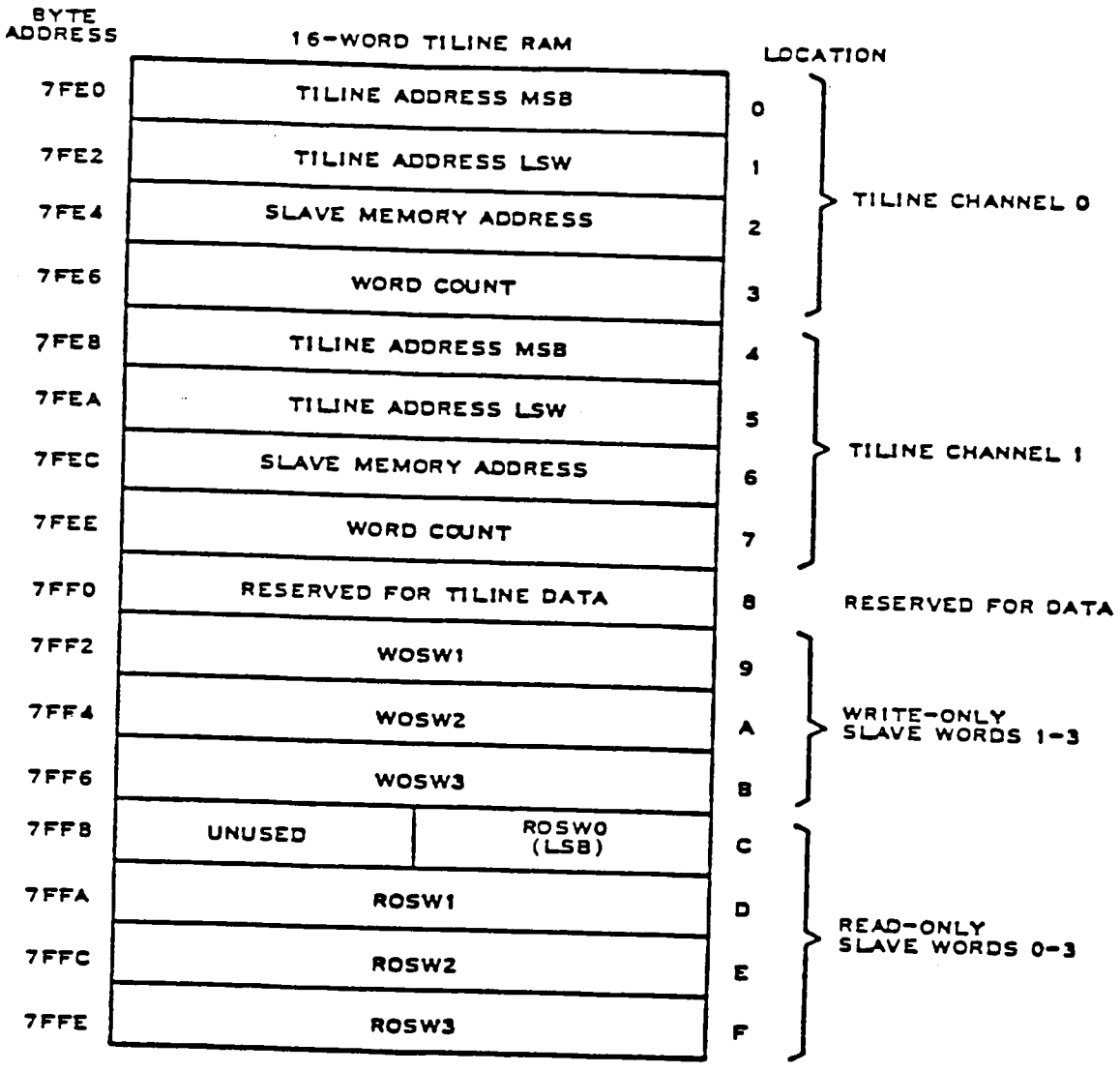


FIGURE III-8. Format of Read-Only-Slave Words.

specified on the FCCC. Figure III-9 is a layout of FCCC memory. 32K bytes of RAM, ranging from addresses >8000 through >FFFE, are available to the user for general READ/WRITE request. FCCC RAM may be reserved for downloaded code by issuing the reserve memory command (5D). For more information about the individual FCCC command, refer to Table 3.3 of the TI-990 manual on FCCC, Part No. 2263878-9701.

The "WRITE" Command for Data Transmission

The write command is used to transmit data over an FCCC communications channel; it may also be used to write parameters to the channel parameters table. When the write command is issued, bits 8-15 of WOSW1 and WOSW2 must contain the address of the data buufer containing the data to be transmitted or written. If a chain of host data buffers is used, then the frame complete flag (word 0, bit 0) should be set to one in the last buffer in the chain. The modifier code specifies the type of write command issued. Modifier 7 is used for write parameters command; modifier 0-3 are used to branch to different initial character detection transmit states.

WRITE PARAMETERS COMMAND: When the write parameters command (modifier 7, opcode 3) is issued, WOSW1 ans WOSW2 must contain the address of a host data buffer containing the protocol

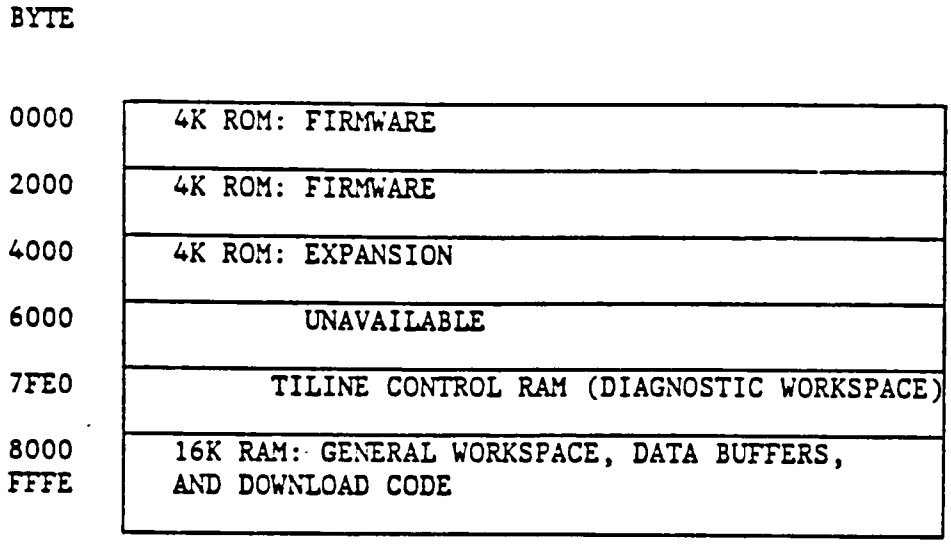


FIGURE III-9. FCCC Memory Layout.

information. The most significant byte of WOSW1 may be used to specify a protocol selection with the write parameters command. In order to select a given protocol supported by the FCCC, the hexadecimal values chosen are the same as in the open channel command. For a given protocol selection, the controller knows the location of its channel parameters table and also its contents.

Word 3 of the host data buffer header must contain an even number representing the offset in bytes from the beginning of the channel parameters table. This specifies the starting address of the parameters in the table to be updated. The data from the host buffer is copied into the channel parameters table starting at the specified offset. The number of bytes of information to be copied into the channel parameters table is given by the count in header word 0, bits 4-15. Chaining of data buffers is not allowed with the write parameters command. Multiple write parameters commands are necessary in order to update non-contiguous blocks of the channel parameters table. If bytes 0 through >31 (hexadecimal) of the channel parameters table are updated, the channel will be re-initialized upon comletion of the update. The format of the channel parameters table is shown in Figure III-10.

MNEMONIC		BYTE NUMBER (HEXADECIMAL)
CPPROS	PROTOCOL SELECTION WORD (BYTE 0)	0
	PROTOCOL SELECTION WORD (BYTE 1)	1
CPXTOC	TRANSMIT INTERCHARACTER TIME-OUT	2
CPRTOC	RECEIVE INTERCHARACTER TIME-OUT	3
CPCRUI	TRANSMIT CRU INSTRUCTION (BYTE 0)	4
	TRANSMIT CRU INSTRUCTION (BYTE 1)	5
CPCNTL	TMS 9903 CONTROL WORD (BYTE 0)	6
	TMS 9903 CONTROL WORD (BYTE 1)	7
CPSYN1	SYNC 1 CHARACTER	8
CPSYN2	SYNC 2 CHARACTER	9
CPSPCL	SPECIAL MODIFIERS	A
CPSCNT	INITIAL SYNC COUNT	B
CPRECL	PROTOCOL DEPENDENT USE (BYTE 0)	C
	PROTOCOL DEPENDENT USE (BYTE 1)	D
CPISRF	ISR FLAGWORD (BYTE 0)	E
	ISR FLAGWORD (BYTE 1)	F
CPIDLE	CONTINUOUS RTS IDLE CHARACTER	10
	RESERVED	11
	ISR RESERVED SPACE	BYTES 12-21
CPCDFL	CHRACTER DETECT FLAGWORD (BYTE 0)	22
	CHARACTER DETECT FLAGWORD (BYTE 1)	23
CPCDTB	CD ENTRY POINT TABLE ADDRESS (BYTE 0)	24
	CD ENTRY POINT TABLE ADDRESS (BYTE 1)	25
CPPOLA	POLL ADDRESS OR POINTERS	BYTES 26-2D
CPSYNI	SYNC INSERTION TIMER (BYTE 0)	2E
	SYNC INSERTION TIMER (BYTE 1)	2F
CPGLFL	GLOBAL FLAG BYTE	30
CPBFCT	BUFFER CHAIN THRESHOLD	31
CPSLCT	RESERVED >20 BYTES	BYTES 32-52

FIGURE III-10. Channel Parameters Table Format.

Device Service Routine (DSR) Issues

A device service routine (DSR) has to be written under the host operating system, which provides a software interface between the operating system of the host TI-990/12 minicomputer and the FCCC firmware. In addition to handling the general requests and commands directed to the FCCC, the DSR must perform the following specific tasks:

* A DSR power-up routine has to be provided to initialize the FCCC board.

*The TILINE Channel Data Transfer Test (Command 0F) must be successfully executed before the FCCC will accept any other commands.

* The DSR has to provide mapping to allow operating system commands to be utilized by the FCCC.

A group of CPU byte addresses in the host computer is reserved for addressing TILINE peripheral device controllers such as the FCCC. This 512 word group of addresses ranges from CPU byte addresses >F800 to >FBFE. Mapping logic in the host computer converts these 16-bit CPU byte addresses to 20-bit TILINE addresses to allow the host computer to directly address hardware registers on the peripheral controllers. The TILINE slave address switches has been set to >F900 (hexadecimal) for the FCCC board. If a task uses the TPCS (TILINE peripheral control spaces) addresses, then several considerations have to be taken through the software codes before

that task can be executed by the DX-10 operating system, of the TI-990/12 minicomputer. First, an LMF (load map file) instruction can be used within the software codes to load map file zero with six words of memory taken from the task status block (TSB) of the TI-990 minicomputer. The map file is a set of six registers that maps the 32K word addresses of the AU into the desired 20-bit addresses of TILINE memory. Second, the task has to be entered on the privileged mode for its execution. A priveleged task does not allow single-step execution of the program for debugging purposes.

A privileged SVC (opcode >2C) has been used to read the six memory words of the task status block, starting at offset >32 (hexadecimal). Now, in order to initiate reading of the TSB, it requires the run-time ID of the task to be passed as a byte information to the SVC. This calls for the need of the self-identification SVC (opcode >2E), from which the value of the run-time ID of the task can be obtained. With all these considerations, it has been found that the DX-10 operating system is not executing the instructions involving the TPCS addresses. It may be necessary to write into the six memory words of the TSB before those data are read. The data to be written, will depend on the range of addresses to be mapped to, for a given range of CPU memory addresses. However, EXTREME CAUTION HAS TO BE TAKEN WHILE WRITING INTO THE TSB (Task Status Block), because NO PROTECTION IS PROVIDED TO PREVENT THE USE OF THIS SVC FROM CAUSING A SYSTEM CRASH.

For more information about the LMF instruction, please refer to the LDD and LMF instructions in the 990 Assembly Language Manual; Part No. 2270509 -9701. Further information about the TSB and READ/WRITE TSB can be found in Systems Programming Guide, Vol-V, Part No. 946250-9705, of the TI-990 minicomputer manual.

The codes to initiate a power-up request for the FCCC and the TILINE channel data test has been shown in Appendix B. The TILINE channel data test must be the first command issued to the FCCC following power-up or master reset. FCCC master resets must be timed out by the DSR. If the reset is not complete within 10 seconds, the FCCC board is considered to be broken. The DSR must take care of the time-out internally, since no operating system support is provided for that. The FCCC firmware performs the TILINE channel data test by writing four data constants to a host memory address (provided as a part of the command when it is issued) over a TILINE data channel. The firmware then reads the four words reflected from the host and checks to see that the values read are the same as the values written. The TILINE test is successful if the values are the same.

Processing a Write Request for Transmission of Data

Before a write request can be processed, the FCCC must pass the power-up or the master test. Also, the TILINE channel data test should be performed successfully. Figure III-11 shows the entire sequence necessary for initiating a write request.

CODES FOR PROCESSING A WRITE REQUEST: The codes for processing a write request have been shown in Appendix A. The software routine uses some SVC to create a sequential file, which offers a way of debugging the program. If the codes function correctly, then it should transmit the same data repeatedly. The number of times the same data are transmitted, is determined by register R1. These codes have been written primarily to check if the transmission would work, using the FCCC via the asynchronous serial telecommunication line. An explanation similar to that of the master reset routine for accessing the TPCS addresses, can also be applied here. The asynchronous UPA protocol supported by the FCCC board has been selected for the communication. Write channel parameters command has been used to modify the channel parameters table for the UPA protocol. FCCC interrupts are disabled in portions of the codes to ensure that the processing of issuing the command (writing to WOSWs) is not interrupted. An unwanted FCCC interrupt can cause a command to be issued to the FCCC by the host. If care is not taken, a command could be issued with part of

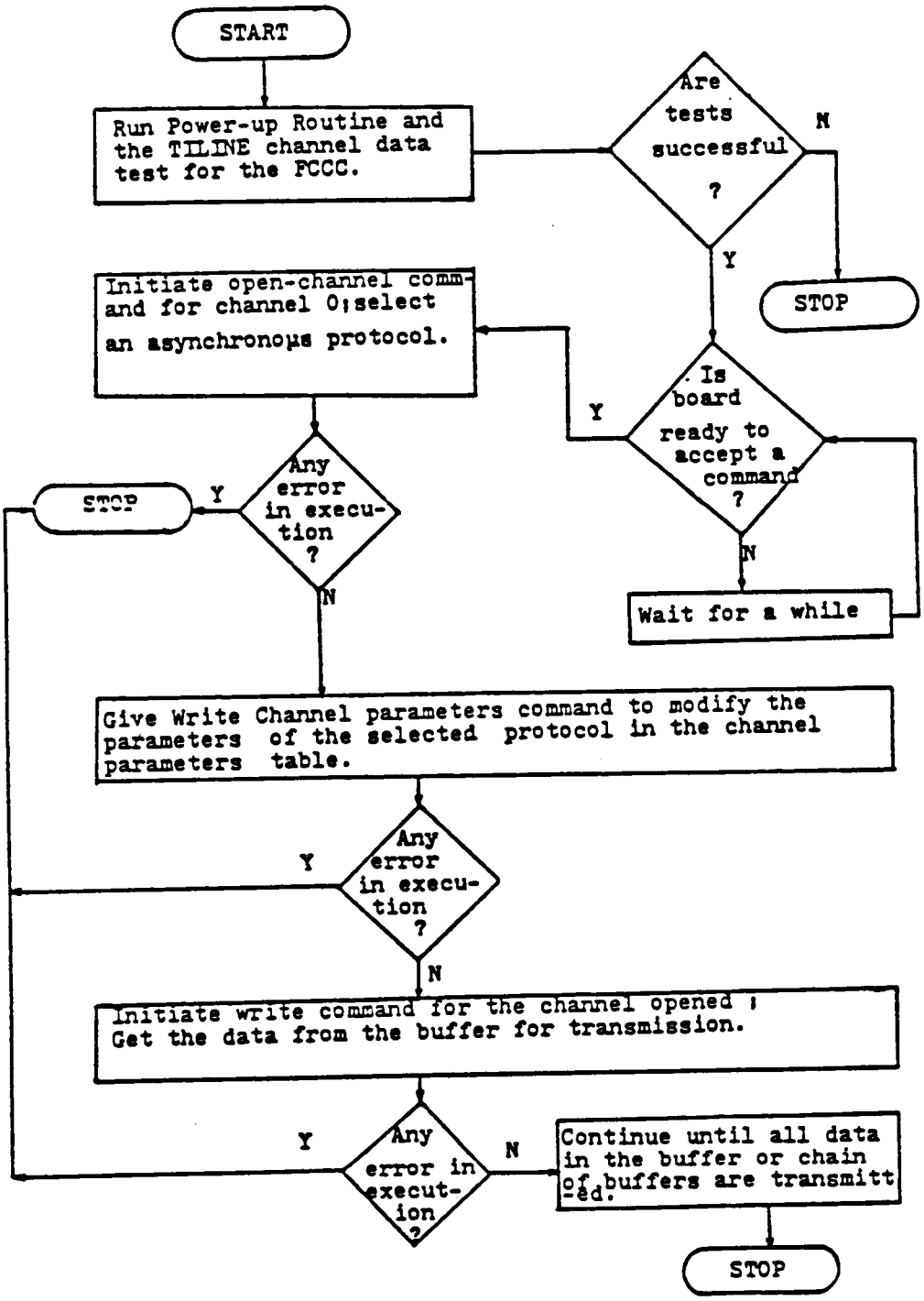


FIGURE III-11. Flowchart for Data Transmission from the Host.

the WOSWs containing parameters for one command and the rest of the WOSWs containing parameters for a different command.

PROCEDURE TO EXECUTE AN ASSEMBLY LANGUAGE PROGRAM ON THE TI-990 MINICOMPUTER: In order to execute a program on the DX-10 operating system, follow the sequences as mentioned below:

(1) Get the Object code of your source program by invoking the Macro-Assembler (using the XMA instruction). Make sure that the Macro Assembly completes without any error or warning.

(2) Create a Command file for the Link Editor by invoking the Text Editor (using the XE instruction). Assume the Object file pathname of the source code to be "AHS.OBJECT.FCCTST" and the name of the task to be executed be "FCCTST". Then the Control file should contain--

```
TASK FCCTST
INCLUDE AHS.OBJECT.FCCTST
END
```

Assume the Control file access name to be "AHS.SOURCE.DIAG".

(3) When the SCI prompt returns, invoke the Link Editor by entering XLE and pressing the return key. Supply the pathname of the control file you created and the pathname of a file where the results of the linking process can be placed by the Link Editor. Specify another pathname for the listing of the linking process. Accept the print width and page length defaults by pressing the return key.

The Link Editor executes in background mode, so you can enter "WAIT" and press the Return key to wait for the linking process to complete. When the Link EDitor terminates, the following message appears:

LINK EDITOR COMPLETED, 0 ERRORS, 0 WARNINGS

Press the Command key to return to SCI.

(4) Install the program-- Enter the Install Task (IT) to place the program on your program file. The name of the program file is chosen to be "AHS.PROG". Object Pathname or LUNO is the linked output access name created during the linking process. Take the default values of other parameters by pressing the Return key.

The system provides the installed ID, and displays it in the following form when the installation completes:

TASK NAME=FCCTST

TASK ID=nn

Press the Command key to return to SCI.

(5) Execute the program using the Execute Task and Suspend (XTS) SCI command. Select the following parameters:

PROGRAM FILE OR LUNO: AHS.PROG

TASK NAME OR ID: FCCTST

PARM1: 0

PARM2: 0

STATION ID: ME

This executes the program. After the execution is over, the control returns to the SCI. Delete the task entry by using the Delete Task (DT) Command as follows:

DELETE TASK

PROGRAM FILE OR LUNO : AHS.PROG

TASK NAME OR ID : FCCTST

If the Linkage Editor has already been invoked for your program, then you can skip steps (1) through (3) and perform steps (4) and (5).

STEPS NECESSARY TO EXECUTE THE POWER-UP AND TRANSMISSION ROUTINES FOR THE FCCC: For both the routines, the Linkage Editors have already been invoked. So, skip steps (1) through (3) and start from step (4).

(A) POWER-UP ROUTINE: For the Power-up and the TILINE channel data test routine, consider the following parameter values while performing steps (4) and (5).

PROGRAM FILE OR LUNO: AHS.PROG

TASK NAME OR ID: FCCTST

OBJECT PATHNAME OR LUNO: AHS.LINK.DIAG

After successful completion of the IT (Install Task) SCI command, the task ID will be displayed on the screen. Record the value of the task ID which will be useful for the next SCI command.

Modify the entry of the task by using Modify Task Entry (MTE) SCI command.

MODIFY TASK ENTRY

PROGRAM FILE PATHNAME: AHS.PROG

MODULE NAME OR ID:

For Module name or ID, use the value of the task ID as displayed on the screen after the execution of IT command. Press return key and take the default values of all the parameters by pressing the Return key repeatedly The following will be displayed on the screen:

SYSTEM: NO

PRIVILEGED: NO

MEMORY RESIDENT: NO

REPLICATABLE: YES

DELETE PROTECTED: NO

EXECUTE PROTECTED: NO

OVERFLOW: NO

WRITABLE CONTROL STORAGE: NO

Take the default values for all the flags with the exception of the priveleged flag. Change the priveleged flag to "YES". In this way, the task entered will be run on the priveleged mode.

