CROSS ASSEMBLER, TEXT EDITOR, AND LINKAGE DEVELOPMENT: PERSONAL COMPUTER AND SDK-85 MICROCOMPUTER

A Thesis Presented to

The Faculty of the College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Hwa-Shing Chen

June, 1983

ACKNOWLEDGEMENTS

I wish to express my gratitude to Professor Harold F. Klock whose guidance makes this work possible. Thanks are also due to my wife Wei-Li and my daughter Kaiting for their patience and understanding throughout this work.

Finally, I want to express my great appreciation to my parents for their spiritual support and encouragement.

TABLE OF CONTENTS

APPENDIX

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

In recent years, user-assembled computer kits have been widely used in schools. These single board computers contain all components required for basic system operation. The simplicity and flexibility make these computers well-suited for student experiments and simple user applications. However, minimal capabilities of these kits restrict system operation. The purpose of this thesis is to upgrade the SDK-85, MCS-85 System Design Kit, and thereby provide a working model for similar small system enhancement.

The SDK-85 basic system contains one page (256 bytes) of RAM memory and an 8085A microprocessor operating at a 3MHZ system clock. A built-in system monitor, a 6-digit LED display, and a 24-key keypad help the user to enter a machine code program and operate the system. On the prototype circuit board, a large wire-wrap area provides the capacity for system expansion and development. Like most of the simple microcomputer learning systems, the SDK-85 lacks the ability to process the symbolic language, and to manage the user files. The user must assemble his program and then enter the hexadecimal machine codes directly through the keypad every time. Due to these inefficiencies and inconveniences, enhancement of the SDK-85 operating capability is the objective of this project.

Development of a resident assembler and file management system require both extensive hardware and software expansion. Besides the editor/assembler and the file management software programming, other additional supporting developments may include ROM/RAM memory expansion, ASCII keyboard input handling, video display circuitry implementation, and floppy disk operating system design. In order to maintain the simplicity and flexibility of the SDK-85, the cross assembling scheme is adopted, instead of resident assembly. This means the SDK-85 enhanced operation is accomplished through the assistance of a complete computer as a host system. The editor/assembler and the file management programs for the SDK-85 are developed by using the existing facilities in the host system. Through the data communication channel, the host system is able to interchange information with the SDK-85, and command the SDK-85 to execute a specified program. In this way, only minor memory expansion and a data communication development are needed to let the SDK-85 perform any function ordered by the host system.

In this project, the OSI-C4PMF (former Ohio Scientific Inc.) microcomputer is selected to perform the role as the host system. The OSI-C4PMF is a 24K RAM machine based on the 6502 microprocessor with two serial ports and two parallel ports. One serial port is used to interchange data with the SDK-85, and send data to printer. The parallel ports are not used. Two floppy disk drives offer a total of 160K bytes of storage capacity for this system. In the system's firmware only a small monitor program and a DOS booting routine are provided. The disk operating system, OS-65 DOS, and the BASIC language interpreter are loaded from disk to RAM locations by user's request. To take advantage of DOS, a software development system for the SDK-85 is designed and operated in the OSI-C4PMF.

The SDK-85 software development system created in this project includes a Text Editor program, an 8085 Cross Assembler program, an SDK-85 Extended Monitor program, and a group of 6502 assembly language subroutines called by the extended monitor for data communication. Except for these assembly language subroutines, the programs are written in the BASIC language. Each of these BASIC programs is loaded from disk to workspace of the OSI-C4PMF by proper menu selection.

Figure 1.1 explains the overall system operation in functional block diagram form. Through the assistance from the system, the user is able to edit the 8085 assembly language source file by using the Text Editor, and is able to call the Assembler to translate this symbolic language to an 8085 machine code program. The object code file generated by the Assembler then can be allocated to the SDK-85 memory locations by the Extended Monitor. The Extended Monitor not only performs the Loader function, it also offers the data modifications and disk file maintenance capabilities which are not available in the SDK-85 resident monitor. A memory buffer managed by the Extended Monitor simulates any 2K range of the SDK-85 memory. The user may order the Extended Monitor to copy a block of memory contents of the SDK-85 into the memory buffer for modification or filing. Therefore, the user is able to enter, debug, and save his program more efficiently.

The structure and algorithm of each hardware/software implementation are detailed in the chapters to follow. Chapter 2 presents the memory expansion and data communication hardware implementation on the SDK-85 system circuit. Chapter 3 depicts the