After step (4) has been successfully completed, perform step (5). This completes execution of the task.

(B) TRANSMISSION ROUTINE: For the execution of the

transmission routine, the steps followed are the same as those of Section (A) above; only the file access names and the task name or ID are different. These are given as:

PROGRAM FILE OR LUNO: AHS.PROG

TASK NAME: TRANSMIT

OBJECT PATHNAME OR LUNO: AHS.LINK.CNTL1

The codes written for both master reset and transmission purposes, are not yet at its working stage. It was not possible to verify if the codes written for transmission would work or not. The reason for this is that the operating system is still unable to execute any of the instructions involving the TPCS addresses. It may be necessary to write into the Task Status Block (TSB) of TI-990/12 minicomputer to initiate this operation. A solution to this problem will help develop an alternative way of transmission for the TI-990/12 minicomputer.

Codes for Processing a Read Request

The TI-990/12 minicomputer is also capable of reading or receiving data from one of the communications channels, using the FCCC board. In case of Read command, the data transfer is from the FCCC memory to the TI-990/12 memory (host memory).

The codes for reception are almost the same as those of transmission except that a "Write" command opcode and modifier code

are substituted by the corresponding opcode and modifier code values for the "Read" command. A modifier code of value 0 is selected for the Read command to branch to the initial character detection receive state for channel 0. Using the Read command with modifier 0, WOSW1 (bits 8-15) and WOSW2 point to the address of the first buffer in a chain of host data buffers set up to receive the incoming data. When a host data buffer chain is set up, the frame complete bit (WOSW0, bit 0) in the last buffer in the chain must be set to 1 to terminate the chain. If the incoming data does not fill all of the data buffers in the chain, the firmware on the FCCC causes the Frame Complete bit to be set in the buffer containing the last byte of data received. The codes written for processing a Read request use only one buffer. Data is received into the TI-990/12 memory starting at location (RCVDAT +8). The software codes for reception have been shown in Appendix B.

CHAPTER IV

OPERATOR INSTRUCTIONS FOR FILE TRANSFER

The file transferring mechanism developed for the TI-990/12 with the DEC-10, uses the AMPL language of the TI-990 minicomputer. AMPL is a language close to the PASCAL. A sequence of operations on both the computers enables the TI-990/12 minicomputer to receive a file from the DEC-10; also, the TI-990 minicomputer can transmit a file to the DEC mainframe using another sequence of operations. The operator does not have to know much about either of the systems in order to do these file transferring operations. Many of the programs on the TI-990 minicomputer for the file transferring mechanism, have been written in the AMPL language of the TI. So, a brief description of the features of this language is given in this chapter. The chapter gives all the pieces of information necessary to transmit a file to, and receive a file from, the DEC-10.

AMPL Language Features

AMPL is a special PASCAL-like programming language to communicate with the emulator and trace modules of the TI-990/12 minicomputer. AMPL language features include:

* Interactive evaluation and display of expressions.

* Symbolic target system debug.

* User-definable and system-defined commands.

* Block-structured concepts and simple, concise syntax.

Support of user-defined commands by the AMPL language allows the user to develop standard support command for emulation, debug and trace data evaluation. Figure IV-1 shows the emulator hardware configuration, which includes a buffer module for the microprocessor to be used in the target system. For our case, the target system is the TM-990/U89 microcomputer kit with TMS 9980 as the microprocessor of the target system. The AMPL language can be used to initiate CRU Read/Write operations on TMS 9902 (Asynchronous Communications Controller) with a CRU base address of >800 (hexadecimal) for the TMS 9902.

Receiving a File from the DEC-10

The contents of a file, as sent by the DEC-10, are read as ASCII characters by the TI-990/12 minicomputer from the serial telecommunication line. Figure IV-2 helps to illustrate the various operations necessary for receiving a file from the DEC-10.

SEQUENCES OF EVENTS FOR RECEIVING A FILE FROM THE DEC-10: The following are the sequences which need to be followed exactly in order to receive a file from the DEC-10. Type in as indicated. VDT-911 is the terminal used by the TI-990/12 minicomputer. Any terminal compatible with RS-232C standard, is considered to be a DEC-10 terminal.

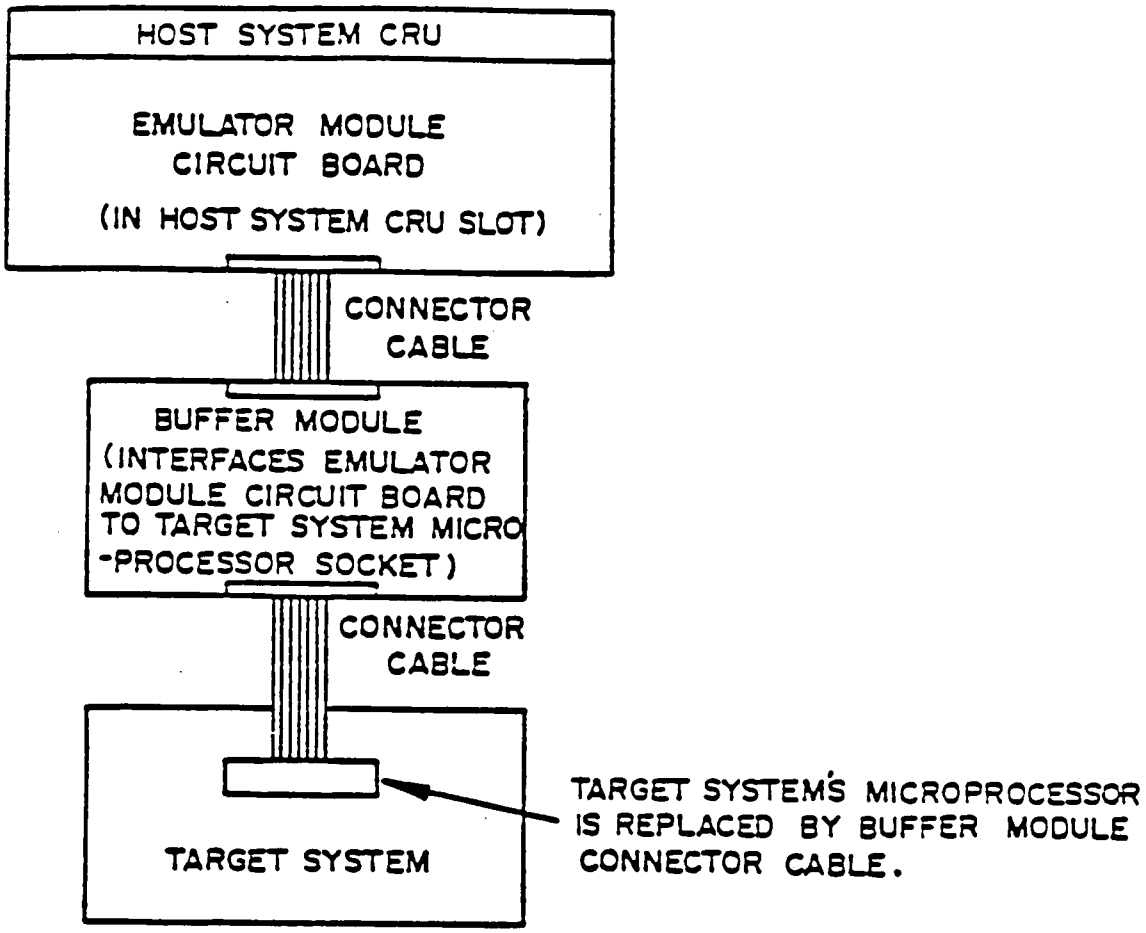


FIGURE IV-1. Emulator Hardware Configuration.

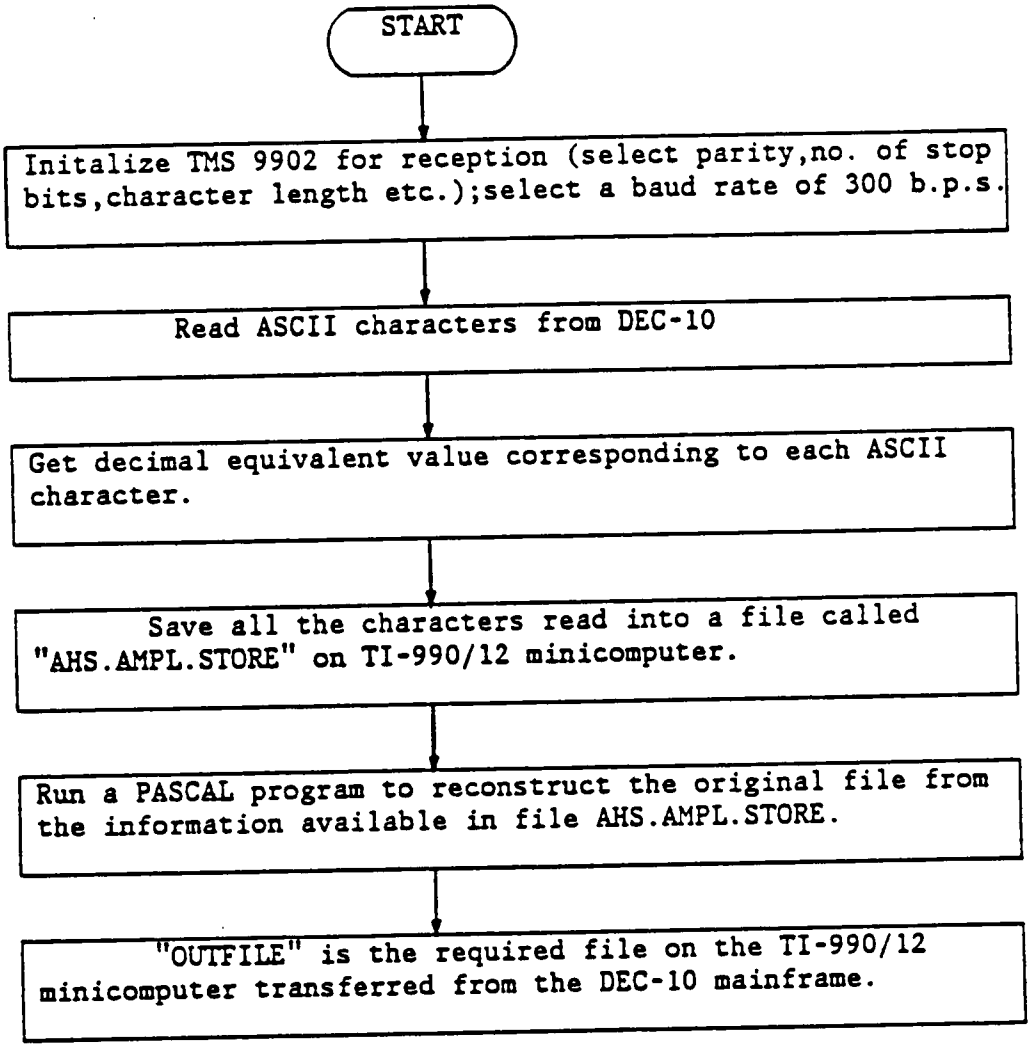


FIGURE IV-2. Flowchart for Receiving a File from DEC-10.

(1) After logging on the TI 990/12, access the AMPL utility of the TI-990/12 minicomputer by typing --

[]AMPLDX <Ret>

Hit <Ret> twice. You will get the prompt "?" on the TI screen.

(2) Now, you are within the AMPL utility of the TI-990 minicomputer. Type in the following on the TI terminal:

? RSTR('AHS.AMPL.RECEIVE') <Ret>

? RECEIVE <Ret>

The following message will appear on the TI screen.

INITIALIZE TMS 9902

NOTE:IT IS ASSUMED THAT THE EMULATOR

HAS BEEN INITIALIZED USING THE TARGET

SYSTEM CLOCK, e.g. EINT("EM01", 0)

BAUD RATE =300

(3) Before you hit <Ret> on the TI terminal, log on to the DEC-10 terminal (located in the same room), and type the following on the DEC-10.

. EX READ.PAS <Ret>

LINK : Loading

[LNKXCT READ execution]

INPUT : <Filename> <Ret>

OUTPUT :

<Filename> is the name of the file you want to send to the TI-990/12 minicomputer from the DEC-10.

(4) Now, hit <Ret> on the TI terminal.

Any character that appears on the screen of the DEC-10 terminal, is read by the TI-990 minicomputer. In addition to the characters appearing on the DEC screen, the TI-990 minicomputer also reads the unprintable control characters sent by the DEC-10.

(5) Look at the screen of the TI terminal. Sometimes, the terminal may show the message -- " STOP SENDING FROM DEC-10". This is an indication of the situation where the AMPL utility will fail to read any further characters from the DEC-10 because of its not being able to save them in the AMPL memory. After the message is displayed on the TI screen, the situation actually arises with the reading of about 300 more characters from the DEC-10. However, the total number of characters sent by the DEC at one time, has been carefully selected so that it does not create any memory overflow for the AMPL utility of the TI-990 minicomputer. Most of the time, this message will not be displayed on the TI screen.

When the DEC-10 has stopped sending the contents of the file you asked for, the DEC-10 terminal will show "EXIT" on its screen, which is not a part of the file you asked for. The

TI-990/12 minicomputer will also read these characters and save them.

(6) When the DEC terminal shows "EXIT" on its screen, hit <CMD> on the TI terminal. This terminates the TI-990 minicomputer from reading any further.

(7) Type on the TI terminal--

? CRFILE <Ret>

This opens a file named "AHS.AMPL.STORE" and saves all the characters (including the control characters) read into that file. The decimal equivalents of the ASCII characters just read are displayed on the TI screen. So, "AHS.AMPL.STORE" contains the decimal values of all the ASCII characters of the file sent from the DEC-10.

(8) Type on the TI--

? EXIT <Ret>

The following message is displayed on the TI--

NORMAL TERMINATION :

Hit <Ret> on the TI terminal. You are now out of the AMPL utility of the TI-990/12 minicomputer.

(9) On the TI terminal-- You can see the contents of the file just created with the help of the following command:

```
[] SF  <Ret>
   SHOW  FILE
         FILE PATHNAME : AHS.AMPL.STORE   <Ret>
```

Make certain that the file does not contain anything other than the decimal data. If it does, you need to delete those characters (using the editor), before that file can be used. Normally you may skip step-9 because the file "AHS.AMPL.STORE" will contain just decimal data except in the rare situation where the TI stops reading by itself. This can be very easily solved by reducing the total number of characters sent by the DEC-10 at one time (as described in step(3)) so as not to overflow the AMPL memory.

(10) Use the contents of the file "AHS.AMPL.STORE" to recover the original file sent by the DEC-10. To accomplish this, run a PASCAL program on the TI-990/12 minicomputer, as follows:

```
[] XPT <Ret>
```

The following message appears on the screen of the VDT-911.

```
EXECUTE  TI PASCAL TASK
     PROGRAM FILE : AHS.PASCAL.RECVLINK   <Ret>
  TASK NAME OR ID : RECEIVE       <Ret>
            INPUT : AHS.AMPL.STORE       <Ret>
           OUTPUT :  <Ret>
```

```
MESSAGES : ME       <Ret>
MODE(F,B,D) : FOREGROUND    <Ret>
MEMORY : 1,1      <Ret>
```

After the PASCAL program is run, it will give some text-file I/O error messages on the TI-990 minicomputer. But, do not worry about this. This occurs due to the fact that the PASCAL program running on the TI-990 minicomputer looks for a text-file input; but the input file "AHS.AMPL.STORE" is a sequential file (not a text file), and hence the error message.

(11) "OUTFILE" is the desired file obtained from the DEC-10. This last line of this file is "EXIT". If you delete this line, then whatever left is the actual file received from the DEC-10.

(12) Create a duplicate copy of the file on the TI-990 minicomputer by giving the following command.

[] CC <Ret>

The following message appears on the TI screen:

```
COPY/CONCATENATE
     INPUT ACCESS NAME  : OUTFILE      <Ret>
     OUTPUT ACCESS NAME : <Filename>    <Ret>
              REPLACE   : NO        <Ret>
   MAXIMUM RECORD LENGTH  : <Ret>
```

Output access name is the name of the duplicate file. If the output access name is "AHS.SOURCE.TEST", then the file "AHS.SOURCE.TEST" is a duplicate of the file "OUTFILE".

(13) If you want to send a large file from the DEC-10 to the TI-990/12 minicomputer, you will not be able to do so at one time. The large file on the DEC mainframe can be split into a number of smaller segments, which in turn, can be transferred from the DEC-10 to the TI-990 minicomputer, following the sequences as mentioned earlier. If you have to split a large program into a number of smaller segments by yourself, then there will be a synchronization problem. The synchronization problem is handled through software by a program called "READ.PAS" on the DEC-10. Proceed as follows if you need to transfer a large file:

(A) Use SOS editor on the DEC-terminal to change line 180 of the file "READ.PAS". This particular line contains the value of the variable called "COUNT". Initially, COUNT is zero. If the entire file is not transferred from the DEC-10 with this value of "COUNT", then increment "COUNT" by one, and substitute the old value by the new value(in this case, the new value of COUNT is 1; so, substitute "COUNT :=0; " by "COUNT :=1; ").

(B) With the new value of COUNT, continue steps (1) through (10) as mentioned earlier. You may skip step-(9). But, you do not need to do step (12).

(C) On the TI terminal-- If "AHS.SOURCE.TEST" is a duplicate

copy of the file created in step (12), then enter into the edit mode of the file by giving the following command.

```
[] XE  <Ret>
EXECUTE TEXT EDITOR
  FILE ACCESS NAME : AHS.SOURCE.TEST    <Ret>
   EXCLUSIVE EDIT  : YES       <Ret>
       LINE LENGTH : 80        <Ret>
```

(D) Hit <CMD> on the TI terminal. Type on the TI terminal--

```
[] IF    <Ret>
INSERT FILE
  INSERT AFTER LINE  : END    <Ret>
      FILE PATHNAME  : OUTFILE     <Ret>
```

You will see end-of-file marker (EOF) on the screen. Hit <CMD> on the TI terminal. Type on the TI terminal--

```
[] QE   <Ret>
QUIT EDIT
     ABORT ?:NO     <Ret>
QUIT EDIT
 OUTPUT FILE ACCESS NAME : AHS.SOURCE.TEST <Ret>
                 REPLACE : YES    <Ret>
    MOD LIST ACCESS NAME : <Ret>
```

With this operation, all the characters received lately with the present COUNT value will be saved consecutively with the characters received earlier, under the file name "AHS.SOURCE.TEST".

(E) Continue steps-(A) through (D) until the entire file has been transferred from the DEC-10 to the TI-990/12 minicomputer.

"AHS.SOURCE.TEST" is the desired file on the TI-990/12 minicomputer, which has been sent by the DEC-10. There will be some extra lines within the file just received. Each line will contain "EXIT". You need to delete these lines and edit the lines--one above and one below the line containing "EXIT" -- so that it matches the original file. Reception routines are shown in Appendix C.

Modified Technique for Reception

The technique for reception as discussed in the last section, may lose some characters because of the TI-990 model computer not being fast enough to read at the same speed as that sent by the DEC-10 mainframe. The reception technique described in this section works more reliably than that of the earlier section. Programs are run interactively between the DEC-10 mainframe and the TI-990 minicomputer. If a data packet is not received correctly by the TI-990, the DEC sends that packet again, and the process continues until the packet has been received

correctly by the TI-990 minicomputer. The flowcharts for the routines running on both the machines are shown in Figures II-8 and II-9.

SEQUENCES FOLLOWED IN RECEPTION:

(1) Get into the AMPL utility of TI-990 computer by following step-1 of the last section for reception.

(2) On the TI-990 computer, type the following:

? RSTR('R') <Ret>

? NEWRECV <Ret>

Do not hit the carriage return on the TI terminal now.

(3) Log on to DEC-10 following step-3 of the previous section. Type the following on the DEC-10 terminal.

. EX RECEIVE.PAS <Ret>

The following appears on the DEC screen.

READ : PASCAL

LINK : Loading

[LNKXCT READ execution]

INPUT : <Ret>

OUTPUT : <Ret>

INFILE : <Filename>

Supply a filename in response to INFILE and hit the carriage

return. This is the name of the file stored in DEC-10, which is to be sent to the TI-990 minicomputer. The following appears on the screen.

[INPUT, end with Z:]

Hit carriage return again. The following is displayed on the DEC.

VALUE OF COUNT =0

WRITE NEW COUNT VALUE

Start with a COUNT value of 0. If this value of count does not cover the entire file that needs to be transmitted to the TI, then increment the value of count by one every time, until the entire file has been transferred to the TI-990/12 minicomputer. So, in response to the above message, enter the correct COUNT value on the DEC-10 terminal. Each value of COUNT will enable the DEC-10 to transfer about 5000 chracters to the TI-990 computer. This is done because of the limitation of the TI-990/12 AMPL memory, which has a capacity of 9K bytes of RAM. Do not hit carriage return on the DEC-10 after entering the COUNT value.

(4) Now, hit carriage return on the TI terminal. The reception is initiated by this, and the ASCII values of the characters received are displayed on the TI screen. When the DEC-10 has stopped sending the characters, press the <CMD> key on the TI

terminal. this terminates the TI-990/12 minicomputer from receiving any further.

(5) Go out of the AMPL utility by following step-8 of the last section.

(6) AHS.AMPL.STORE is the filename, which contains all the ASCII values of the characters received from the DEC-10.

(7) Edit the file AHS.AMPL.STORE so that it contains only the decimal values.

(8) Follow step-10 of the last section. All names and parameters passed are the same with the exception of the name of the Program File. Enter the name of the program file as "AHS.PASCAL.RECVLNK" instead of "AHS.PASCAL.RECVLINK". Now, OUTFILE is the desired file obtained from the DEC-10 mainframe.

(9) For receiving a large file from the DEC-10, follow step-13 of the last section. The only difference in this case is that the file RECEIVE.PAS does not have to be edited for entering the COUNT value. Before going to step-13 of the last section, a duplicate file has to be created following step-12 of the same section. Step-12 will have to be performed only once with COUNT =0.

The modified reception routines are shown in Appendix C.

Transmission

In order to send a file to the DEC-10 from the TI-990/12 minicomputer, an SOS editor is used on the DEC terminal to create a new file. The file sent by the TI-990 minicomputer will be stored under this new filename on the DEC-10. A program has been written in the AMPL language of the TI- minicomputer, which can send a character to the serial telecommunication line if the ASCII value of the character is passed as an argument to the program. Although AMPL is a language similar to PASCAL, it does not have any way of reading a character from a file so that it can be passed as an argument to that program. This necessitates some pre-processing of the contents of the file to be transmitted. The various operations necessary to send a file from the TI-990/12 minicomputer to the DEC-10 mainframe can be understood with the help of Figure IV-3.

SEQUENCES FOR TRANSMISSION: The following are the sequences that need to be followed strictly in order to send a file from the TI-990/12 minicomputer to the DEC-10. Like reception, transmission also occurs through the asynchronous serial telecommunication line using a baud rate of 300 b.p.s. Transmission software routines also use the AMPL utility of the TI-990 minicomputer. Type in as indicated by the following sequences of transmission:

(1) Execute a PASCAL program on the TI-990/12 minicomputer with the help of the following instruction.

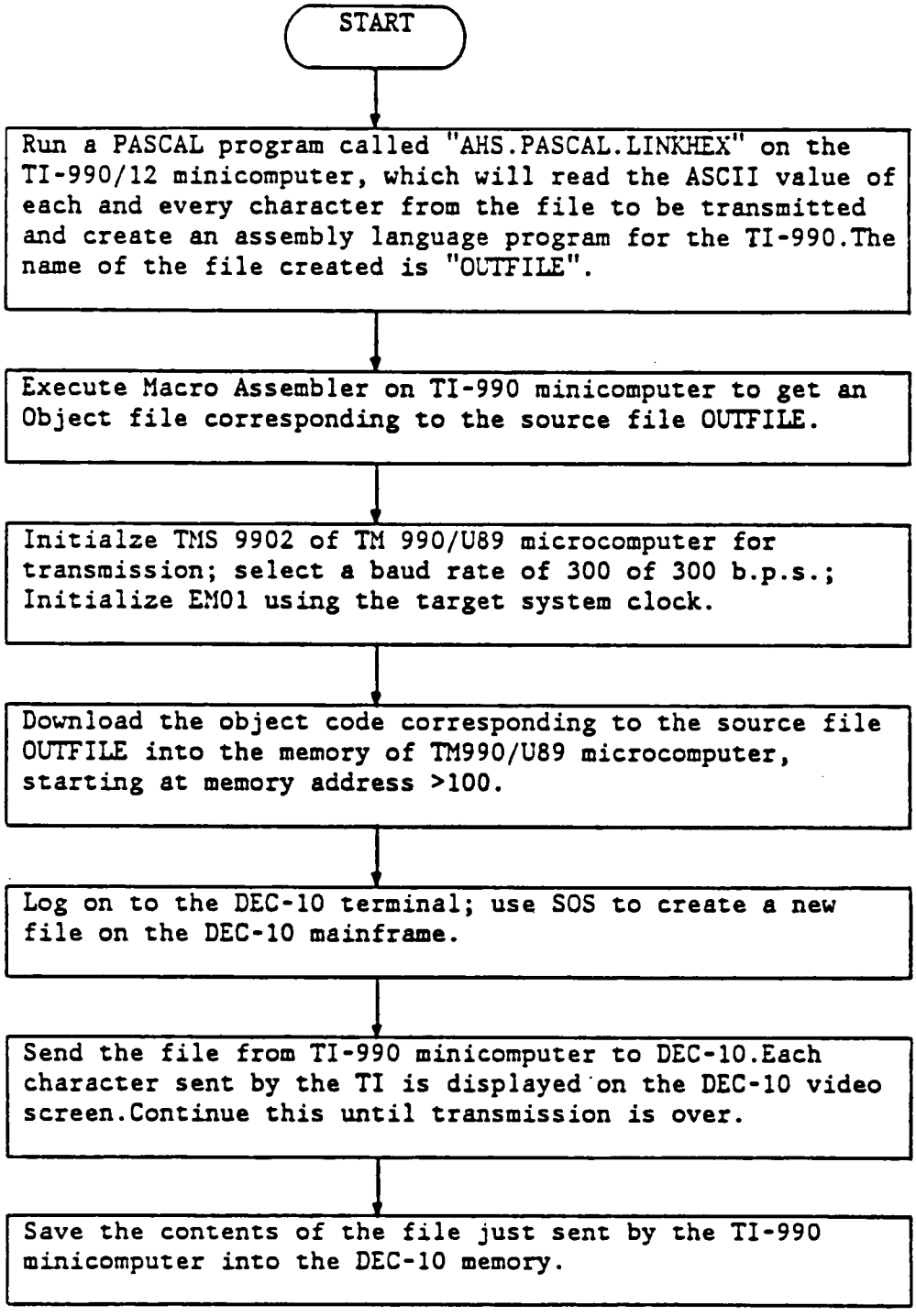


FIGURE IV-3. Flowchart for a File Transmission from TI-990 Minicomputer.

```
[] XPT    <Ret>
EXECUTE PASCAL TASK
   PROGRAM FILE : AHS.PASCAL.LINKHEX    <Ret>
TASK NAME OR ID : TOHEX   <Ret>
          INPUT : <Filename>   <Ret>
         OUTPUT : ME       <Ret>
       MESSAGES : ME       <Ret>
  MODE(F,B,D) : FOREGROUND   <Ret>
         MEMORY : 1,1          <Ret>
```

After the program has finished execution, record the value of a variable called "COUNT", which is displayed on the TI terminal. If COUNT is odd, then add one to its value to make it even. COUNT represents the total number of characters to be sent from the TI to the DEC-10 mainframe.

With the execution of the XPT instruction, a program named OUTFILE written in the assembly language of the TI-990/12 minicomputer, is created. The contents of this file are nothing but the decimal data written in assembly language format of the TI-990 minicomputer. Each data in OUTFILE represents a value corresponding to an ASCII character in the file to be sent. Input file pathname is the name of the file to be transferred to DEC-10 mainframe.

(2) On the TI terminal: Execute Macro Assembler to get the

object code of the source file named "OUTFILE", using the following command. Assume the object access name to be "AHS.OBJECT.OUTFILE".

```
[] XMA   <Ret>
EXECUTE MACRO ASSEMBLER
   SOURCE ACCESS NAME : OUTFILE      <Ret>
   OBJECT ACCESS NAME : AHS.OBJECT.OUTFILE  <Ret>
  LISTING ACCESS NAME : AHS.LIST.OUTFILE    <Ret>
    ERROR ACESS NAME :   <Ret>
              OPTIONS :   <Ret>
MACRO LIBRARY PATHNAME:   <Ret>
          PRINT WIDTH : 80     <Ret>
          PAGE LENGTH : 60     <Ret>
[] WAIT     <Ret>
```

The following message will appear on the top of the screen--"WAITING FOR BACKGROUND TASK TO COMPLETE--". At the bottom of the screen, the message displayed is--

```
== FOREGROUND COMMAND EXECUTING ==
```

Wait for a while until the execution is complete. After execution of the macro assembly is over, the following message is displayed on the screen.

```
MACRO ASSEMBLY COMPLETE, 0000 ERRORS, 0000 WARNINGS:
```

You can delete the OBJECT and LISTING files corresponding to the source file "OUTFILE", after the transmission is over.

(3) On the TI terminal-- To use the AMPL utility, type the following on the TI terminal:

[] AMPLDX <Ret>

Hit <Ret> twice. The prompt "?" will appear on the screen of the TI-terminal.

(4) On the TI terminal-- You are now within the AMPL utility of the TI-990/12 minicomputer. Proceed as follows:

? RSTR('T') <Ret>

? EINT('EM01', 0) <Ret>

? LOAD('AHS.OBJECT.OUTFILE', >100) <Ret>

"AHS.OBJECT.OUTFILE" is the object code corresponding to the source file "OUTFILE". You must download the object code as shown, starting at memory address >100 (hexadecimal value).

(5) On the DEC-10 terminal -- Use SOS editor to create a new file on the DEC-10. The file sent by the TI-990 minicomputer to the DEC-10 will be saved in the DEC mainframe under this filename.

(6) On the TI terminal-- To initiate transmission, proceed as follows:

? TRANSMIT <Ret>

The follwing message appears on the screen:

INITIALIZE TMS 9902

NOTE:IT IS ASSUMED THAT THE EMULATOR
HAS BEEN INITIALIZED USING THE TARGET
SYSTEM CLOCK, e.g. EINT('EM01', 0)

BAUD RATE =300

Hit <Ret> on the TI terminal. Then the following message will appear on the screen of the TI terminal:

WRITE THE VALUE OF COUNT TO BE PASSED AS AN
ARGUMENT. IF COUNT IS ODD THEN MAKE IT EVEN BY
ADDING 1 TO IT. VALUE OF COUNT CAN BE OBTAINED
WHILE EXECUTING THE PASCAL TASK [XPT]. COUNT
REPRESENTS TOTAL NUMBER OF CHARACTERS TO BE
TRANSMITTED. DO NOT FORGET TO LOAD THE OBJECT
CODE OF "OUTFILE" STARTING AT ADDRESS >100.

COUNT VALUE =

After you hit <Ret> on the TI terminal, the transmission is initiated. The characters sent to the DEC-10 wiil be displayed on its screen. Wait until the transmission is over. The end of transmission is indicated by the prompt "?" on the TI terminal. Also, when the DEC screen does not display any new charcters, then it is an indication of the end of transmission.

(7) The entire file has now been transferred from the TI-990/12 minicomputer to the DEC-10. Hit <ESC> on the DEC-10

terminal. The DEC terminal will show "*" on its screen. Type on the DEC terminal--

* ES <Ret>

So, the file just transferred from the TI-990 minicomputer is saved in the DEC-10. You can terminate your transmission any time from the TI- minicomputer by hitting <CMD> on the TI terminal. In that case, you will have to start the sequences all over again, starting from step-(1).

(8) If you have a large file to transfer from the TI-990/12 minicomputer to the DEC-10, you will not be able to do so in one session because of the limitation of the memory space of the target system (in this case, TM 990/U89 microcomputer). If the number of characters per line is large (say 50 characters or more), then a total of 150 lines (about 7800 characters) can be transmitted in one session to the DEC-10. Proceed as follows in order to transfer a long file. For instance, you want to transmit a file named "AHS.SOURCE.THESIS" from the TI-990/12 minicomputer to the DEC-10 mainframe.

(A) On the TI terminal-- Begin editing the file to be sent by typing --

[] XE <Ret>

EXECUTE TEXT EDITOR

FILE ACCESS NAME : AHS.SOURCE.THESIS <Ret>

```
EXCLUSIVE EDIT : YES      <Ret>
   LINE LENGTH : 80       <Ret>
```

File access name is the name of the file you want to send to the DEC-10 mainframe.

(B) Hit <CMD> on the TI terminal. Then type the following:

```
[] SVL    <Ret>
SAVE LINES
     START LINE :  1      <Ret>
       END LINE : 150     <Ret>
  SAVE FILE PATHNAME : <Filename>  <Ret>
 OPTION(ADD/REP/EXT) : ADD     <Ret>
```

The number shown here for the end line is arbitrary; it can be more than 150. Input file pathname is the name of the file which will have these 150 lines saved. Assume the input file pathname to be "AHS.TEST.TRY".

(C) Type on the TI terminal--

```
[] QE  <Ret>
QUIT EDIT
    ABORT : YES    <Ret>
```

(D) Continue steps (1)through (6). In step (1), choose the input file pathname as the one chosen in step 8(B), e.g.

"AHS.TEST.TRY". When step (6) is initiated, you can see the file being transferred to the DEC-10.

(E) Delete the file "AHS.TEST.TRY" by executing the following command:

```
[] DF  <Ret>
    DELETE FILE
      FILE PATHNAME : AHS.TEST.TRY    <Ret>
```

(F) Continue steps (A) through (E) with the line numbers changed in step 8(B) every time, until the entire file has been transferred to the DEC-10 mainframe. For the line numbers in step 8(B), it is better that you choose the next set of 150 lines (e.g. lines 151-300, lines 301-450 and so on) in order to keep some consistency with the transferring.

(G) When the entire file has been transferred, follow step (7) in order to store the file into the DEC-10.

The software routines needed to initiate various file-transferring operations like the transmission and the reception between the TI-990/12 minicomputer and the DEC-10 mainframe, are shown in Appendix D. The method that has been developed for file transferring, has been tested to to work fine between the minicomputer and the mainframe with no lost characters. As an illustration of this, this entire thesis written by the author was transferred from the TI-990 minicomputer to the DEC-10

mainframe. Then a program named "RUNOFF" was run on the DEC-10 to get a laser printing of the file transferred. Invertibility was tested for at least two different files on the DEC-10 -- "WRITE.PAS" and "SLOWWR.PAS". Also, the same was tested for a file named "AHS.SOURCE.WRITE" on the TI-990/12 minicomputer.

CHAPTER V

SUMMARY AND CONCLUSION

The file-transferring mechanism developed for the TI-990/12 minicomputer is simple and inexpensive. It is simple and convenient because it requires only that simple procedures be followed for its operation; and inexpensive because it just uses an asynchronous serial telecommunication line with a full duplex modem, following an RS-232C standard. The reception routines on the DEC-10 are written in standard PASCAL. These programs are system-independent. So, any computer systems accepting standard PASCAL can use these programs almost without any alterations.

The modified version of the reception technique works more reliably than that of the usual reception procedure, which had been developed earlier by the author. The modified technique uses the same checksum as that used by Kermit. The checksum is formed with a contribution of every bit from every data character in the packet except the checksum itself and the control characters (e.g. end-of-line, end-of-packet, end-of-file etc. characters). The probability that an error will remain undetected by a correctly transmitted arithmetic checksum is the ratio of the number of possible errors that cancel each other out to the total number of possible errors, which works out to $1/2^n$, where n is the number of

bits in the checksum, assuming all errors are equally likely [REFERENCE: Frank da Cruz and Bill Catchings; "Kermit -- A File Transfer Protocol for Universities";BYTE, July 1984, page 274). This is 1/64 for the single character checksum and (1/4096) for the two-character checksum. But the probability of errors remaining undetected cannot be easily determined because in reality all errors are not equally likely. However, the reliability of the reception technique can be increased significantly with the consideration of a two-character checksum.

Receiving a file from the DEC-10 mainframe to the TI-990/12 minicomputer requires some CPU time of the mainframe. Most of this CPU time is contributed by a software delay which is invoked between two successive writings of characters on the line by the DEC-10. However, this delay can also be created by some monitor calls of the DEC-10 mainframe, without using any of its CPU time. The transmission technique does not use any CPU time of the DEC-10 computer and works faster than that of reception without losing any character.

Because of the limitation of the AMPL memory (9 K bytes of user RAM), the TI- minicomputer can receive about 5000 characters at one time from the DEC-10. Although the DEC sends about 5000 characters, the TI reads more than 5000 characters from its line. This is because at the end of each line, the DEC-10 sends 5/6 more control charcters, which are not counted by the DEC, but are

read by the TI minicomputer. However, with the installation of the AMPL expansion memory board, the memory size will increase to 64K bytes. This will enable the TI minicomputer to receive much bigger files without causing any overflow of the AMPL memory.

The techniques which have been developed for transmission as well as reception are capable of transferring only sequential and textual files. The method will not be applicable to a binary file, control file or any non-sequential file.

As discussed in Chapter II, the receiving Kermit can be implemented on the TI- minicomputer using the AMPL utility, if the procedure named "SPACK" can be made to work within the Kermit packet. The possible limitations of the receiving Kermit and its solutions have also been discussed. Since, the AMPL has a limitation of its memory size of about 9K bytes, it is better that the transmission and reception are treated separately for the Kermit. In that case each operation, whether it is transmission or reception, will have more memory space available for its use. However, there is one difficulty with the transmission routine written in the AMPL language. The AMPL does not have any way of reading a character from a file and then passing it as an argument. This may put some serious restrictions on writing a Kermit program in the AMPL language for transmission.

The file transferring mechanism developed has been tested to send a file reliably from the TI-990/12 minicomputer to the

VAX-11/780 computer within the department. Receiving files from the VAX-11 computer to the TI-990/12 minicomputer can be initiated by following almost similar procedures as has been explained for the DEC-10 with the TI-minicomputer.

The need for a cheap, convenient file transfer capability among diverse systems is pressing. The kind of work that has been done with the TI-990 minicomputer, is the first of its kind in the department towards attaining that goal. Although much has been done for a convenient file transferring capability between two diverse systems -- the TI-990/12 minicomputer and the DEC-10 mainframe-- much more remains unexplored. It is expected that future endeavor and research will continue to grow in this field to enable the TI-990/12 minicomputer to communicate with more diverse systems and increase its capability within the department.

LIST OF REFERENCES

LIST OF REFERENCES

1. Tanenbaum, Andrew S. Computer Networks. New Jersey: Prentice-Hall, 1981.

2. Mcnamara, John E. Technical Aspects of Data Communication. Digital Equipment Corporation, 1982.

3. Dell, Nell, and David Orshalic. Introduction to PASCAL and Structured Design. Lexington,Massachusetts: DC Heath and Company, 1983.

4. Freeman, Harvey A. and Kenneth J. Thurber. Microcomputer Networks. New York: I.E.E.E. Computer Society.

5. Texas Instruments. TM 990/U89 Microcomputer User's Guide Houston, Texas, 1981.

6. Texas Instruments. 9900 Family Systems Design. First Edition. Dallas, texas.

7. Da Cruz, Frank and Bill Catchings. Kermit: A File Transfer Protocol for Universities; Part-I. BYTE, June 1984, page 255.

8. Da Cruz, Frank and Bill Catchings. Kermit: A File Transfer Protocol for Universities; Part-II. BYTE, July 1984, Page 143.

9. Texas Instruments. Model 990 Computer DX10 Pascal Programmer's Guide. Part Number: 2270528-9701, January 1984.

10. Texas Instruments. Model 990 Computer TI Pascal Reference Manual. Part Number: 2270519-9701, January 1984.

11. Texas Instruments. Model 990 Computer Assembly Language Reference Manual. Part Number: 2270509-9701, January 1981.

12. Texas Instruments. Model 990 Computer DX10 Operating System Concepts and Facilities. Volume I. Part Number: 946250-9701, September 1983.

13. Texas Instruments. Model 990 Computer DX10 Operating System Operations Guide. Volume II. Part number: 946250-9702, September 1983.

14. Texas Instruments. Model 990 Computer DX10 Operating System Application Programming Guide. Volume III. Part number: 946250-9703, September 1983.

15. Texas Instruments. Model 990 Computer DX10 Operating System Systems Programming Guide. Volume V. Part number: 946250-9705, September 1983.

16. Texas Instruments. Model 990 Computer AMPL Microprocessor Prototyping Laboratory Operation Guide. Part Number: 946275-9701, November 1980.

17. Texas Instruments. Model 990 Computer AMPL Microprocessor Prototyping Laboratory Tutorial. Part Number: 946276-9701, November 1980.

18. Texas Instruments. Model 990 Computer AMPL Microprocessor Prototyping Laboratory User's Guide. Part Number: 946277-9701, November 1980.

APPENDICES

APPENDIX A

```
.....................................................................
.. RECEIVING KERMIT PROGRAM  FOR THE TI-990/12 MINICOMPUTER
.. THE PROGRAM CAN BE FOUND UNDER THE FILENAME "AHS.TEST.RECEIVE"
.. FOR THE ORIGINAL VERSION OF KERMIT SEE THE PROGRAM UNDER THE
.. FILENAME "MTSKERMIT.PAS" WRITTEN FOR THE MICHIGAN TERMINAL
.. SYSTEM (MTS) BY WILLIAM S. HALL, MATHEMATICAL REVIEWS,ANN ARBOR,
.. MI, IN PASCAL/VS.
.....................................................................


PROC INIT(1,1)
     EINT('EM01',0)     ..INITIALIZE EMULATOR
     ..ASSUME EITHER USING TM990/101 OR TM990/U89
     ..MUST BE USING TARGET CLOCK (ECLK MUST BE ZERO)
     ..ASSUME 3 MHZ CLOCK IF TMS9900 (ETYP =1)
     ..ASSUME 2 MHZ CLOCK IF TMS9980 (ETYP=2)
     ..ASSUME 7 BITS/CHARACTER,2 STOP BITS,EVEN PARITY
     ..CLOCK 4M=0; INTERNAL FREQ IS PHASE 3 DIVIDED BY 3
     IF ECLK NE 0 THEN 'WARNING:TARGET CLOCK MUST BE USED';NL
     IF ETYP EQ 1 THEN BEGIN CLOCK =30000     ..3 MHZ FREQ/100
          CRUB=080
          END
     ELSE
          IF ETYP EQ 2 THEN BEGIN CLOCK =20000     ..2 MHZ FREQ/100
          CRUB=0800
          END
          ELSE
               'WARNING:MUST USE EITHER 9900 OR 9980 BUFFER';NL
     CRUW(31,1,1)          .."SBO 31"      9902 RESET
     CRUW(0,8,>A2)         .."LDCR @CNTRL,8"   LOAD CONTROL REGISTER
     CRUW(13,1,0)          .."SBZ 13"      SKIP INTERVAL REGISTER
     BAUD=ARG 1/100
     IF CLOCK /(3*2*BAUD) GE >0400 THEN PRE=8
     ELSE PRE=1
     DRVAL =CLOCK /(3*2*PRE*BAUD)
     IF PRE EQ 8 THEN DRVAL= >0400 OR DRVAL
     CRUW(0,12,DRVAL)           .."LDCR @BAUD"       LOAD XMIT/RECEIVE RATE
     END
FORM INIT
          ':INITIALIZE TMS9902';
          ':NOTE: IT IS ASSUMED THAT THE EMULATOR';
          ':      HAS BEEN INITIALIZED USING THE TARGET';
          ':      SYSTEM CLOCK;E.G. EINT("EM01",0)';
          'BAUD RATE' ='300';
     END
```

```
.....................................................................
.. PROCEDURE DISPLAY MAY BE  USED FOR DEBUGGING PURPOSES.IT CAN BE USED
.. ANYWHERE INSIDE THE KERMIT PROGRAM  TO DISPLAY SOME CHARACTERS
.. (MESSAGES) WHICH INDICATES WHETHER A PARTICULAR SEGMENT OF THE
.. PROGRAM IS CORRECTLY WORKING OR NOT.
.....................................................................


PROC DISPLAY(1,1)
LOC 1 = ARG 1
LOC 1 = LOC 1 + >2000       ..TO SUPPRESS DISPLAYING OF CONTROL CHARACTER
FOR I = 1 TO 2 DO LOC 1:A   ..DISPLAY THE SAME CHARACTER TWICE
END

.....................................................................
.. ALL GLOBAL VARIABLES HAVE BEEN DEFINED IN THE PROCEDURE GLOBAL.
.....................................................................


PROC GLOBAL
     ARRAY DATA(100)
     ARRAY TEMP(100)
     ARRAY BUFFER(100)
     MAXPACK =94     ..MAXIMUM PACKET SIZE
     MYQUOTE =" "    ..QUOTE CHARACTER I WILL USE
     FLAG =0
     LEN=0
     MYPAD =0; MYPCHAR =0
     MYEOL =13; MYTIME =5 ; SOH=1
     NUMTRY = 0; OLDTRY =0;
     STATE =1; EOL = 13
     SPSIZ =0; TIMINT =0; PAD = 0; PADCHAR = 0
     QUOTE = 0; L =0;CHKSUM =0
END
.............................................................
.. PROCEDURE ONERD READS JUST ONE CHARACTER FROM THE SERIAL
.. COMMUNICATION LINE.
.............................................................

PROC ONERD(0,1)
     IF (ETYP EQ 1) THEN CRUB =>080
     ELSE CRUB =>0800     ..CRU BASE ADDRESS FOR TMS9902
BEGIN
     WHILE ((CRUR(21,1) AND 1) EQ 0) DO NULL    ..WAIT TIL RBRL=1
     LOC 1 =CRUR(0,8)     ..READ 8 BITS CORRESPONDING TO THE ASCII
                          ..VALUE OF THE CHARACTER AND SAVE IT
                          ..TEMPORARILY IN A LOCATION.
     T =LOC 1     ..T CONTAINS ASCII VALUE OF CHARACTER JUST READ
     CRUW(18,1,0)         ..ENABLE NEXT CHARACTER
```

```
     END
END

.....................................................................
.. FUNCTION UNCHAR CONVERTS A PRINTING CHARACTER TO AN INTEGER IN
.. THE RANGE 0-94. THIS FUNCTION UNDOES THE ACTION OF "TOCHAR".
.....................................................................

FUNC UNCHAR(1,2)
     BEGIN
          LOC 1 = ARG 1
          LOC 2 = LOC 1 -32
     END
     RETURN LOC 2
END

.....................................................................
.. FUNC "READCH" READS MULTIPLE CHARACTERS AND RETURNS THE VALUE 1
.. IF SOH(START-OF-HEADER) ISN'T READ;OTHERWISE RETURNS THE VALUE 0
.. ALL CHARACTERS READ ARE STORED IN THE ARRAY NAMED "TEMP"
.....................................................................
FUNC READCH(0,1)
     ONERD                        ..READS ONE CHARACTER
     LEN= UNCHAR(T)-3             ..GET LENGTH OF PACKET
     IF (LEN HI 100) THEN RETURN(0)
     TEMP(1) =T
     FOR J = 2 TO (LEN+3+1) DO BEGIN
          ONERD                   ..READ A CHARACTER
          TEMP(J) =T              ..STORE THE CHAR. VALUE IN ARRAY TEMP
          IF (T EQ SOH) THEN RETURN(0)  ..IF START OF HEADER CHAR. THEN
                                        ..  RETURN VALUE 0
          END
       RETURN(1)
END
.....................................................................
.. PROC "ONEWR" WRITES (SENDS TO THE DEC) ONE CHARACTER ON TO THE
.. ASYNCHRONOUS SERIAL TELECOMMUNICATION LINE.
.....................................................................

PROC ONEWR(1)
     IF (ETYP EQ 1) THEN CRUB =080
     ELSE CRUB =>0800         ..CRU BASE ADDRESS OF TMS-9902
     CRUW(16,1,1)             ..SET RTSON IN TMS-9902
     WHILE ((CRUR(22,1) AND 1) EQ 0) DO NULL    ..WAIT TIL XBRE=1
     CRUW(0,8,ARG 1)          ..WRITE THE CHARACTER ON THE LINE,THE ASCII
                              ..VALUE OF WHICH HAS BEEN PASSED AS AN
                             ..ARGUMENT (ARG 1) WHILE INVOKING THE PROCEDURE
END
```

```
.....................................................................
.. PROC "WRITCH" WRITES(SENDS) MULTIPLE CHARACTERS ON TO THE SERIAL
.. TELECOMMUNICATION LINE; THE ARGUMENT PASSED WHILE INVOKING THE PROC,
.. DETERMINES THE NUMBER OF CHARACTERS TO BE WRITTEN.
.....................................................................

PROC WRITCH(1)
     IF (ETYP EQ 1) THEN CRUB =>080
     ELSE CRUB =>0800
     FOR I=1 TO ARG 1 DO BEGIN
       ONEWR(BUFFER(I))  ..WRITES ONE CHARACTER TO THE LINE.BUFFER(I)
                         ..CONTAINS ASCII VALUE OF CHARACTER TO BE SENT
       BUFFER(I):A       ..TO THE DEC
       END
END

.....................................................
.. FUNC "TOUPPER" CONVERTS LOWER CASE TO UPPER CASE
.....................................................

FUNC TOUPPER(1,2)
     BEGIN
          LOC 1 =ARG 1    ..C STORED
          IF (LOC 1 GE "a") AND (LOC 1 LE "z") THEN LOC 2= LOC 1-32
          ELSE LOC 2 =LOC 1    ..ARG 1
     END
     RETURN LOC 2
END

.....................................................................
.. FUNC "CHECKSUM" RETURNS ASCII CHECKSUM OF THE CHARACTERS READ;
.. COMPUTES A CHECKSUM IN THE RANGE 0- 63 AND RETURNS THE VALUE.
.....................................................................

FUNC CHECKSUM(1,2)
     BEGIN
          X = (ARG 1 MOD 256)
          LOC 1=X
          X=DIV(64,LOC 1)
          X = X+ ARG 1
          LOC 2 = (X MOD 64)
     END
     RETURN LOC 2
     END
```

```
.....................................................................
.. CONVERTS AN INTEGER IN THE RANGE 0-94  TO A PRINTING CHARACTER
.....................................................................

FUNC TOCHAR(1,2)
     BEGIN
          LOC 1 =ARG 1
          LOC 2 = LOC 1 +32
     END
     RETURN LOC 2
END

.....................................................................
.. FUNC "CTL" CHANGES THE PRINTING CHARACTERS TO CONTROL CHARACTERS;
.. USED TO UNQUOTE A QUOTED CONTROL CHARACTER IN A PACKET.
.....................................................................

FUNC CTL(1,2)
     BEGIN
          LOC 1= ARG 1
          LOC 2 =LOC 1-64
     END
     RETURN LOC 2
END

.....................................................................
.. FUNC "UNCTL" CHANGES A CONTROL CHARACTER TO ITS CORRESPONDING
.. PRINTING FORM
.....................................................................

FUNC UNCTL(1,2)
     BEGIN
          I =ARG 1
          LOC 1 = I+64
     END
     RETURN LOC 1
END

.....................................................................
.. FUNC "WRITEOPN" OPENS A FILE FOR WRITING DURING RECEIVE MODE.
.. IF THE DEFAULT VALUE IS TAKEN, THEN "AHS.AMPL.STORE" IS THE NAME
.. OF THE FILE BEING OPENED.
.....................................................................

FUNC WRITEOPN(1)
     LIST('AHS.AMPL.STORE')
END
..FORM WRITEOPN
..    ':ADD A SINGLE QUOTATION MARK ';
```

```
..    ':IN THE FRONT AND AT THE END OF ';
..    ':THE FILE NAME WHEN YOU GET THE PROMPT';
..    'RECEIVE FILE NAME'='AHS.AMPL.STORE'
..    END

.....................................................................
.. PROC "RPAR" GETS THE OTHER SIDE'S SENT-INIT PACKET.THE TIME-OUT
..  IS N/A.
.....................................................................

PROC RPAR
     BEGIN
          SPSIZ = UNCHAR(DATA(1))   ..MAXIMUM SEND PACKET SIZE
          TIMINT= UNCHAR(DATA(2))   ..WHEN I SHOULD TIME OUT
          PAD =UNCHAR(DATA(3))      ..NUMBER OF PADS TO SEND.
          PADCHAR = CTL(DATA(4))    ..PADDING CHAR TO SEND
          EOL =UNCHAR(DATA(5))      ..END-OF-LINE CHAR TO SEND
          QUOTE = DATA(6)           ..INCOMING DATA QUOTE CHAR
     END
END

.............................................
..FILL DATA-ARRAY WITH MY SENT-INIT PACKET.
.............................................

PROC SPAR
     BEGIN
          DATA(1) = TOCHAR(MAXPACK)      ..MY MAX PACKET SIZE
          DATA(2) = TOCHAR(MYTIME)       ..WHEN I SHOULD BE TIMED OUT
          DATA(3) = TOCHAR(MYPAD)        ..HOW MUCH PADDING I NEED
          DATA(4) = TOCHAR(MYPCHAR)      ..MY PAD CHAR
          DATA(5) = TOCHAR(MYEOL)        ..MY END-OF-LINE CHAR
          DATA(6) = MYQUOTE              ..QUOTE CHAR I SEND
     END
END

.....................................................................
.. READ A PACKET BEING SENT; COMPUTE CHECKSUM AND RETURN PACKET-TYPE
.....................................................................

FUNC RPACK(0,4)
BEGIN
LOC 1 = 1
  WHILE (LOC 1 EQ 1) DO BEGIN
  WHILE (T NE SOH) DO ONERD         ..LOOK FOR SYNCH. CHAR SOH
  'ZERO '          ..DISPLAY MESSAGE ON THE SCREEN TO INDICATE "RPACK"
   LOC 2 = 0                        ..IS RUNNING
  WHILE (LOC 2 EQ 0) DO BEGIN
       IF (READCH EQ 0) THEN ESCAPE   .. IF SYNCH CHAR SOH,THEN START
```

```
        'ONE'                                .. AGAIN
        NUM = UNCHAR(TEMP(2))               .. GET PACKET NUMBER
        'THREE'                             ..WHEN DISPLAYED, INDICATES "RPACK"
                                            .. IS RUNNING
        CLASS = TEMP(3)                     .. GET PACKET TYPE
        FOR I= 1 TO LEN DO BEGIN
             DATA(I) = TEMP(I+3) ..STORE DATA IN THE ARRAY NAMED "DATA"
        END
        T =TEMP(LEN +3+1)               .. GET SENDER'S CHECKSUM
        'FOUR'
        LOC 2= 1
        LOC 1 = 0
  END
     CHKSUM=0
     FOR I=1 TO (LEN +3) DO CHKSUM=CHKSUM +TEMP(I)    ..CALCULATE
                                       ..CHECKSUM OF CHARS JUST READ
     IF (TEMP(LEN +4) EQ TOCHAR(CHECKSUM(CHKSUM))) THEN LOC 4 = CLASS
       ELSE LOC 4 = "E"          ..COMPARE CHECKSUM; RETURN "E" IF BAD
END
FOR I =1 TO LEN  DO BEGIN
     LOC 3 = DATA(I)
     LOC 3 = >2000 + LOC 3     ..REPLACE NON-PRINTABLE ASCII CHAR ON
                               ..THE MSB BY SPACE
     LOC 3:A
     END
RETURN (LOC 4)
END
END

.....................................................................
.. PROC "SPACK" SENDS A PACKET TO THE OTHER SIDE; THE PROCEDURE
.. USES TWO ARRAYS -- "BUFFER" AND "DATA"
.....................................................................

PROC SPACK(3,3)
     BEGIN
          'SENDING'
          IF (PAD GT 0) THEN BEGIN       ..SEND PADDING IF NEEDED
             FOR I =1 TO PAD DO BEGIN
             ONEWR(PADCHAR)          ..WRITCH(PADCHAR)
             'PADCHAR=';PADCHAR:D    ..MESSAGES DISPLAYED ON SCREEN
             'I=';I:D
             END
          END
          BUFFER(1) =SOH                  ..SYNCH CHARACTER
          LOC 1 = TOCHAR(ARG 3 +3)        ..CHAR REPRESENTATION OF LENGTH
          BUFFER(2) = LOC 1
          CHKSUM = LOC 1        ..CHAR. REPRESENTATION OF CHECKSUM
          LOC 2= TOCHAR(ARG 2)
```

```
          BUFFER(3) =LOC 2     ..CHAR REPRESENTATION OF PACKET NUMBER
                               ..STORED
          CHKSUM = CHKSUM + LOC 2  ..ACCUMULATE CHECKSUM
          BUFFER(4) =(ARG 1)       ..PACKET TYPE
          CHKSUM = CHKSUM + ARG 1  ..ACCUMULATE CHECKSUM
          FOR I=1 TO ARG 3 DO BEGIN  ..ACCUMULATE DATA AND CHECKSUM
               BUFFER(I+4) = DATA(I)
               CHKSUM = CHKSUM +DATA(I)
               END
          LOC 3 = TOCHAR(CHECKSUM(CHKSUM))
          BUFFER(ARG 3 +5)= LOC 3  ..CHAR REPRESENTATION OF CHECKSUM
          BUFFER(ARG 3 +6)=EOL      ..EOL CHAR. WANTED BY OTHER END
          WRITCH(ARG 3 +6)
          I=1
     END
END       ..SPACK


  FUNC RINIT(0,1)
    BEGIN
    IF (NUMTRY GT 5) THEN LOC 1 = "A"   ..TOO MANY TRIES,SO ABORT
    ELSE BEGIN
     NUMTRY = NUMTRY + 1            ..BUMP TRY COUNT
     CASE RPACK OF                  ..GET A PACKET
         "S":: BEGIN                ..GOT A SEND-INIT
               RPAR              ..RETRIEVE PARAMETERS FROM SENDER
               DISPLAY("B")      ..MESSAGE FOR DEBUGGING PURPOSE
               SPAR              ..FILL UP PACKET WITH MY INFORMATION
               DISPLAY("C")
               SPACK("Y",L,6)    ..ACK WITH MY PACKET
               DISPLAY("D")
               OLDTRY = NUMTRY      ..SAVE OLD TRY COUNT
               NUMTRY = 0           ..START A NEW COUNTER
               L = (L + 1) MOD 64   ..BUMP COUNT, MOD 64
               LOC 1 = "F"          ..FILE SEND-STATE SAVED TEMPORARILY
          END
        "E" :: LOC 1 = STATE        ..DIDN'T GET PACKET; KEEP WAITING
         ELSE LOC 1= "A"            ..SOME OTHER TYPE, ABORT
    END
END
RETURN LOC 1             ..RETURN FILE-SEND STATE
END
END
```

```
.............................................
..              RECEIVE FILE  NAME
.............................................

FUNC RFILE(0,2)
  BEGIN
     IF (NUMTRY GT 5) THEN LOC 1 = "A"    ..ABORT IF TOO MANY TRIES
     ELSE BEGIN
       NUMTRY = NUMTRY + 1           ..BUMP COUNT
       CASE RPACK OF                 .. GET A PACKET
        "S" :: BEGIN      .. SEND-INIT, MAY BE "ACK" HAS BEEN LOST
               IF (OLDTRY GT 5) THEN LOC 1 = "A"    .. IF TOO MANY
                                     .. TRIES, THEN ABORT.
               ELSE BEGIN
                    OLDTRY = OLDTRY + 1   ..BUMP OLDTRY COUNT AS WELL
                    K = L - 1        ..PREVIOUS PACKET MOD 64 ?
                    IF (K LT 0) THEN K = 63
                    IF (NUM EQ K) THEN BEGIN    ..YES,SO "ACK" IT AGAIN
                         SPAR     ..SEND OUR SEND-INIT PACKET
                         SPACK("Y",NUM,6)
                         NUMTRY = 0      ..RESET  TRY COUNTER
                         LOC 1 = STATE    ..STAY IN THIS STATE
                    END
                    ELSE LOC 1 = "A"     ..NOT PREVIOUS PACKET,ABORT
               END
          END
         "Z" :: BEGIN          ..END-OF-FILE
               IF (OLDTRY GT 5) THEN LOC 1 = "A"
               ELSE BEGIN
                    OLDTRY = OLDTRY + 1
                    K = L - 1        ..PREVIOUS PACKET, MOD 64?
                    IF (K LT 0) THEN K = 63
                    IF (NUM EQ K) THEN BEGIN    ..YES, SO "ACK" IT AGAIN
                         SPACK ("Y",NUM,0)
                         NUMTRY = 0
                         LOC 1 = STATE          ..STAY IN THIS STATE
                    END
                    ELSE LOC 1 = "A"     ..NOT PREVIOUS PACKET, ABORT
               END
          END
        "F" :: BEGIN                      ..FILE -HEADER
               IF (NUM NE L) THEN LOC 1 = "A"    ..WHAT WE REALLY WANT
                              ..SO THE PACKET NUMBER MUST BE CORRECT
               ELSE BEGIN
                    IF (WRITEOPN NE 1) THEN LOC 1 = "A"
                                    ..TRY TO OPEN A NEW FILE
                    ELSE BEGIN      .. IF OK, THEN
                         SPACK("Y",L,0) .. "ACK" THE FILE-HEADER
                         OLDTRY = NUMTRY  .. RESET COUNTERS
```

```
                    NUMTRY = 0
                    L = (L+ 1) MOD 64   ..BUMP PACKET NUMBER MOD 64
                    LOC 1 = "D"     ..SWITCH TO DATA PACKET
                END
            END
        END
       "B" :: BEGIN                              ..BREAK TRANSMISSION
            IF (NUM NE L) THEN LOC 1 = "A"   ..NEED CORRECT PACKET
                                              .. NUMBER
            ELSE BEGIN
                 SPACK("Y",L,0)        .. SAY OK
                 LOC 1 = "C"           ..SWITCH TO COMPLETE STATE
            END
        END
      "E" :: LOC 1 = STATE     ..SHOULDN'T GET PACKET,KEEP TRYING
        ELSE LOC 1 = "A"       ..SOMETHING ELSE , ABORT
    END
END
END
RETURN LOC 1
END


FUNC RDATA(0,1)           ..RECEIVE DATA
    BEGIN
        IF (NUMTRY GT 5) THEN LOC 1= "A"    ..ABORT IF TOO MANY TRIES
        ELSE BEGIN
            NUMTRY = NUMTRY + 1       ..BUMP TRY COUNTER
            CASE RPACK OF             .. GET PACKET
               "D" :: BEGIN           .. GOT A DATA PACKET
                        IF (NUM NE L) THEN BEGIN
                                       ..LOOKS LIKE WRONG NUMBER
                           IF (OLDTRY GT 5) THEN LOC 1= "A"
                        ELSE BEGIN
                           OLDTRY =OLDTRY +1  ..BUMP OLDTRY COUNTER
                           K =L-1     ..SEE IF WE HAVE PREVIOUS
                                      ..PACKET AGAIN
                           IF (K LT 0) THEN K=63
                           IF (NUM EQ K) THEN BEGIN
                                              .. YES, GOT PREVIOUS ONE
                              SPACK("Y",NUM,0)  ..RE-"ACK" THE PACKET
                              NUMTRY =0      ..RESET TRY COUNTER
                              LOC 1 =STATE
                                ..STAY IN "D", DON'T WRITE OUT DATA
                           END
                        ELSE LOC 1 = "A"   ..SORRY,WRONG NUMBER
                        END
                      END
                      FOR I =1 TO LEN DO DATA(I):D
```

```
                                ..WRITE THE PACKET TO FILE
                        SPACK("Y",L,0)  ..ACKNOWLEDGE THE PACKET
                        OLDTRY =NUMTRY  ..RESET THE COUNTERS
                        NUMTRY =0
                        L = (L+1) MOD 64 ..COUNT PACKETS, MOD 64
                        LOC 1 = "D"     ..STAY IN THIS STATE
                      END
                "F" :: BEGIN        ..GOT A FILE HEADER
                        IF (OLDTRY GT 5) THEN LOC 1= "A"
                                ..TOO MANY TRIES ,SO QUIT
                        ELSE BEGIN
                             OLDTRY =OLDTRY + 1  ..BUMP TRY COUNTER
                             K =L -1    ..SEE IF PREVIOUS PACKET
                             IF (K LT 0) THEN K =63
                             IF (NUM EQ K) THEN BEGIN
                                      ..YES, SO "ACK" IT AGAIN
                                SPACK("Y",NUM,0)
                                NUMTRY = 0
                                LOC 1= STATE  ..STAY IN DATA STATE
                             END
                             ELSE LOC 1 = "A" ..NOT PREVIOUS PACKET,
                        END                    .. SO ABORT
                      END
                "Z" :: BEGIN
                          IF (NUM NE L) THEN LOC 1 = "A"
                                          ..MUST HAVE RIGHT PACKET
                          ELSE BEGIN
                             SPACK("Y",L,0)    ..OK, SO "ACK" IT
                             LIST(EOF)         ..CLOSE THE FILE
                             L =(L+1) MOD 64   ..BUMP PACKET COUNTER
                             LOC 1 = "F"    ..GO BACK TO RECEIVE FILE
                          END                .. STATE
                        END
              "E" :: LOC 1= STATE      ..NOTHING,KEEP WAITING
            ELSE LOC 1 = "A"     ..SOME OTHER TYPE, SO ABORT
        END        ..CASE
    END
  END
  RETURN LOC 1
END        ..RDATA


FUNC RECSW(0,2)           .. STATE TABLE SWITCHER FOR RECEIVING FILES
    LOC 2 =0              ..INITIALIZE
    STATE ="R"            ..ALWAYS START IN RECEIVE STATE
    L = 0                 .. INITIALIZE MESSAGE NUMBER
    NUMTRY =0             .. NO TRIES YET
    WHILE (LOC 2 NE 1) DO BEGIN    ..DO UNTIL DONE
      CASE STATE OF
```

```
        "R" :: STATE =RINIT          ..SEND INITIATE STATE
        "F" :: STATE =RFILE          ..FILE RECEIVE STATE
        "D" :: STATE =RDATA          ..DATA RECEIVE STATE
        "C" :: BEGIN                 .. COMPLETED STATE
               LOC 1 = 1
               LOC 2 = 1
               END
        "A" :: BEGIN                 ..ABORT STATE
               LOC 1 = 0
               LOC 2 = 1
               END
      END
    END
    'RECEIVE OVER '
  RETURN LOC 1
END

.....................................................................
.. INVOKES ALL THE PROCEDURES AND FUNCTIONS NECESSARY TO ACTIVATE
.. THE KERMIT ON THE TI-990/12 MICROCOMPUTER
.....................................................................

PROC KERMIT
INIT
GLOBAL
RECSW
END
```

APPENDIX B

```
*****************************************************************************
*   THE NAME OF SOURCE FILE IS 'AHS.SOURCE.FCCTST'
*   THIS PROGRAM INITIATES A POWER-UP REQUEST FOR THE FCCC AND
*   THE TILINE CHANNEL DATA TEST FOR THE FCCC
*****************************************************************************
*                TESTING THE FCCC

        IDT  'FCCTST'

        DATA WSP
        DATA START
        DATA 0
WSP     BSS  32

TBASE   EQU >F900          FCCC TILINE BASE ADDRESS
ROSW0   EQU >0             ROSW0  OFFSET
IPWRD   DATA >8000         INTERRUPT PENDING WORD
MRWRD   DATA >4000         MASTER RESET WORD
INRWRD  DATA >0800         INPUT READY WORD
OBFWRD  DATA >0400        *OUTPUT BUSY FLAG WORD
FAIWRD  DATA >0200        *FAIL FLAG WORD
TSTWRD  DATA >0100        *TEST FLAG WORD
ROSW1   EQU >2            *ROSW1(OFFSET)
ROSW2   EQU >4            *ROSW2(OFFSET)
ROSW3   EQU >6            *ROSW3(OFFSET)
*
*
*OUTPUT STATUS REGISTER
*
WOSW0   EQU 0             *WOSW0 (OFFSET)
IE  EQU  >2000            *INTERRUPT ENABLE
IEWRD   DATA  IE          *INTERRUPT ENABLE WORD
NIE     EQU  >1000        *NOT BUSY INTERRUPT ENABLE
WOSW1   EQU  2            *WOSW1 (OFFSET)
WOSW2   EQU  4            *WOSW2 (OFFSET)
WOSW3   EQU  6            *WOSW3 (OFFSET)
TSTCMD  DATA >F00         *TILINE TEST COMMAND WORD
DSRMAP  EQU  MAPFIL       *DSR MAP FILE ADDRESS
WDAAAA  DATA >AAAA        *FOR TILINE TEST
WDFF00  DATA >FF00        *CONSTANT
*
*    REQUEST BLOCK
*
REQBLK  EQU  $
MAPFIL  DATA 0            *MAP FILE ADDRESS OF BUFFER HEADER
BUFHPT  DATA 0            *POINTER TO BUFFER HEADER
OPCODE  BYTE 0            *OPCODE
CHNUMB  BYTE 0            *CHANNEL NUMBER OF FCCC
TIMOUT  BYTE 0            *TIME-OUT REQUEST
```

```
CCODE   BYTE 0          *COMPLETION CODE
*
*
* SAVE AREA FOR READ ONLY SLAVE WORDS
*

SWSAVE  DATA 0                  *ROSW0 SAVED FROM FCCC
        DATA 0                  *ROSW1 SAVED FROM FCCC
        DATA 0                  *ROSW2 SAVED FROM FCCC
        DATA 0                  *ROSW3 SAVED FROM FCCC



*                       SELF-IDENTIFICATION SVC
        EVEN
SCBJ    BYTE  >2E       SVC OPCODE
RTID    BYTE  0         SYSTEM RETURNS RUN-TIME ID HERE
INID    BYTE  0         SYSTEM RETURNS INSTALLED ID HERE
STID    BYTE  0         SYSTEM RETURNS STATION ID HERE
        DATA  0         SET TO ZERO

*******************************************************************************
*           PRIVELEGED SVC FOR READ/WRITE TSB.
*           SHOULD BE USED WITH EXTREME CAUTION WHILE WRITING INTO
*           THE TSB, AS THERE IS NO SYSTEM PROTECTION FOR THIS.
*******************************************************************************

        EVEN
RWTTSB  BYTE  >2C       PRIVILEGED  SVC OPCODE
        BYTE  0         RESERVED FOR ERROR CODE.
SVCFLG  DATA  0         READ TSB(TASK STATUS BLOCK)
RUNID   BYTE  0         RUN-TIME ID OF THE TASK WHOSE TSB IS TO BE READ
OFFSET  BYTE  >32       BYTE OFFSET IN THE TSB TO BE READ
VALUE   DATA  0         VALUE OF THE WORD READ FROM THE TSB IS RETURNED
        DATA  0
TEMP    DATA  0         SIX WORD SPACES ARE RESERVED
        DATA  0         FOR STORING THE WORDS READ
        DATA  0         FROM THE TSB
        DATA  0
        DATA  0
        DATA  0

*************************************************************

START   XOP   @SCBJ,15      INITIATES SELF-IDENTIFICATION SVC
        MOVB  @RTID,@RUNID
        LI    R3,TEMP
        LI    R2,>3200      MSB CONTAINS OFFSET OF THE WORD INTO TSB
```

```
*                                       TO BE READ
LOOP     XOP   @RWTTSB,15      INITIATE READING OF THE WORD FROM THE TSB
         MOV   @VALUE,*R3+     STORE WORD READ FROM TSB
         AI    R2,>200         TO POINT TO THE NEXT WORD IN THE TSB
         MOVB  R2,@OFFSET      GET THE NEXT OFFSET VALUE
         CI    R2,>3E00        OFFSET FOR THE LAST WORD TO BE READ.
         JNE   LOOP            CONTINUE UNTIL ALL THE SIX WORDS HAVE
         LI    R1,TEMP         BEEN READ.STORE SIX WORDS READ FROM TSB.
         LMF   R1,0            LOAD MEMORY MAP FILE 0 WITH 6 WORDS OF
*                               MEMORY, STARTING AT THE ADDRESS SPECIFIED
*                               IN REGISTER R1.


*****************************************************************
* DSR POWER-UP ENTRY
*THIS ROUTINE PROCESSES A POWER-UP REQUEST.
*IT IS ENTERED WITH INTERRUPTS MASKED TO LEVEL 2.
*****************************************************************

POWRUP   LI    R12,TBASE      *INITILIZE BASE REGISTER
         SOC   @MRWRD,*R12    *INITIATE FCCC MASTER RESET
LUPTST   MOV   *R12,R10       *GET ROSW0 CONTENTS
         COC   @TSTWRD,R10    *IS TEST OVER?
         JNE   TESTFF         *IF TRUE, CHECK FOR ERRORS
         LI    R6,20          *20 INTERVALS OF TIME OF .5 SEC EACH
         BL    @TOUT          *CHECK THE TIME-OUT
         JEQ   TERROR         *FCCC MASTER RESET TIMED OUT
         COC   @INRWRD,R10    *9903 CHANNEL ERROR DETECTED?
         JNE   LUPTST         *IF NOT, KEEP WAITING
*9903 SELF TEST ERROR HAS OCCURED.INPUT READY BIT
*_NEEDS TO BE CLEARED
*
         MOV   *R12,R2        *GET ROSW0 CONTENTS IN R2
         ANDI  R2,>0800
         MOV   R2,*R12        *CLEARS INPUT READY BIT
         JMP   LUPTST         *WAIT FOR END OF SELF TEST
TESTFF   COC   FAIWRD,R10     *HAS THE BOARD FAILED SELF-TEST
         JEQ   FAILUR         *YES,GIVE ERROR MESSAGE


*
*  WAIT FOR END OF MASTER RESET SEQUENCE
*

CKRSET   MOV   *R12,R10       *GET ROSW0 CONTENTS
         CZC   @MRWRD,R10     *IS MASTER RESET COMPLETE?
         JEQ   RESCMP         *YES,ISSUE TILINE TEST
         BL    @TOUT          *NO, CHECK TIME-OUT
```

```
        JNE  CKRSET
        JMP  TERROR
*
*TIME-OUT SUBROUTINE STARTS HERE
*


TOUT    LI   R3,>8B82
DEL     DEC  R3
        JNE  DEL       *TO CREATE A DELAY OF 0.5 SEC
        DEC  R6        *0.5 SEC DELAY CONTINUED UPTO 20 TIMES
        B    *R11
*
*
*FOLLOWING POWER-UP OR MASTER RESET,TILINE CHANNEL DATA TEST
*(COMMAND OF) MUST BE THE FIRST COMMAND TO BE ISSUED TO THE
*FCCC.THE FCCC FIRMWARE PERFORMS THIS TEST BY WRITING 4 DATA
*CONSTANTS TO A HOST MEMORY ADDRESS OVER A TILINE DATA CHANNEL
*THE TILINE READS THESE WORDS AND CHECKS TO SEE IF THE VALUES
*READ ARE THE SAME AS THE VALUES WRITTEN.IF BOTH THE VALUES
*ARE THE SAME,THEN THE TEST IS SUCCESSFUL
*
*THE FOLLOWING IS THE CODING FOR THE TILINE TEST WHICH MAKES
*USE OF A ROUTINE (CALADD) TO CALCULATE THE REAL MEMORY
*ADDRESS OF A WORD IN MEMORY.
*
*
RESCMP  COC   @OBFWRD,R10    *IS FCCC READY TO ACCEPT A COMMAND?
        JEQ   FAILUR         *NO,FCCC IS BROKEN
        LI    R5,DSRMAP      *FETCH DSR MAP FILE ADDRESS
        LI    R6,SWSAVE      *POINT TO TABLE ROSW0 SAVE AREA
        CLR   *R6            *MAKE SURE TEST WORD IS ZERO
        BL    @CALADD        *CALCULATE REAL MEMORY ADDRESS
        JMP   POWRXT         *ERROR RETURN
C1      B     @IDL            SHOW THE MESSAGE
        SWPB  R5             *PUT LSB IN LOW ORDER BYTE
        MOV   R5,@WOSW1(R12)  *ISSUE COMMAND
        MOV   R6,@WOSW2(R12)
        MOV   @TSTCMD,@WOSW3(R12)    *TILINE TEST COMMAND
*
* WAIT FOR TILINE TEST TO COMPLETE
*
*

LUPTLT  MOV   *R12,R10       *READ ROSW0
        COC   @INRWRD,R10    *HAS TEST COMPLETED?
        JNE   LUPTLT         *NO,WAIT (TILINE TEST WILL TERMINATE)
        C     @WDAAAA,@SWSAVE  *IS FIRST WORD OF TEST
*                             *BUFFER = >AAAA?
```

```
        JNE   FAILUR            *NO, THIS IS A CRITICAL ERROR
        COC   @FAIWRD,R10       *TILINE TEST FAILURE DETECTED?
        JEQ   FAILUR            *YES,SHOW ERROR MESSAGE.
        SOC   @INRWRD,@ROSW0(R12)
*                               *NO,CLEAR INPUT READY INTERRUPT
POWRXT  RTWP
*
*THIS SUBROUTINE CALCULATES A 21-BIT REAL MEMORY ADDRESS
*FROM THE TASK MAP FILE AND TASK RELATIVE ADDRESS.
*

CALADD  MOV   R5,R10              *IS MAP FILE SPECIFIED?
        JEQ   CALXIT            *NO,RETURN WITH CALLING VALUE
        LI    R7,3              *LOAD SEGMENT COUNTER
CALAD1  MOV   *R10+,R9          *LOAD LIMIT REGISTER
        INV   R9                *AND SET MAXIMUM ADDRESS
        C     R6,R9             *IS THIS THE CORRECT BASE?
        JLE   CALAD2            *YES,NOW CALCULATE THE ADDRESS
        INCT  R10               *SKIP BASE INFORMATION
        DEC   R7                *IS THIS THE END?
        JGT   CALAD1            *NO,CONTINUE PROCESSING

*
*ILLEGAL ADDRESS,USE ERROR RETURN
*
        JMP   CERRXT            *SET ERROR RETURN
*
*
*THE CORRECT BASE HAS BEEN FOUND,NOW MAKE A 21-BIT ADDRESS
*
*
CALAD2  MOV   R6,R9             *LOAD ADDRESS
        AI    R7,-3             *HOW MANY TIMES THROUGH LOOP?
        JEQ   CALAD3            *ONLY ONE,SKIP BIAS ADJUSTMENT
        MOV   @-6(R10),R7       *FETCH ADJUSTMENT FOR BIAS
        A     R7,R9             *THEN CALCULATE OFFSET IN SEGMENT
        NEG   R7
        SRL   R7,R5             *FETCH SEGMENT OFFSET
CALAD3  MOV   *R10,R5           *THEN FETCH SEGMENT BIAS
        A     R7,R5             *AND ADJUST BIAS
        MOV   R5,R6             *COPY FOR PROCESSING
        SLA   R6,5              *THEN SHIFT FOR BIAS
        SRL   R5,11             *AND SET UP IN HIGH ORDER BYTE
        A     R9,R6             *INCLUDE OFFSET
        JNC   CALXIT            *NO CARRY INCLUDED
        INC   R5                *BUMP MSB ON CARRY
*
*THE CORRECT ADDRESS HAS BEEN PROCESSED
*
```

```
*
CALXIT  SWPB  R5              *SET IN HIGH ORDER BYTE
        INCT  R11             *BUMP PAST ERROR RETURN
CERRXT  B     *R11            *SO RETURN TO CALLER
TERROR  JMP   ERR1
FAILUR  B     @ERR2
*****************************************************************************
*       SVC DATA BLOCKS TO PRINT MESSAGES ON THE SCREEN                     *
*       START HERE. THE VDT IS ASSIGNED A LUNO BY THE SYSTEM.               *
*****************************************************************************
ASGN    DATA  0               I/O REQUEST
        BYTE  >91             ASSIGN  LUNO
NUMB    BYTE  0
        DATA  0
        DATA  0
        DATA  0
        DATA  0
        DATA  0
        DATA  0
        DATA  0
        BYTE  >04
        DATA  0
        DATA  0
        BYTE  0
        DATA  NAME
NAME    BYTE  >04
        TEXT  'ST12'
        DATA  0
        DATA  0
OPEN    DATA  0               I/O  REQUEST
        BYTE  0               OPEN
N1      BYTE  0               LUNO
        DATA  0
        DATA  0
        DATA  0
        DATA  0
MSSG0   DATA  0               I/O  REQUEST
        BYTE  >B              WRITE  ASCII
N2      BYTE  0               ON  LUNO
        DATA  0
        DATA  ERROR           MESSAGE LOCATION
        DATA  0
        DATA  MSSG1-ERROR     MESSAGE LENGTH
*****************************************************************
*       SPECIFY THE MESSAGE:-MASTER RESET HAS FAILED            *
*****************************************************************
ERROR   DATA  >0A0D
        TEXT  'MASTER RESET HAS FAILED'
        DATA  >0A0D
MSSG1   DATA  0               I/O  REQUEST
```

```
        BYTE    >B              WRITE ASCII
N3      BYTE    0               ON LUNO
        BYTE    0,>40
        DATA    FATAL           MESSAGE LOCATION
        DATA    0
        DATA    15              MASSAGE LENGTH
        DATA    STR1            LOCATION OF INPUT PARAMETERS
STR1    DATA    STORE           SAVE PARAMETERS IN STORE
        DATA    4               STORE 4 CHARACTERS
        DATA    0               CHARACTER COUNT AFTER INPUT
STORE   BSS     12
*****************************************************************
*       SPECIFY  THE MESSAGE :- FATAL ERROR                     *
*****************************************************************
FATAL   DATA    >0A0D
        TEXT    'FATAL ERROR'
        DATA    >0A0D
MSSG2   DATA    0               I/O  REQUEST
        BYTE    >B
N4      BYTE    0               WRITE ASCII ON LUNO
        BYTE    0,>40
        DATA    FINE            MESSAGE  LOCATION
        DATA    0
        DATA    15              MESSAGE LENGTH
        DATA    STR2
STR2    DATA    MEM
        DATA    4
        DATA    0
MEM     BSS     12
*****************************************************************
*       SPECIFY THE MESSAGE :- FCCC IS WORKING FINE             *
*****************************************************************
FINE    DATA    >0A0D
        TEXT    'FCCC IS WORKING FINE'
        DATA    >0A0D
CLOSE   DATA    0               I/O  REQUEST
        BYTE    01              CLOSE
N5      BYTE    0               LUNO
        DATA    0
        DATA    0
        DATA    0
        DATA    0
EOP     BYTE    >16,0           TERMINATE  TASK
        MOVB    @NUMB,@N1
        MOVB    @NUMB,@N2
        MOVB    @NUMB,@N3
        MOVB    @NUMB,@N4
        MOVB    @NUMB,@N5
ERR1    NOP                     ASSIGN  LUNO
```

```
        XOP     @OPEN,15        OPEN  LUNO
        XOP     @MSSG0,15       WRITE MESSAGE
        XOP     @CLOSE,15       CLOSE LUNO
        XOP     @EOP,15         TERMINATE TASK
        JMP     B1
ERR2    NOP
        XOP     @OPEN,15
        XOP     @MSSG1,15
        XOP     @CLOSE,15
        XOP     @EOP,15
        JMP     B1
IDL     NOP
        XOP     @OPEN,15
        XOP     @MSSG2,15
        XOP     @CLOSE,15
        RTWP
B1      END
```

PROGRAM FOR TRANSMISSION OF DATA THROUGH FCCC

```
*****************************************************************
*  THE NAME OF THE SOURCE FILE IS 'AHS.SOURCE.TRANSMIT'
*  THIS IS A PROGRAM FOR PROCESSING A 'WRITE' REQUEST
*  USING THE FCCC.IT IS ASSUMED THAT THE ASCII DECIMAL
*  VALUES OF THE CHARACTERS TO BE TRANSMITTED ARE STORED
*  IN THE MEMORY,STARTING FROM A KNOWN MEMORY LOCATION.
*****************************************************************

          IDT     'TRANSMIT'

          DATA    WSP
          DATA    START
          DATA    0
WSP       BSS     32



ERWRD     DATA    >8000          ERROR FLAG WORD
IEWRD     DATA    >2000          INTERRUPT ENABLE WORD
OBFWRD    DATA    >0400          OUTPUT BUSY FLAG WORD
INRWRD    DATA    >0800          INPUT READY WORD
ROSW0     EQU     0              ROSW 0 (OFFSET)
ROSW1     EQU     2              ROSW 1 (OFFSET)
ROSW2     EQU     4              ROSW 2 (OFFSET)
ROSW3     EQU     6              ROSW 3 (OFFSET)
WOSW0     EQU     0              WOSW 0 (OFFSET)
WOSW1     EQU     2              WOSW 1 (OFFSET)
WOSW2     EQU     4              WOSW 2 (OFFSET)
WOSW3     EQU     6              WOSW 3 (OFFSET)


*****************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED        *
*       'VOL1.RES.RES560.ERR1'                                  *
*****************************************************************

          EVEN
ERR1      DATA    0
          BYTE    >90,0,0,0          CREATE FILE
          DATA    0,0,0,0,0
          BYTE    0,>8D
          DATA    40
          DATA    288                     PHYSICAL RECORD LENGTH
          DATA    PATH1                   PATHNAME ADDRESS
```

```
         DATA   0,0
         DATA   0,1000                  INITIAL ALLOCATION
         DATA   0,500                   SECONDARY ALLOCATION
PATH1    BYTE   NAME1-$-1               PATHNAME LENGTH
         TEXT   'VOL1.RES.RES560.ERR1'  PATHNAME
NAME1    EQU    $

*****************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                    *
*       'VOL1.RES.RES560.ERR2'                                              *
*****************************************************************************

         EVEN
ERR2     DATA   0
         BYTE   >90,0,0,0          CREATE FILE
         DATA   0,0,0,0,0
         BYTE   0,>8D
         DATA   40
         DATA   288                     PHYSICAL RECORD LENGTH
         DATA   PATH2                   PATHNAME ADDRESS
         DATA   0,0
         DATA   0,1000                  INITIAL ALLOCATION
         DATA   0,500                   SECONDARY ALLOCATION
PATH2    BYTE   NAME2-$-1               PATHNAME LENGTH
         TEXT   'VOL1.RES.RES560.ERR2'  PATHNAME
NAME2    EQU    $

*****************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                    *
*       'VOL1.RES.RES560.ERR3'                                              *
*****************************************************************************

         EVEN
ERR3     DATA   0
         BYTE   >90,0,0,0          CREATE FILE
         DATA   0,0,0,0,0
         BYTE   0,>8D
         DATA   40
         DATA   288
         DATA   PATH3
         DATA   0,0
         DATA   0,1000
         DATA   0,500
PATH3    BYTE   NAME3-$-1
         TEXT   'VOL1.RES.RES560.ERR3'
NAME3    EQU    $
```

```
*********************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                        *
*       'VOL1.RES.RES560.CORR1'                                                 *
*********************************************************************************

          EVEN
CORR1     DATA   0
          BYTE   >90,0,0,0          CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288
          DATA   PATH4
          DATA   0,0
          DATA   0,1000
          DATA   0,500
PATH4     BYTE   NAME4-$-1
          TEXT   'VOL1.RES.RES560.CORR1'
NAME4     EQU    $

*********************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                        *
*       'VOL1.RES.RES560.CORR2'                                                 *
*********************************************************************************

          EVEN
CORR2     DATA   0
          BYTE   >90,0,0,0          CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288
          DATA   PATH5
          DATA   0,0
          DATA   0,1000
          DATA   0,500
PATH5     BYTE   NAME5-$-1
          TEXT   'VOL1.RES.RES560.CORR2'
NAME5     EQU    $

*********************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                        *
*       'VOL1.RES.RES560.CORR3'                                                 *
*********************************************************************************

          EVEN
CORR3     DATA   0
```

```
          BYTE   >90,0,0,0          CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288
          DATA   PATH6
          DATA   0,0
          DATA   0,1000
          DATA   0,500
PATH6     BYTE   NAME6-$-1
          TEXT   'VOL1.RES.RES560.CORR3'
NAME6     EQU    $


*********************************************************************************
*         CHANNEL PARAMETERS DATA BLOCK STARTS FROM LOCATION CPPROS.
*********************************************************************************

          EVEN
BUFAD     DATA   >8022
          DATA   >08
          DATA   0
          DATA   0
CPPROS    DATA   >36E0     SELECT 300 B.P.S.;CONTINUOUS RTS,FULL DUPLEX
CPXOTC    BYTE   >02       TRANSMIT INTER-CHARACTER TIME-OUT=2*0.25=0.5SEC
CPROTC    BYTE   >02       RECEIVE  INTER-CHARACTER TIME-OUT
CPCRUI    DATA   1         ONE BYTE / CHARACTER.
CPCNTL    DATA   >09A2     TMS 9903 CONTROL WORD; SELECTS 1STOP BIT,7 BITS
*                          ING; f =1 MHZ.
CPSYN1    BYTE   0
CPSYN2    BYTE   0
CPSPCL    BYTE   >80       TMS 9903 TRANSPARENCY SELECTED.
CPSCNT    BYTE   0         NOT APPLICABLE, SINCE ASYNCHRONOUS PROTOCOL IS
CPRECL    DATA   0         SELECTED.
CPISRF    BYTE   04,04     ASYNCHRONOUS MODE
CPIDLE    BYTE   >41,0     IDLE FILL CHARACTER IS "A"
CPINTR    DATA   0,0,0,0
          DATA   0,0,0,0


*********************************************************************
*         DATA TRANSMISSION STARTS FROM ADDRESS (XMTDAT +8)
*********************************************************************

          EVEN
XMTDAT    DATA   >8100       >100 (HEXADECIMAL) BYTES OF DATA TRANSM.
          DATA   >08           OFFSET
```

```
DATA    >0F00
DATA    0
DATA    >0D0D     TRANSMISSION OF DATA  STARTS HERE.THE VALUES
DATA    >0D0D     OF DATA ARE IN ASCII
DATA    >430D
DATA    >440D
DATA    >450D
DATA    >460D
DATA    >470D
DATA    >480D
DATA    >490D
DATA    >4A0D
DATA    >4B0D
DATA    >4C0D
DATA    >4D0D
DATA    >4E0D
DATA    >4F0D
DATA    >500D
DATA    >510D
DATA    >520D
DATA    >530D
DATA    >540D
DATA    >550D
DATA    >560D
DATA    >570D
DATA    >580D
DATA    >590D
DATA    >5A0D
DATA    >300D
DATA    >310D
DATA    >320D
DATA    >330D
DATA    >340D
DATA    >350D
DATA    >360D
DATA    >370D
DATA    >380D
DATA    >390D
DATA    >412C,>420D
DATA    >432C,>440D
DATA    >452C,>460D
DATA    >472C,>480D
DATA    >492C,>4A0D
DATA    >4B2C,>4C0D
DATA    >4D2C,>4E0D
DATA    >4F2C,>500D
DATA    >512C,>520D
DATA    >532C,>540D
DATA    >552C,>560D
```

```
DATA    >572C,>580D
DATA    >592C,>5A0D
DATA    >302C,>300D
DATA    >312C,>310D
DATA    >322C,>320D
DATA    >332C,>330D
DATA    >342C,>340D
DATA    >352C,>350D
DATA    >362C,>360D
DATA    >372C,>370D
DATA    >382C,>380D
DATA    >392C,>390D
DATA    >3030,>300D
DATA    >3131,>310D
DATA    >3232,>320D
DATA    >3333,>330D
DATA    >3434,>340D
DATA    >3535,>350D
DATA    >3636,>360D
DATA    >3737,>370D
DATA    >3838,>380D
DATA    >3939,>390D
DATA    >4141,>410D
DATA    >4242,>420D
DATA    >4343,>430D
DATA    >4444,>440D
DATA    >4545,>450D
DATA    >4646,>460D
DATA    >4747,>470D
DATA    >4848,>480D
DATA    >4949,>490D
DATA    >4A4A,>4A0D
DATA    >4B4B,>4B0D
DATA    >4C4C,>4C0D
DATA    >4D4D,>4D0D
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
DATA    >A0A0,>A0A0
```

```
          DATA   >A0A0,>A0A0
          DATA   >A0A0,>A0A0
          DATA   >A0A0,>A0A0
          DATA   >A0A0,>A0A0
          DATA   >A0A0,>A0A0
          DATA   >A0A0,>A0A0

*****************************************************
*                    SELF-IDENTIFICATION SVC
*****************************************************

          EVEN
SCBJ      BYTE   >2E        SVC OPCODE
RTID      BYTE   0          SYSTEM RETURNS RUN-TIME ID HERE
INID      BYTE   0          SYSTEM RETURNS INSTALLED ID HERE
STID      BYTE   0          SYSTEM RETURNS STATION ID HERE
          DATA   0          SET TO ZERO

*****************************************************************************
*            PRIVELEGED SVC FOR READ/WRITE TSB.
*            SHOULD BE USED WITH EXTREME CAUTION WHILE WRITING INTO
*            THE TSB, AS THERE IS NO SYSTEM PROTECTION FOR THIS.
*****************************************************************************

          EVEN
RWTTSB    BYTE   >2C        PRIVILEGED  SVC OPCODE
          BYTE   0          RESERVED FOR ERROR CODE.
SVCFLG    DATA   0          READ TSB(TASK STATUS BLOCK)
RUNID     BYTE   0          RUN-TIME ID OF THE TASK WHOSE TSB IS TO BE READ
OFFSET    BYTE   >32        BYTE OFFSET IN THE TSB  BE READ
VALUE     DATA   0          VALUE OF THE WORD READ FROM THE TSB IS RETURNED
          DATA   0
TEMP      DATA   0          SIX WORD SPACES ARE RESERVED
          DATA   0          FOR STORING THE WORDS READ
          DATA   0          FROM THE TSB
          DATA   0
          DATA   0
          DATA   0

*****************************************************

START     XOP    @SCBJ,15       INITIATES SELF-IDENTIFICATION SVC
          MOVB   @RTID,@RUNID
          LI     R3,TEMP
          LI     R2,>3200       MSB CONTAINS OFFSET OF THE WORD INTO TSB
*                                TO BE READ
REPEAT    XOP    @RWTTSB,15     INITIATE READING OF THE WORD FROM THE TSB
```

```
          MOV   @VALUE,*R3+     STORE WORD READ FROM TSB
          AI    R2,>200         TO POINT TO THE NEXT WORD IN THE TSB
          MOVB  R2,@OFFSET      GET THE NEXT OFFSET VALUE
          CI    R2,>3E00        OFFSET FOR THE LAST WORD TO BE READ.
          JNE   REPEAT          CONTINUE UNTIL ALL THE SIX WORDS HAVE
          LI    R1,TEMP         BEEN READ.STORE SIX WORDS READ FROM TSB.
          LMF   R1,0            LOAD MEMORY MAP FILE 0 WITH 6 WORDS OF
*                               MEMORY, STARTING AT THE ADDRESS SPECIFIED
*                               IN REGISTER R1.

*****************************************************************************
*   CHECKING THE STATE OF THE BOARD BEFORE ISSUING A COMMAND *
*****************************************************************************

          LI    R12,>F900       POINT TO THE FCCC
          CLR   R0
LOOP      SZC   @IEWRD,@WOSW0(R12)  DISABLE FCCC INTERRUPTS TO THE HOST
          MOV   *R12,R10        RECOVER ROSW0 CONTENT
DEL1      COC   @OBFWRD,R10     WILL THE BOARD TAKE A COMMAND ?
          JEQ   DEL1            NO, THE BOARD IS BUSY.WAIT UNTIL
A1        CI    R0,1            THE FLAG IS CLEARED.
          JEQ   A2
          JMP   A3
A2        B     *R11            RETURN ADDRESS FOR THE SUBROUTINE

*****************************************************************************
*  THE BOARD IS READY AT THIS POINT FOR WRITING INTO THE    *
*  SLAVE WORDS(WOSW0 THROUGH WOSW3).INITIATE THE OPEN       *
*  CHANNEL COMMAND TO THE FCCC.                             *
*****************************************************************************

A3        LI    R12,>F900
          LI    R5,>0F05        TIME-OUT= 250*15 mSEC.,SELECTS UPA
*                               PROTOCOL
          LI    R6,>2100        OPEN CHANNEL 0;
          MOV   R5,@WOSW1(R12)
          MOV   R6,@WOSW3(R12)
          LI    R10,IEWRD       SET UP TO ENABLE INTERRUPT
          SOC   R10,@WOSW0(R12) ENABLE INTERRUPTS;ALLOWS HOST TO BE
*                               INTERRUPTED BY THE FCCC.

*****************************************************************************
*       TEST IF THE COMMAND HAS BEEN EXECUTED                *
*****************************************************************************

          BL    @CHECK
          XOP   @ERR1,15        CREATE A SEQUENTIAL FILE
          B     @STOP           IF ERROR, BRANCH TO END OF PROGRAM
          XOP   @CORR1,15       ELSE CREATE ANOTHER SEQUENTIAL FILE
```

```
*****************************************************************************
*       OPEN CHANNEL COMMAND HAS PERFORMED CORRECTLY.                       *
*       FCCC KNOWS WHERE ITS CHANNEL PARAMETERS TABLE IS                    *
*       LOCATED FOR A KNOWN PROTOCOL.NOW,INITIATE A WRITE                   *
*       CHANNEL PARAMETERS COMMAND TO MODIFY THE PARAMETERS                 *
*       ON THE TABLE, OTHERWISE,DEFAULT VALUES FOR THE GIVEN                *
*       PROTOCOL WILL BE TAKEN.                                             *
*****************************************************************************

          LI      R12,>F900
          LI      R0,1
          BL      @LOOP
          CLR     R0
          LI      R5,>0500        HIGH BYTE SELECTS UPA PROTOCOL
          LI      R6,BUFAD        LSW OF HOST DATA BUFFER ADDRESS
          LI      R7,>7300        OPCODE FOR WRITE CHANNEL PARAMETERS
*                                 COMMAND FOR CHANNEL 0
          MOV     R5,@WOSW1(R12)
          MOV     R6,@WOSW2(R12)
          MOV     R7,@WOSW3(R12)
          LI      R0,IEWRD        SET UP TO ENABLE INTERRUPTS AGAIN
          SOC     R10,@ROSW0(R12)    ENABLE INTERRUPTS

*****************************************************************************
*       TEST TO EXAMINE IF THE COMMAND HAS BEEN EXECUTED                    *
*       CORRECTLY                                                           *
*****************************************************************************

          BL      @CHECK
          XOP     @ERR2,15        IF ERROR,CREATE A SEQUENTIAL FILE AND
          B       @STOP           BRANCH TO THE END OF THE PROGRAM
          XOP     @CORR2,15       OTHERWISE, CREATE ANOTHER SEQUENTIAL FILE

*****************************************************************************
*       WRITE PARAMETERS COMMAND HAS BEEN EXECUTED CORRECTLY                *
*       NOW ISSUE THE WRITE/READ COMMAND.                                   *
*****************************************************************************

          LI      R5,>50          NO. OF TIMES TO PERFORM WRITING
          MOV     R5,R1
WRITE     LI      R12,>F900
          LI      R0,1
          BL      @LOOP           CHECK TO SEE THE STATE OF THE BOARD
*                                 BEFORE ISSUING A COMMAND
          CLR     R0
          LI      R5,>0F00        TIME-OUT=15*250 mSEC.;MSB ADDRESS OF
*                                 DATA BUFFER IS ZERO.
```

```
        LI      R6,XMTDAT       LSW(LEAST SIGNIFICANT WORD) ADDRESS OF
*                               HOST DATA BUFFER CONTAINING THE DATA TO
*                               BE TRANSMITTED
        LI      R7,>0300        OPCODE FOR WRITE COMMAND;BRANCHING TO
*                               INITIAL CHARACTER DETECTION(CD) TRANSMIT
*                               STATE WITH MODIFIER 0 FOR CHANNEL 0
        MOV     R5,@WOSW1(R12)
        MOV     R6,@WOSW2(R12)
        MOV     R7,@WOSW3(R12)
        LI      R10,IEWRD
        SOC     R10,@WOSW0(R12)     ENABLE FCCC INTERRUPTS TO THE HOST

*****************************************************************
*       CHARACTER DETECTION ROUTINE AND THE INTERRUPT SERV- *
*       ICE ROUTINE WILL BE HANDLED BY THE FCCC FIRMWARE.   *
*       COMPLETION STATUS IS REPORTED IN ROSW0 ( BITS 8     *
*       THROUGH 15).COMPLETION STATUS IN ROSW0 INDICATES AN *
*       ERROR ONLY WHEN THE ERROR FLAG IS SET (ROSW3, BIT 0)*
*****************************************************************

TEST    MOV     @ROSW3(R12),R6
        COC     @ERWRD,R6           HAS ANY ERROR OCCURED?
        JEQ     A4                  YES, JUMP TO CREATE AN ERROR FILE
        DEC     R1                  NO, CONTINUE WRITING
        JNE     WRITE
        XOP     @CORR3,15           CREATE SEQUENTIAL FILE
        B       @STOP
A4      XOP     @ERR3,15            CREATE SEQUENTIAL FILE
        B       @STOP

*****************************************************************
*       SUBROUTINE CHECK STARTS HERE                            *
*****************************************************************

CHECK   MOV     @ROSW3(R12),R6      READ ROSW3 INFORMATION AFTER COMMAND
*                                   EXECUTION
        COC     @ERWRD,R6           HAS ANY ERROR OCCURED?
        JEQ     A6                  YES, JUMP TO A6
        AI      R11,4
        B       *R11                RETURN FOR CORRECT COMMAND EXECUTION
A6      B       *R11                ERROR RETURN
STOP    END
```

PROGRAM FOR RECEIVING DATA THROUGH FCCC

```
*****************************************************************
*   THE NAME OF SOURCE FILE IS 'AHS.TEST.READ'.
*****************************************************************

          IDT    'RECEIVE'

          DATA   WSP
          DATA   START
          DATA   0
WSP       BSS    32


ERWRD     DATA   >8000          ERROR FLAG WORD
IEWRD     DATA   >2000          INTERRUPT ENABLE WORD
OBFWRD    DATA   >0400          OUTPUT BUSY FLAG WORD
INRWRD    DATA   >0800          INPUT READY WORD
ROSW0     EQU    0              ROSW 0 (OFFSET)
ROSW1     EQU    2              ROSW 1 (OFFSET)
ROSW2     EQU    4              ROSW 2 (OFFSET)
ROSW3     EQU    6              ROSW 3 (OFFSET)
WOSW0     EQU    0              WOSW 0 (OFFSET)
WOSW1     EQU    2              WOSW 1 (OFFSET)
WOSW2     EQU    4              WOSW 2 (OFFSET)
WOSW3     EQU    6              WOSW 3 (OFFSET)


*****************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED        *
*       'VOL1.RES.RES560.ERR1'                                  *
*****************************************************************

          EVEN
ERR1      DATA   0
          BYTE   >90,0,0,0       CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288                  PHYSICAL RECORD LENGTH
          DATA   PATH1                PATHNAME ADDRESS
          DATA   0,0
          DATA   0,1000               INITIAL ALLOCATION
          DATA   0,500                SECONDARY ALLOCATION
```

```
PATH1    BYTE   NAME1-$-1                PATHNAME LENGTH
         TEXT   'VOL1.RES.RES560.ERR1'   PATHNAME
NAME1    EQU    $

*****************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                    *
*       'VOL1.RES.RES560.ERR2'                                              *
*****************************************************************************

         EVEN
ERR2     DATA   0
         BYTE   >90,0,0,0          CREATE FILE
         DATA   0,0,0,0,0
         BYTE   0,>8D
         DATA   40
         DATA   288                     PHYSICAL RECORD LENGTH
         DATA   PATH2                   PATHNAME ADDRESS
         DATA   0,0
         DATA   0,1000                  INITIAL ALLOCATION
         DATA   0,500                   SECONDARY ALLOCATION
PATH2    BYTE   NAME2-$-1               PATHNAME LENGTH
         TEXT   'VOL1.RES.RES560.ERR2'  PATHNAME
NAME2    EQU    $

*****************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                    *
*       'VOL1.RES.RES560.ERR3'                                              *
*****************************************************************************

         EVEN
ERR3     DATA   0
         BYTE   >90,0,0,0          CREATE FILE
         DATA   0,0,0,0,0
         BYTE   0,>8D
         DATA   40
         DATA   288
         DATA   PATH3
         DATA   0,0
         DATA   0,1000
         DATA   0,500
PATH3    BYTE   NAME3-$-1
         TEXT   'VOL1.RES.RES560.ERR3'
NAME3    EQU    $
```

```
*********************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                        *
*       'VOL1.RES.RES560.CORR1'                                                 *
*********************************************************************************

          EVEN
CORR1     DATA   0
          BYTE   >90,0,0,0          CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288
          DATA   PATH4
          DATA   0,0
          DATA   0,1000
          DATA   0,500
PATH4     BYTE   NAME4-$-1
          TEXT   'VOL1.RES.RES560.CORR1'
NAME4     EQU    $

*********************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                        *
*       'VOL1.RES.RES560.CORR2'                                                 *
*********************************************************************************

          EVEN
CORR2     DATA   0
          BYTE   >90,0,0,0          CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288
          DATA   PATH5
          DATA   0,0
          DATA   0,1000
          DATA   0,500
PATH5     BYTE   NAME5-$-1
          TEXT   'VOL1.RES.RES560.CORR2'
NAME5     EQU    $

*********************************************************************************
*       SVC DATA BLOCK TO CREATE A SEQUENTIAL FILE NAMED                        *
*       'VOL1.RES.RES560.CORR3'                                                 *
*********************************************************************************

          EVEN
CORR3     DATA   0
```

```
          BYTE   >90,0,0,0          CREATE FILE
          DATA   0,0,0,0,0
          BYTE   0,>8D
          DATA   40
          DATA   288
          DATA   PATH6
          DATA   0,0
          DATA   0,1000
          DATA   0,500
PATH6     BYTE   NAME6-$-1
          TEXT   'VOL1.RES.RES560.CORR3'
NAME6     EQU    $


*****************************************************************************
*         CHANNEL PARAMETERS DATA BLOCK STARTS FROM LOCATION CPPROS.
*****************************************************************************

          EVEN
BUFAD     DATA   >8022
          DATA   >08
          DATA   0
          DATA   0
CPPROS    DATA   >36E0     SELECT 300 B.P.S.;CONTINUOUS RTS,FULL DUPLEX
CPXOTC    BYTE   >02       TRANSMIT INTER-CHARACTER TIME-OUT=2*0.25=0.5SEC
CPROTC    BYTE   >02       RECEIVE  INTER-CHARACTER TIME-OUT
CPCRUI    DATA   1         ONE BYTE / CHARACTER.
CPCNTL    DATA   >09A2     TMS 9903 CONTROL WORD; SELECTS 1STOP BIT,7 BITS

*                          ING; f =1 MHZ.
CPSYN1    BYTE   0
CPSYN2    BYTE   0
CPSPCL    BYTE   >80       TMS 9903 TRANSPARENCY SELECTED.
CPSCNT    BYTE   0         NOT APPLICABLE, SINCE ASYNCHRONOUS PROTOCOL IS
CPRECL    DATA   0         SELECTED.
CPISRF    BYTE   04,04     ASYNCHRONOUS MODE
CPIDLE    BYTE   >41,0     IDLE FILL CHARACTER IS "A"
CPINTR    DATA   0,0,0,0
          DATA   0,0,0,0


*****************************************************************************
*         DATA RECEPTION STARTS FROM ADDRESS (RCVDAT +8)
*****************************************************************************

          EVEN
RCVDAT    DATA   >8FFF        A MAXIMUM OF >FFF (HEXADECIMAL) BYTES OF
*                             DATA CAN BE RECEIVED;USES ONLY 1 BUFFER.
```

```
          DATA   >08            OFFSET
          DATA   >0F00
          DATA   0
DATRD     DATA   >0000     DATA RECEPTION  STARTS HERE

*****************************************************
*                    SELF-IDENTIFICATION SVC
*****************************************************

          EVEN
SCBJ      BYTE   >2E         SVC OPCODE
RTID      BYTE   0           SYSTEM RETURNS RUN-TIME ID HERE
INID      BYTE   0           SYSTEM RETURNS INSTALLED ID HERE
STID      BYTE   0           SYSTEM RETURNS STATION ID HERE
          DATA   0           SET TO ZERO

*****************************************************************************
*            PRIVELEGED SVC FOR READ/WRITE TSB.
*            SHOULD BE USED WITH EXTREME CAUTION WHILE WRITING INTO
*            THE TSB, AS THERE IS NO SYSTEM PROTECTION FOR THIS.
*****************************************************************************

          EVEN
RWTTSB    BYTE   >2C         PRIVILEGED  SVC OPCODE
          BYTE   0           RESERVED FOR ERROR CODE.
SVCFLG    DATA   0           READ TSB(TASK STATUS BLOCK)
RUNID     BYTE   0           RUN-TIME ID OF THE TASK WHOSE TSB IS TO BE READ
OFFSET    BYTE   >32         BYTE OFFSET IN THE TSB TO BE READ
VALUE     DATA   0           VALUE OF THE WORD READ FROM THE TSB IS RETURNED
          DATA   0
TEMP      DATA   0           SIX WORD SPACES ARE RESERVED
          DATA   0           FOR STORING THE WORDS READ
          DATA   0           FROM THE TSB
          DATA   0
          DATA   0
          DATA   0

*************************************************

START     XOP    @SCBJ,15        INITIATES SELF-IDENTIFICATION SVC
          MOVB   @RTID,@RUNID
          LI     R3,TEMP
          LI     R2,>3200        MSB CONTAINS OFFSET OF THE WORD INTO TSB
*                                TO BE READ
REPEAT    XOP    @RWTTSB,15      INITIATE READING OF THE WORD FROM THE TSB
          MOV    @VALUE,*R3+     STORE WORD READ FROM TSB
          AI     R2,>200         TO POINT TO THE NEXT WORD IN THE TSB
          MOVB   R2,@OFFSET      GET THE NEXT OFFSET VALUE
          CI     R2,>3E00        OFFSET FOR THE LAST WORD TO BE READ.
```

```
        JNE   REPEAT          CONTINUE UNTIL ALL THE SIX WORDS HAVE
        LI    R1,TEMP         BEEN READ.STORE SIX WORDS READ FROM TSB.
        LMF   R1,0            LOAD MEMORY MAP FILE 0 WITH 6 WORDS OF
*                             MEMORY, STARTING AT THE ADDRESS SPECIFIED
*                             IN REGISTER R1.

*****************************************************************
*   CHECKING THE STATE OF THE BOARD BEFORE ISSUING A COMMAND    *
*****************************************************************

        LI    R12,>F900       POINT TO THE FCCC
        CLR   R0
LOOP    SZC   @IEWRD,@WOSW0(R12)  DISABLE FCCC INTERRUPTS TO THE HOST
        MOV   *R12,R10        RECOVER ROSW0 CONTENT
DEL1    COC   @OBFWRD,R10     WILL THE BOARD TAKE A COMMAND ?
        JEQ   DEL1            NO, THE BOARD IS BUSY.WAIT UNTIL
A1      CI    R0,1            THE FLAG IS CLEARED.
        JEQ   A2
        JMP   A3
A2      B     *R11            RETURN ADDRESS FOR THE SUBROUTINE

*****************************************************************
*  THE BOARD IS READY AT THIS POINT FOR WRITING INTO THE      *
*  SLAVE WORDS(WOSW0 THROUGH WOSW3).INITIATE THE OPEN         *
*  CHANNEL COMMAND TO THE FCCC.                               *
*****************************************************************

A3      LI    R12,>F900
        LI    R5,>0F05        TIME-OUT= 250*15 mSEC.,SELECTS UPA
*                             PROTOCOL
        LI    R6,>2100        OPEN CHANNEL 0;
        MOV   R5,@WOSW1(R12)
        MOV   R6,@WOSW3(R12)
        LI    R10,IEWRD       SET UP TO ENABLE INTERRUPT
        SOC   R10,@WOSW0(R12) ENABLE INTERRUPTS;ALLOWS HOST TO BE
*                             INTERRUPTED BY THE FCCC.

*****************************************************************
*      TEST IF THE COMMAND HAS BEEN EXECUTED                    *
*****************************************************************

        BL    @CHECK
        XOP   @ERR1,15        CREATE A SEQUENTIAL FILE
        B     @STOP           IF ERROR, BRANCH TO END OF PROGRAM
        XOP   @CORR1,15       ELSE CREATE ANOTHER SEQUENTIAL FILE
```

```
*****************************************************************************
*       OPEN CHANNEL COMMAND HAS PERFORMED CORRECTLY.                       *
*       FCCC KNOWS WHERE ITS CHANNEL PARAMETERS TABLE IS                    *
*       LOCATED FOR A KNOWN PROTOCOL.NOW,INITIATE A WRITE                   *
*       CHANNEL PARAMETERS COMMAND TO MODIFY THE PARAMETERS                 *
*       ON THE TABLE, OTHERWISE,DEFAULT VALUES FOR THE GIVEN                *
*       PROTOCOL WILL BE TAKEN.                                             *
*****************************************************************************

          LI      R12,>F900
          LI      R0,1
          BL      @LOOP
          CLR     R0
          LI      R5,>0500        HIGH BYTE SELECTS UPA PROTOCOL
          LI      R6,BUFAD        LSW OF HOST DATA BUFFER ADDRESS
          LI      R7,>7300        OPCODE FOR WRITE CHANNEL PARAMETERS
*                                 COMMAND FOR CHANNEL 0
          MOV     R5,@WOSW1(R12)
          MOV     R6,@WOSW2(R12)
          MOV     R7,@WOSW3(R12)
          LI      R0,IEWRD        SET UP TO ENABLE INTERRUPTS AGAIN
          SOC     R10,@ROSW0(R12)    ENABLE INTERRUPTS

*****************************************************************
*       TEST TO EXAMINE IF THE COMMAND HAS BEEN EXECUTED        *
*       CORRECTLY                                               *
*****************************************************************

          BL      @CHECK
          XOP     @ERR2,15        IF ERROR,CREATE A SEQUENTIAL FILE AND
          B       @STOP           BRANCH TO THE END OF THE PROGRAM
          XOP     @CORR2,15       OTHERWISE, CREATE ANOTHER SEQUENTIAL FILE

*****************************************************************
*       WRITE PARAMETERS COMMAND HAS BEEN EXECUTED CORRECTLY    *
*       NOW ISSUE THE WRITE/READ COMMAND.                       *
*****************************************************************

READ      LI      R12,>F900
          LI      R0,1
          BL      @LOOP           CHECK TO SEE THE STATE OF THE BOARD
*                                 BEFORE ISSUING A COMMAND
          CLR     R0
          LI      R5,>0F00        TIME-OUT=15*250 mSEC.;MSB ADDRESS OF
*                                 DATA BUFFER IS ZERO.
          LI      R6,RCVDAT       LSW(LEAST SIGNIFICANT WORD) ADDRESS OF
*                                 HOST DATA BUFFER TO RECEIVE INCOMING DATA
          LI      R7,>0400        OPCODE FOR READ COMMAND;BRANCHING TO
*                                 INITIAL CHARACTER DETECTION(CD) RECEIVE
```

```
*                                   STATE WITH MODIFIER 0 FOR CHANNEL 0
          MOV     R5,@WOSW1(R12)
          MOV     R6,@WOSW2(R12)
          MOV     R7,@WOSW3(R12)
          LI      R10,IEWRD
          SOC     R10,@WOSW0(R12)     ENABLE FCCC INTERRUPTS TO THE HOST

*****************************************************************************
*       CHARACTER DETECTION ROUTINE AND THE INTERRUPT SERV- *
*       ICE ROUTINE WILL BE HANDLED BY THE FCCC FIRMWARE.   *
*       COMPLETION STATUS IS REPORTED IN ROSW0 ( BITS 8     *
*       THROUGH 15).COMPLETION STATUS IN ROSWO INDICATES AN *
*       ERROR ONLY WHEN THE ERROR FLAG IS SET (ROSW3, BIT 0)*
*****************************************************************************

TEST      MOV     @ROSW3(R12),R6
          COC     @ERWRD,R6           HAS ANY ERROR OCCURED?
          JEQ     A4                  YES, JUMP TO CREATE AN ERROR FILE
          XOP     @CORR3,15           CREATE SEQUENTIAL FILE
          B       @STOP
A4        XOP     @ERR3,15            CREATE SEQUENTIAL FILE
          B       @STOP

*****************************************************************************
*       SUBROUTINE CHECK STARTS HERE                                *
*****************************************************************************

CHECK     MOV     @ROSW3(R12),R6      READ ROSW3 INFORMATION AFTER COMMAND
*                                     EXECUTION
          COC     @ERWRD,R6           HAS ANY ERROR OCCURED?
          JEQ     A6                  YES, JUMP TO A6
          AI      R11,4
          B       *R11                RETURN FOR CORRECT COMMAND EXECUTION
A6        B       *R11                ERROR RETURN
STOP      END
```

APPENDIX C

```
..............................................................
..   THIS IS A PROGRAM WRITTEN IN THE AMPL LANGUAGE OF TI-990/12
..   MINICOMPUTER TO READ DATA FROM THE ASYNCHRONOUS SERIAL LINE.
..   DEC-10 MAINFRAME WRITES THE CHARACTERS ON THE LINE;TI-990
..   READS THE ASCII DEIMAL VALUES OF THE CHARACTERS SENT BY THE
..   DEC-10 MAINFRAME.  THE SOURCE FILENAME IS "AHS.SOURCE.RECEIVE".
..............................................................


PROC INIT(1,1)
     EINT('EM01',0)
     ..ASSUME EITHER USING TM990/101 OR TM990/U89
     ..MUST BE USING TARGET CLOCK (ECLK MUST BE ZERO)
     ..ASSUME 3 MHZ CLOCK IF TMS9900 (ETYP =1)
     ..ASSUME 2 MHZ CLOCK IF TMS9980 (ETYP=2)
     ..ASSUME 7 BITS/CHARACTER,2 STOP BITS,EVEN PARITY
     ..CLOCK 4M=0; INTERNAL FREQ IS PHASE 3 DIVIDED BY 3
     IF ECLK NE 0 THEN 'WARNING:TARGET CLOCK MUST BE USED';NL
     IF ETYP EQ 1 THEN BEGIN CLOCK =30000    ..3 MHZ FREQ/100
          CRUB=080
          END
     ELSE
          IF ETYP EQ 2 THEN BEGIN CLOCK =20000    ..2 MHZ FREQ/100
          CRUB=0800
          END
          ELSE
               'WARNING:MUST USE EITHER 9900 OR 9980 BUFFER';NL
     CRUW(31,1,1)         .."SBO 31"     9902 RESET
     CRUW(0,8,>A2)        .."LDCR @CNTRL,8"   LOAD CONTROL REGISTER
     CRUW(13,1,0)         .."SBZ 13"     SKIP INTERVAL REGISTER
     BAUD=ARG 1/100
     IF CLOCK /(3*2*BAUD) GE >0400 THEN PRE=8
     ELSE PRE=1
     DRVAL =CLOCK /(3*2*PRE*BAUD)
     IF PRE EQ 8 THEN DRVAL= >0400 OR DRVAL
     CRUW(0,12,DRVAL)          .."LDCR @BAUD"      LOAD XMIT/RECEIVE RATE
     END
FORM INIT
          ':INITIALIZE TMS9902';
          ':NOTE: IT IS ASSUMED THAT THE EMULATOR';
          ':      HAS BEEN INITIALIZED USING THE TARGET';
          ':      SYSTEM CLOCK;E.G. EINT("EM01",0)';
          'BAUD RATE' ='300';
     END

PROC WRITCH(1)
     IF ETYP EQ 1 THEN CRUB=080
     ELSE CRUB =0800
```

```
      CRUW(16,1,1)                ..SET RTSON
      WHILE ((CRUR(22,1) AND 1) EQ 0) DO NULL    ..WAIT TILL XBRE=1
      CRUW(0,8,ARG 1)
      END

PROC MULTIRD(0,1)
      INIT
      ARRAY STORE(4800)
      IF (ETYP EQ 1) THEN CRUB=>080
      ELSE CRUB =>0800
      WRITCH(13)
      FOR I=1 TO 4800  DO BEGIN
          WHILE ((CRUR(21,1) AND 1) EQ 0) DO NULL
          LOC 1=CRUR(0,8)
          STORE(I)= LOC 1
          IF (I EQ 4500) THEN BEGIN
             'STOP SENDING FROM DEC-10 '
             'STOP SENDING FROM DEC-10 '
             'STOP SENDING FROM DEC-10 '
             'STOP SENDING FROM DEC-10 '
              END
          IF (STORE(I) EQ 0) THEN ESCAPE
          CRUW(18,1,0)
        END
END
PROC CRFILE          ..CREATES A FILE FOR THE DATA RECEIVED
      BEGIN
        LIST('AHS.AMPL.STORE')
        FOR J=1 TO I DO STORE(J):D
        LIST(EOF)
      END
END

PROC RECEIVE
      BEGIN
          MULTIRD
          CRFILE
      END
END
```

****

```
PROGRAM RECOVER;
(*-----------------------------------------------------------------
THIS IS THE PROGRAM USED TO RESTORE THE ORIGINAL FILE SENT BY DEC-10
TO THE TI-990/12 MINICOMPUTER.  THE FILE IS RECONSTRUCTED FROM
THE INFORMATION AVAILABLE IN INPUT FILE "AHS.AMPL.STORE". THE PROGRAM
READS EQUIVALENT INTEGER VALUES OF ASCII CHARACTERS (BOTH PRINTABLE
AND NON-PRINTABLE OR CONTROL CHARACTERS) FROM AN INPUT FILE AND
CONVERTS THEM INTO THE CORRESPONDING ASCII CHARACTERS.  ALL THESE
CHARACTERS ARE THEN SAVED INTO A TEXT FILE NAMED "OUTFILE".
-----------------------------------------------------------------*)

TYPE FILEA = FILE OF INTEGER;
     FILEB = TEXT;
VAR I,J :INTEGER;
    NUM:INTEGER;
    RDATA :CHAR;
    INFILE : FILEA;
    OUTFILE: FILEB;
BEGIN REWRITE(OUTFILE);  (* START FROM THE BEGINNING OF THE FILE *)
      RESET(INPUT);      (* RESET INPUT FILE FOR READING *)
      READ(NUM);         (* READ AN INTEGER VALUE FROM INPUT FILE *)
      WHILE NOT EOF(INPUT) DO      (* CONTINUE UNTIL END-OF-FILE *)
        BEGIN                              .
            READ(NUM);
            IF (NUM >= ORD('a')) AND (NUM <= ORD('z'))THEN NUM:=NUM -32
            ELSE NUM :=NUM;    (* CONVERT ASCII VALUES OF LOWER CASE *)
              (* LETTERS TO THE CORRESPONDING VALUES OF THE UPPER CASE*)
            RDATA := CHR(NUM);    (*CONVERT INTO ASCII CHARACTER *)
          IF (NUM = 13) THEN WRITELN(OUTFILE)
          ELSE IF (NUM <127) AND (NUM >= 32) THEN WRITE(OUTFILE,RDATA);
                 (* WRITE ONLY THE PRINTABLE CHARACTERS IN THE FILE *)
        END;
        CLOSE(OUTFILE)
END.
```

```
(*-------------------------------------------------------------
THIS IS A PROGRAM RUN ON DEC-10 WHILE RECEIVING CHARACTERS
FROM THE DEC-10 MAINFRAME TO THE TI-990/12 MINICOMPUTER.  THE
PROGRAM IS SAVED UNDER THE FILENAME "READ.PAS". IF THE FILE TO
BE SENT TO THE TI-990 HAS MORE THAN 3000 CHARACTERS,  MULTIPLE
VALUES OF "COUNT" IS NECESSARY TO TRANSFER THE WHOLE FILE.  WITH
COUNT=1, THE PROGRAM DOES A DUMY READING OF THE  FIRST 3000
CHARACTERS FROM THE INPUT FILE WITHOUT WRITING THEM ON THE LINE.
FOR THE NEXT SET OF 3000 CHARACTERS, EACH CHARACTER READ FROM
THE INPUT FILE IS WRITTEN ON THE SERIAL LINE.  WITH COUNT =2,
THE NEXT SET OF 3000 CHARACTERS OF THE INPUT FILE ARE ADDED TO
THE DUMY READING.  THE PROCESS IS CONTINUED UNTIL END-OF-FILE
CONDITION IS TRUE.
-------------------------------------------------------------*)

PROGRAM READ(INPUT,OUTPUT);
 VAR
        CHARIN:CHAR;
        I,J :INTEGER;
        COUNT,NUM:INTEGER;
 BEGIN                      (* MAIN *)
        J :=0;
        I:=0;
        COUNT:=0;
        NUM :=0;
        IF (COUNT >0)
          THEN
             BEGIN        (* DUMY READING *)
                WHILE (COUNT <> 0 ) DO
                   BEGIN
                     FOR NUM := 1 TO 3000 DO
                        READ(CHARIN);
                        COUNT :=COUNT-1;
                   END;
            END;          (* DUMY READING *)
         BEGIN            (* START WHEN COUNT =0 *)
           WHILE  (NOT EOF) AND (I <= 3000) DO
           BEGIN
                 I :=I+1;
                READ(CHARIN);       (* READ A CHARACTER FROM THE LINE *)
                IF EOLN(INPUT)      (* IF END-OF-LINE  *)
                   THEN             (* THEN      *)
                        BEGIN
                              WRITELN(CHARIN);    (* WRITE THE CHAR. *)
                                                  (* GO TO NEXT LINE *)
                              WHILE (J <100000) DO J:= J+1;  (* DELAY *)
```

```
                        J :=0;
                    END
                ELSE                (* OTHERWISE *)
                    BEGIN
                        WRITE(CHARIN);    (* WRITE THE CHAR. *)
                        WHILE (J <15000) DO J:=J+1;
                        (* DELAY *)
                        J :=0;
                    END;
            END
        END;
END.            (* MAIN *)
```

MODIFIED RECEPTION TECHNIQUE

```
.....................................................................
..  THIS IS A PROGRAM WRITTEN IN  AMPL LANGUAGE OF THE TI-990
..  MINICOMPUTER TO READ ASCII CHARACTERS FROM THE SERIAL LINE.
..  THE PROGRAM IS RUN INTERACTIVELY WITH ANOTHER PROGRAM ON THE
..  DEC-10.  ALL THE CHARACTERS READ BY TI-990 ARE SAVED IN A FILE
..  NAMED "AHS.AMPL.STORE".  THE FILE CONTAINS ASCII VALUES OF
..  CHARACTERS READ.
.....................................................................


PROC INIT(1,1)
     EINT('EM01',0)
     ..ASSUME EITHER USING TM990/101 OR TM990/U89
     ..eST BE USING TARGET CLOCK (ECLK MUST BE ZERO)
     ..ASSUME 3 MHZ CLOCK IF TMS9900 (ETYP =1)
     ..ASSUME 2 MHZ CLOCK IF TMS9980 (ETYP=2)
     ..ASSUME 7 BITS/CHARACTER,2 STOP BITS,EVEN PARITY
     ..CLOCK 4M=0; INTERNAL FREQ IS PHASE 3 DIVIDED BY 3
     IF ECLK NE 0 THEN 'WARNING:TARGET CLOCK MUST BE USED';NL
     IF ETYP EQ 1 THEN BEGIN CLOCK =30000     ..3 MHZ FREQ/100
          CRUB=080
          END
     ELSE
          IF ETYP EQ 2 THEN BEGIN CLOCK =20000     ..2 MHZ FREQ/100
          CRUB=0800
          END
          ELSE
               'WARNING:MUST USE EITHER 9900 OR 9980 BUFFER';NL
     CRUW(31,1,1)          .."SBO 31"      9902 RESET
     CRUW(0,8,>A2)         .."LDCR @CNTRL,8"   LOAD CONTROL REGISTER
     CRUW(13,1,0)          .."SBZ 13"      SKIP INTERVAL REGISTER
     BAUD=ARG 1/100
     IF CLOCK /(3*2*BAUD) GE >0400 THEN PRE=8
     ELSE PRE=1
     DRVAL =CLOCK /(3*2*PRE*BAUD)
     IF PRE EQ 8 THEN DRVAL= >0400 OR DRVAL
     CRUW(0,12,DRVAL)          .."LDCR @BAUD"       LOAD XMIT/RECEIVE RATE
     END
FORM INIT
          ':INITIALIZE TMS9902';
          ':NOTE: IT IS ASSUMED THAT THE EMULATOR';
          ':      HAS BEEN INITIALIZED USING THE TARGET';
          ':      SYSTEM CLOCK;E.G. EINT("EM01",0)';
          'BAUD RATE' ='300';
```

```
     END

.....................................................................
..   PROC STORAGE DEFINES A STORAGE AREA FOR SAVING CHARACTERS
..   TEMPORARILY.
.....................................................................

PROC STORAGE
     BEGIN
          ARRAY STORE(200)
     END
END

.....................................................................
..   PROC WRITCH WRITES A SINGLE CHARACTER ON THE SERIAL LINE
.....................................................................

PROC WRITCH(1)
     IF ETYP EQ 1 THEN CRUB=080
     ELSE CRUB =0800
     CRUW(16,1,1)              ..SET RTSON
     WHILE ((CRUR(22,1) AND 1) EQ 0) DO NULL    ..WAIT TILL XBRE=1
     CRUW(0,8,ARG 1)
     END


.....................................................................
..  FUNC CHECKSUM CALCULATES THE CHECKSUM OF THE CHARACTERS READ.
..  THIS IS THE SAME CHECKSUM USED BY THE KERMIT PROGRAM.
.....................................................................

FUNC CHECKSUM(1,2)
     BEGIN
          X = (ARG 1 MOD 256)
          LOC 1=X
          X=DIV(64,LOC 1)
          X = X+ ARG 1
          LOC 2 = (X MOD 64)
          LOC 2= LOC 2+ 32
     END
     RETURN LOC 2
END
```

```
.....................................................
.. PROCEDURE ONERD READS JUST ONE CHARACTER FROM THE SERIAL
.. COMMUNICATION LINE.
.....................................................

PROC ONERD(0,1)
     IF (ETYP EQ 1) THEN CRUB =>080
     ELSE CRUB =>0800     ..CRU BASE ADDRESS FOR TMS9902
BEGIN
     WHILE ((CRUR(21,1) AND 1) EQ 0) DO NULL    ..WAIT TIL RBRL=1
     LOC 1 =CRUR(0,8)     ..READ 8 BITS CORRESPONDING TO THE ASCII
                          ..VALUE OF THE CHARACTER AND SAVE IT
                          ..TEMPORARILY IN A LOCATION.
     T =LOC 1     ..T CONTAINS ASCII VALUE OF CHARACTER JUST READ
     CRUW(18,1,0)         ..ENABLE NEXT CHARACTER
     END
END


.....................................................
..  PROC MULTIRD READS ALL THE CHARACTERS IN A DATA PACKET
..  SENT BY THE DEC-10.
.....................................................

PROC MULTIRD(0,1)
     BEGIN
         I=1
         ONERD            ..READ ONE CHARACTER
         STORE(I)=T       ..SAVE THE CHARACTER TEMPORARILY
         WHILE (STORE(I) NE 3) DO BEGIN      ..WHILE NOT END-OF-PACKET
            ONERD              ..READ A SINGLE CHARACTER
            I=I+1
            STORE(I)= T        .. SAVE THE CHARACTER TEMPORARILY
            IF (STORE(I) EQ 0) THEN ENDFILE=0    .. IF END-OF-FILE, THEN
            ELSE ENDFILE =1                      ..ENDFILE=0 ELSE
ENDFILE=1
            IF (ENDFILE EQ 0) THEN ESCAPE        .. IF END-OF-FILE,ESCAPE
         END                                     ..THE LOOP
         CHKSUM =0
         FOR K=2 TO (I-2) DO BEGIN               ..
            NUM =STORE(K)
            IF (NUM LT 127) AND (NUM GE 32) THEN NUM =NUM
            ELSE NUM=0
            CHKSUM =CHKSUM +NUM     ..ASCII SUM OF ALL THE CHARS. READ
            P=CHECKSUM(CHKSUM)      ..ACTUAL CHECSUM OF THE CHARS. READ
         END
     END
END
```

```
.....................................................................
..  PROC RECEIVE PROCESSES READING CHARACTERS FROM THE SERIAL LINE
..  AND SAVING THEM IN THE FILE "AHS.AMPL.STORE",  IF ALL THE
..  CHARACTERS IN THE PACKET ARE READ CORRECTLY
.....................................................................

PROC RECEIVE(0,2)
     BEGIN
       LOC 1 =1
       I= 1
       NUM=0
       WHILE (LOC 1 EQ 1) DO BEGIN    ..CONTINUE READING UNTIL
                                      ..END-OF-FILE
            MULTIRD              ..READ A PACKET
            STORE(150) =0
            IF (P EQ STORE(K+1)) THEN BEGIN        ..IF THE PACKET IS
                                                   ..READ
                 FOR J= 2 TO (I-2) DO STORE(J):D   .. CORRECTLY THEN
                 LOC 2=1             ..SAVE THE CHARS. IN THE PACKET
                 WRITCH(13)          ..INITIATE THE
                 WAIT(20)            .. DEC-10 TO SEND THE
                 WRITCH(13)          ..NEXT DATA PACKET
            END
            ELSE REPEAT BEGIN        ..ELSE INITIATE THE DEC-10
                      LOC 2=0       TO SEND THE SAME PACKET AGAIN
                      WAIT(10)
                      IF (CHKSUM GE STORE(150)) THEN STORE(150) = CHKSUM
                      WRITCH("")
                      WRITCH(13)
                      MULTIRD      ..READ THE SAME PACKET AGAIN SENT BY
                                   ..DEC
                      IF (STORE(150) EQ CHKSUM) THEN P =STORE(K+1)
            END UNTIL (P EQ STORE(K+1))      .. CONTINUE UNTIL CORRECT
                                             .. CHECKSUM
                        .. IS OBTAINED FROM THE CHARACTERS READ.
            IF (LOC 2 EQ 0) THEN BEGIN
                 FOR L=2 TO (I-2) DO STORE(L):D  ..SAVE THE CHARS. IF
                 WRITCH(13)          ..THE FIRST PACKET SENT BY DEC IS NOT
                 WAIT(10)            .. RECEIVED CORRECTLY
                 WRITCH(13)
            END
            IF (ENDFILE EQ 0) THEN LOC 1=0    IF END-OF-FILE, THEN STOP.
       END
     END
END
```

```
.....................................................................
..   PROC NEWRECV INVOKES ALL THE PROCEDURES AND FUNCTIONS NECESSARY
..   TO INITIATE THE RECEPTION PROCEDURE.
.....................................................................

PROC NEWRECV
     BEGIN
       INIT
       STORAGE
       LIST('AHS.AMPL.STORE')
       WRITCH(13)
       RECEIVE
     END
END


PROC LAST
     BEGIN
       FOR L=2 TO I DO STORE(L):D
       LIST(EOF)
     END
END


(*-----------------------------------------------------------
THIS IS A PROGRAM ON DEC-10.  THE SOURCE FILENAME IS RECEIVE.PAS.
THE PROGRAM IS RUN INTERACTIVELY WITH ANOTHER PROGRAM ON THE
TI-990/12 MINICOMPUTER.  A DATA PACKET IS FORMED OUT OF EVERY
CHARACTER ON A LINE OF THE INPUT FILE.  AN END-OF-LINE CHARACTER,
A CHECKSUM CHARACTER AND AN END-OF-PACKET CHARACTER FOLLOW THE
ACTUAL DATA CHARACTERS IN EACH PACKET.  IF THE TI-MINICOMPUTER
HAS NOT RECEIVED A PACKET CORRECTLY, THE SAME PACKET IS SENT BY
THE DEC-10 AGAIN.  THE PROCESS CONTINUES UNTIL TI-990 HAS RECEI-
VED THE PACKET CORRECTLY.  THEN THE NEXT PACKET IS SENT BY
DEC-10; THE PROCESS IS REPEATED UNTIL THE ENTIRE FILE HAS BEEN
TRANSFERRED.  "INFILE" IS THE NAME OF THE FILE TO BE TRANSFERRED.
-----------------------------------------------------------*)

PROGRAM READ(INPUT,OUTPUT,INFILE);
 LABEL 10;
 TYPE  A= ARRAY[1..100] OF CHAR;
 VAR
        TEMP:A;
        INFILE : TEXT;
        CHARIN:CHAR;
        I,J,K,M :INTEGER;
        COUNT,NUM: INTEGER;
        SYNC : INTEGER;
        CHECKSUM : INTEGER;
```

```
        X,C,T : INTEGER;
        ERRORFLAG : CHAR;
BEGIN                   (* MAIN *)
        CHECKSUM :=0;
        J :=0;
        I:=0;
        K:= 0;
        COUNT ;=0;
        NUM :=0;
        WRITELN('VALUE OF COUNT = 'COUNT);
        WRITELN(' WRITE NEW COUNT VALUE  ');
        READLN;READ(SYNC);
        COUNT := SYNC;
        RESET(INFILE);
        IF (COUNT >0)
           THEN
             BEGIN        (* DUMY READING *)
                WHILE COUNT <> 0 ) DO
                   BEGIN
                        FOR NUM:= 1 TO 5000 DO
                        READ(INFILE,CHARIN);
                        COUNT := COUNT -1;
                   END;  (* DUMY READING *)
            END;
        BEGIN
           WHILE NOT EOF(INFILE) AND (I <=5000) DO
              BEGIN
                K :=0;
                READ(INFILE,CHARIN);  (* READ A CHARACTER FROM INFILE *)
                I:= I+1;
                K := K+1;
                TEMP[K] := CHARIN;   (*SAVE THE CHARACTER IN A BUFFER*)
                WHILE NOT EOLN(INFILE) DO   (* WHILE NOT END-OF-LINE *)
                BEGIN
                     READ(INFILE,CHARIN);   (* READ A CHARACTER FROM *)
                     I := I+1;               (* THE INPUT FILE *)
                     K:= K+1;
                     TEMP[K] := CHARIN; (*SAVE THE CHAR. TEMPORARILY*)
                END;
                K := K+1;
                TEMP[K] := CHR(2);      (* END-OF-LINE CHARACTER *)
                CHECKSUM := 0;
                FOR M:= 1 TO K DO
                BEGIN
                        IF (TEMP[M] <=CHR(31))
                           THEN
                                CHECKSUM :=CHECKSUM
```

```
                    ELSE
                         CHECKSUM := CHECKSUM + ORD(TEMP[M]);
                         (* SUM OF CHARAACTERS *)
               END;
               BEGIN
                    C:= CHECKSUM;
                    X:=(C MOD 256) DIV 64;
                    X:= X+ C;
                    CHECKSUM := X MOD 64;
                    CHECKSUM := CHECKSUM + 32;
                    TEMP[K+1] := CHR(CHECKSUM);
                    (* CHECKSUM CHARACTER *)
                    TEMP[K+2] := CHR(03);
                    (* END-OF-PACKET CHARACTER *)
               END;
10:            WHILE (J < 125000) DO J:=J+1;
               J:=0;
               FOR M := 1 TO (K+2) DO
               BEGIN    (* WRITE THE PACKET ON THE LINE *)
                    WRITE(TEMP[M]);
                    WHILE (J < 15000) DO J := J+1;        (* DELAY *)
                    J := 0;
               END;
               READLN;READ(ERRORFLAG);
                      (* READ ERRORFLAG SENT BY TI *)
               IF (ERRORFLAG = '' )      (* IF TRUE,  THEN SEND THE *)
                  THEN
                    GOTO 10;             (* SAME PACKET AGAIN *)
             END
          END;
          TEMP[K+1] :=CHR(0);          (* END-OF-FILE CHARACTER *)
          WRITE(TEMP[K+1]);            (* WRITE EOF CHARACTER *)
END.
```

APPENDIX D

```
.....................................................................
..   THIS IS A PROGRAM WRITTEN IN THE AMPL LANGUAGE OF THE TI-990
..   MINICOMPUTER TO TRANSMIT (WRITE) ASCII CHARACTERS TO THE
..   DEC-10 MAINFRAME.
.....................................................................

PROC INIT(1,1)
     EINT('EM01',0)       ..INITALIZE EMULATOR
     ..ASSUME EITHER USING TM990/101 OR TM990/U89
     ..MUST BE USING TARGET CLOCK (ECLK MUST BE ZERO)
     ..ASSUME 3 MHZ CLOCK IF TMS9900 (ETYP =1)
     ..ASSUME 2 MHZ CLOCK IF TMS9980 (ETYP=2)
     ..ASSUME 7 BITS/CHARACTER,2 STOP BITS,EVEN PARITY
     ..CLOCK 4M=0; INTERNAL FREQ IS PHASE 3 DIVIDED BY 3
     IF ECLK NE 0 THEN 'WARNING:TARGET CLOCK MUST BE USED';NL
     IF ETYP EQ 1 THEN BEGIN CLOCK =30000     ..3 MHZ FREQ/100
          CRUB=080
          END
     ELSE
          IF ETYP EQ 2 THEN BEGIN CLOCK =20000     ..2 MHZ FREQ/100
          CRUB=0800
          END
          ELSE
               'WARNING:MUST USE EITHER 9900 OR 9980 BUFFER';NL
     CRUW(31,1,1)          .."SBO 31"      9902 RESET
     CRUW(0,8,>A2)         .."LDCR @CNTRL,8"   LOAD CONTROL REGISTER
     CRUW(13,1,0)          .."SBZ 13"      SKIP INTERVAL REGISTER
     BAUD=ARG 1/100
     IF CLOCK /(3*2*BAUD) GE >0400 THEN PRE=8
     ELSE PRE=1
     DRVAL =CLOCK /(3*2*PRE*BAUD)
     IF PRE EQ 8 THEN DRVAL= >0400 OR DRVAL
     CRUW(0,12,DRVAL)          .."LDCR @BAUD"      LOAD XMIT/RECEIVE RATE
     END
FORM INIT
          ':INITIALIZE TMS9902';
          ':NOTE: IT IS ASSUMED THAT THE EMULATOR';
          ':      HAS BEEN INITIALIZED USING THE TARGET';
          ':      SYSTEM CLOCK;E.G. EINT("EM01",0)';
          'BAUD RATE' ='300';
     END



PROC WRITCH(1)
     IF ETYP EQ 1 THEN CRUB =080
```

```
     ELSE CRUB = 0800
     CRUW(16,1,1)                    ..SET RTSON
     WHILE ((CRUR(22,1) AND 1) EQ 0 ) DO NULL      ..WAIT TIL XBRE =1
     CRUW(0,8,ARG 1)
     END
FORM WRITCH
     ':WRITE CHARCTER TO 9902';
     ':CHARACTER TO TRANSMIT'= '"A"';
END
..
.....................................................................
..   THIS PROCEDURE CALLS "WRITCH" REPEATEDLY TO SEND THE SAME CHARACTER
.....................................................................
..
PROC SAME(1)
     IF ETYP EQ 1 THEN CRUB =080
     ELSE CRUB = 0800
     WHILE 1 DO WRITCH(ARG 1)
     END
FORM SAME
     ':SAME - TRANSMIT CHARACTER REPEATEDLY';
     'CHARACTER TO TRANSMIT'='"A"';
END

PROC WRITE(1)
   FOR I=>100 TO (ARG 1 +256) BY 2 DO BEGIN
     VALMSB = @I          ..CONTAINS ASCII VALUES IN BOTH THE BYTES
     VALLSB = VALMSB                          ..LSB
     VALMSB = VALMSB AND >FF00      ..MSB CONTAINS CHARACTER VALUE
     VALLSB  =VALLSB AND >00FF      ..LSB CONTAINS CHARACTER VALUE
     VALMSB =VALMSB >>8       ..CHARACTER VALUE BROUGHT TO THE LSB
     WRITCH(VALMSB)
     IF (VALMSB EQ 13) THEN WAIT(12)
     WRITCH(VALLSB)
     IF (VALLSB  EQ 13) THEN WAIT(12)
   END
END

FORM WRITE
     ':WRITE THE VALUE OF COUNT TO BE PASSED  ';
     ':AS AN ARGUMENT.IF COUNT IS ODD THEN MAKE IT EVEN ';
     ': BY ADDING 1 TO IT.VALUE OF COUNT CAN BE OBTAINED';
     ': WHILE EXECUTING THE PASCAL TASK [XPT] .COUNT    ';
     ': REPRESENTS TOTAL NUMBER OF CHARACTERS TO BE TRANSMITTED';
     ': DO NOT FORGET TO LOAD THE OBJECT CODE OF "OUTFILE" (SOURCE)';
     ': STARTING AT ADDRESS  >100';
     'COUNT VALUE '=;
END
```

```
PROC TRANSMIT
     INIT
     WRITE
     END
```

```
**********************************************************************************
```

```
PROGRAM HEXCONV;
(*-------------------------------------------------------------------
THIS IS THE PROGRAM USED TO PRE-PROCESS THE CONTENTS OF A FILE,  BEFORE
THE FILE CAN BE TRANSMITTED TO THE DEC-10.  SO,  THE INPUT FILE PATHNAME
IS THE FILE TO BE SENT TO THE DEC-10 MAINFRAME.  BASICALLY,  THE PROGRAM
CONVERTS ALL THE CHARACTERS IN A  TEXT FILE INTO ITS HEX EQUIVALENT
VALUES AND SAVES THOSE VALUES IN A FILE CALLED "OUTFILE".  SO,  THE
CONTENTS OF "OUTFILE" ARE PURE DATAS CORRESPONDING TO THE INPUT FILE.
-------------------------------------------------------------------*)
TYPE FILEB = TEXT;
VAR
       I,NUM:INTEGER;
       COUNT :INTEGER;
       RDATA : CHAR;
       OUTFILE :FILEB;
BEGIN                        (* MAIN *)
       REWRITE(OUTFILE);
       RESET(INPUT);
       COUNT := 0;
       WHILE NOT EOF DO
       BEGIN
         WHILE NOT EOLN(INPUT) DO
         BEGIN
            WRITE(OUTFILE,'         BYTE  ');
            READ(RDATA);
            COUNT := COUNT +1;
            NUM := ORD(RDATA);    (*STORE ASCII VALUE OF RDATA IN "NUM"*)
            WRITE(OUTFILE,NUM);
            WRITELN(OUTFILE);
                                  (* WRITE ASCII EQUIVALENT INTO OUTFILE*)
         END;
         WRITELN(OUTFILE,'         BYTE  13 ');
                                (* WRITE END-OF-LINE INTO OUTFILE   *)
         READ(RDATA);
         COUNT := COUNT +1;  (* UPDATE NUMBER OF CHARS. READ *)
       END;
       CLOSE(OUTFILE);          (* CLOSE THE FILE OUTFILE *)
       WRITE(' COUNT = ',COUNT);    (*WRITE TOTAL NO. OF CHARS. READ *)
END.             (* MAIN *)
```

APPENDIX E

The six fields in a packet of information in the Kermit protocol:-

MARK — Start-of-packet character, normally SOH (control-a)

LEN — The number of ASCII characters, including prefixing characters and the checksum, in the rest of the packet that follows this field: in other words, the packet length minus two. Since this number is expressed as a single character via the char function , packet character counts of 0-94 are permitted, and 96 is the maximum total packet length , including the MARK and LEN fields.

SEQ — The packet sequence number, between 0 and 63. The sequence number wraps around to zero after each group of 64 packets.

TYPE — The packet type, a single printable ASCII character, is one of the following:-

D Data .

Y Acknowledge (ACK)

N Negative acknowledge(NAK)

S Send Initiate (SEND-INIT)

R Receive initiate

B Break transmission (EOT)

F File header

Z End-of-file (EOF)

E Error

G Generic command. A single character in the data fields possibly followed by operands, requests host -independent remote execution of the specified command:

L Log out, bye

F Finish, but don't log out

D Directory querry (followed by optional file specifications)

U Disk-usage query

E erase (followed by file specification)

T Type (followed by file specation)

Q Query server status

and others.

C Host command;the data field contains a string to be executed as a system-dependent (literal) command by the host.

X Text display header. To indicate the arrival of text

to be displayed on the screen, for instance, as the result of a generic or host command executed at the other end. Operation is exactly like a file transfer.

DATA — The contents of the packet. Non-printable ASCII characters are prefixed with special characters and then converted to printable characters by complementing the seventh bit. Characters with the eighth bit set may also be prefixed and a repeated character can be prefixed by a count. A prefixed sequence of characters may not be broken across packets.

CHECK — The block check sequence , based on all the characters in the packet between, but not including the mark and the check itself, can be one, two or three characters in length, each character transformed by the char function. Normally, the single character checksum is used.

VITA

Zafrul Ahsan was born in Pabna, Bangladesh, on March 3, 1957. He attended elementary school in Dhaka. He was admitted to Momenshahi Cadet College, Tangail, in 1969. He completed both the secondary and higher secondary school educations from the same school. In 1977 he entered Bangladesh University of Engineering and Technology (B.U.E.T.) and in May 1981 he received the Bachelor of Science degree in Electrical Engineering.

Upon graduation in 1981, he was immediately employed in Bangladesh University of Engineering and Technology as a lecturer of Electrical and Electronic Engineering (E.E.E.) department. He worked with the E.E.E. department for about a year. Then he came to the U.S.A. for graduate studies in Electrical Engineering. He entered the Graduate School of the University of Tennessee, Knoxville, in September 1982. He received a Master of Engineering degree with a major in Electrical Engineering in December, 1985.