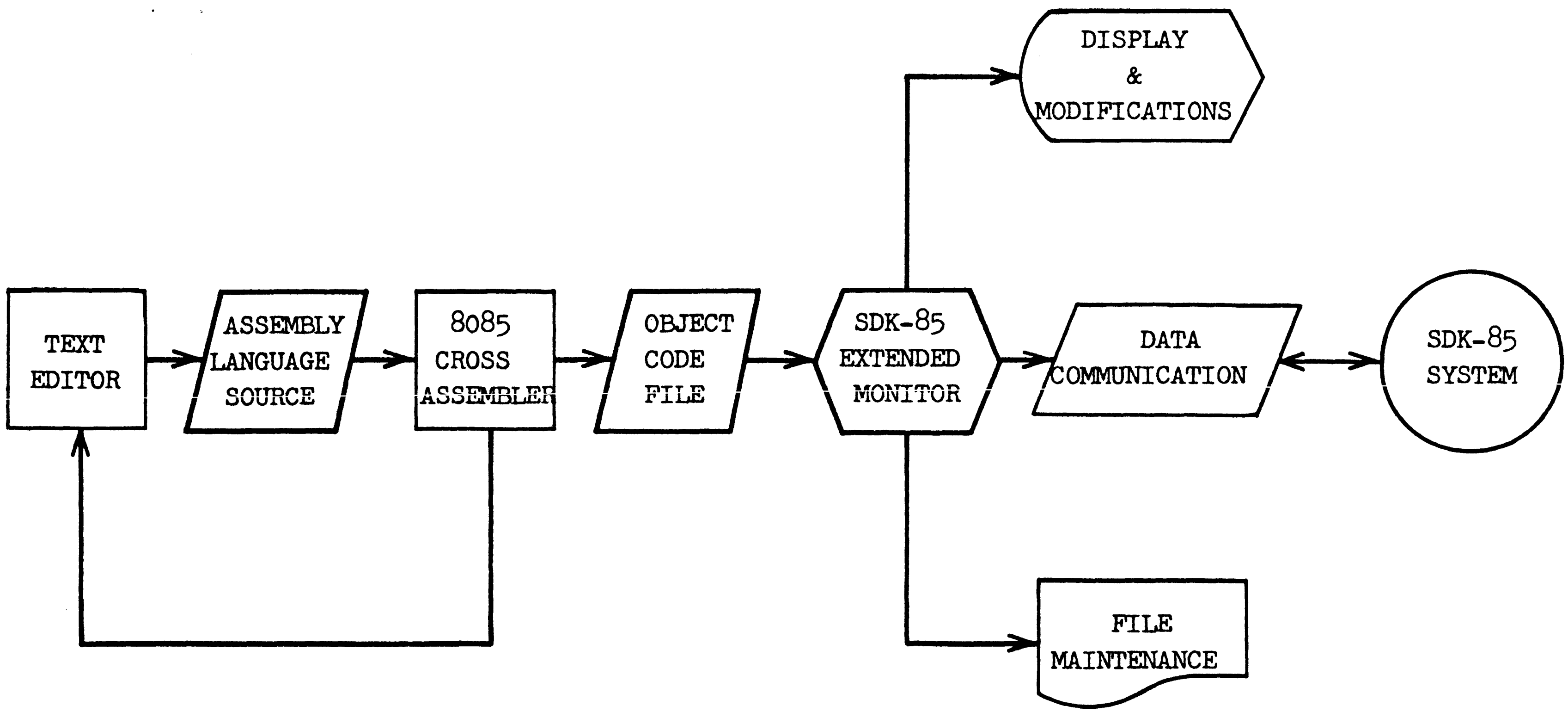

FIGURE 1.1 SDK-85 Development System Functional Block Diagram

hardware/software design of the data communication between the SDK-85 and the OSI-C4PMF. It also explains the execution procedure of each assembly language communication routine of both systems. Chapter 4 highlights the overall linkage of the software developed and executed in the OSI-C4PMF. Chapter 5, 6, and 7 describe the program logic for the Extended Monitor, the Text Editor, and the Assembler respectively. Chapter 8 uses a typical example to demonstrate the operation and performance of this development system. Chapter 9 summarizes what has been accomplished and what possibilities still exist for improvement. Appendix A provides the explanation on the Assembler error code messages. Seven other appendixes document those developed programs in source listing form.

CHAPTER 2 EXPANSION OF THE SDK-85 SYSTEM

2.1 Basic System

The SDK-85 is a simple microcomputer system based on the Intel 8085A microprocessor. In addition to the 3MHZ 8085A CPU, this on-board system also includes the following devices:

8355 2K ROM with I/O

8155 256 Byte RAM with I/O Ports & Timer

8205 3 to 8 Decoder

8279 Programmable Keyboard/display Interface

Hexadecimal Keypad/display Circuit

TTY Interface Circuit

The SDK-85 monitor program resides in 8355 ROM memory, from hexadecimal location 0000 to location 07FF. It provides utility functions employing either a teletypewriter, terminal, or the on-board keypad. Only one page of RAM is provided by the 8155 for user programming. This RAM can be addressed at locations 2000 to 27FF. One page of 8155 RAM thus occupies eight pages of mapped memory. Multiple copies of RAM are due to incomplete decoding of the 8155.

On the circuit board, prototype space is allocated for additional 8355/8755 expansion ROM and 8155 RAM. For further enhancement of the basic system, an optional expansion driver area is provided. This may not be addressed by the 8205, but affords space for 8212 latches and 8216 buffers for driving auxiliary systems. The optional expansion drivers leading to the board's prototyping area are enabled only over the address range 8000-FFFF.

2.2 Expanded System

As described in the previous section, the fundamental system does not have enough memory space to accommodate the complete development program. Therefore, a minimal hardware expansion is required.

In order to be more flexible in further developments, the method of expanding optional driver area was adopted. By installing two 8212 address latches and five 8216 buffers in the appropriate board position, the external decoding circuit and external memory devices could be developed in the wire-wrap area.

2.2.1 Expansion Driver Circuits

The circuit layout of the expansion driver area was already designed and printed by the manufacture in the upper right region of the SDK-85 circuit board.

One 8212 latch is employed for address/data bus demultiplexing (DA0-DA7). Another 8212 buffers the unmultiplexed half of the address (A8-A15), and five 8216 drivers, buffer the data bus and control signals. All buffered I/O buses are connected to the external circuits through the bus expansion connectors J1 and J2. A completed circuit diagram of this expansion driver area is duplicated in Figure 2.1.

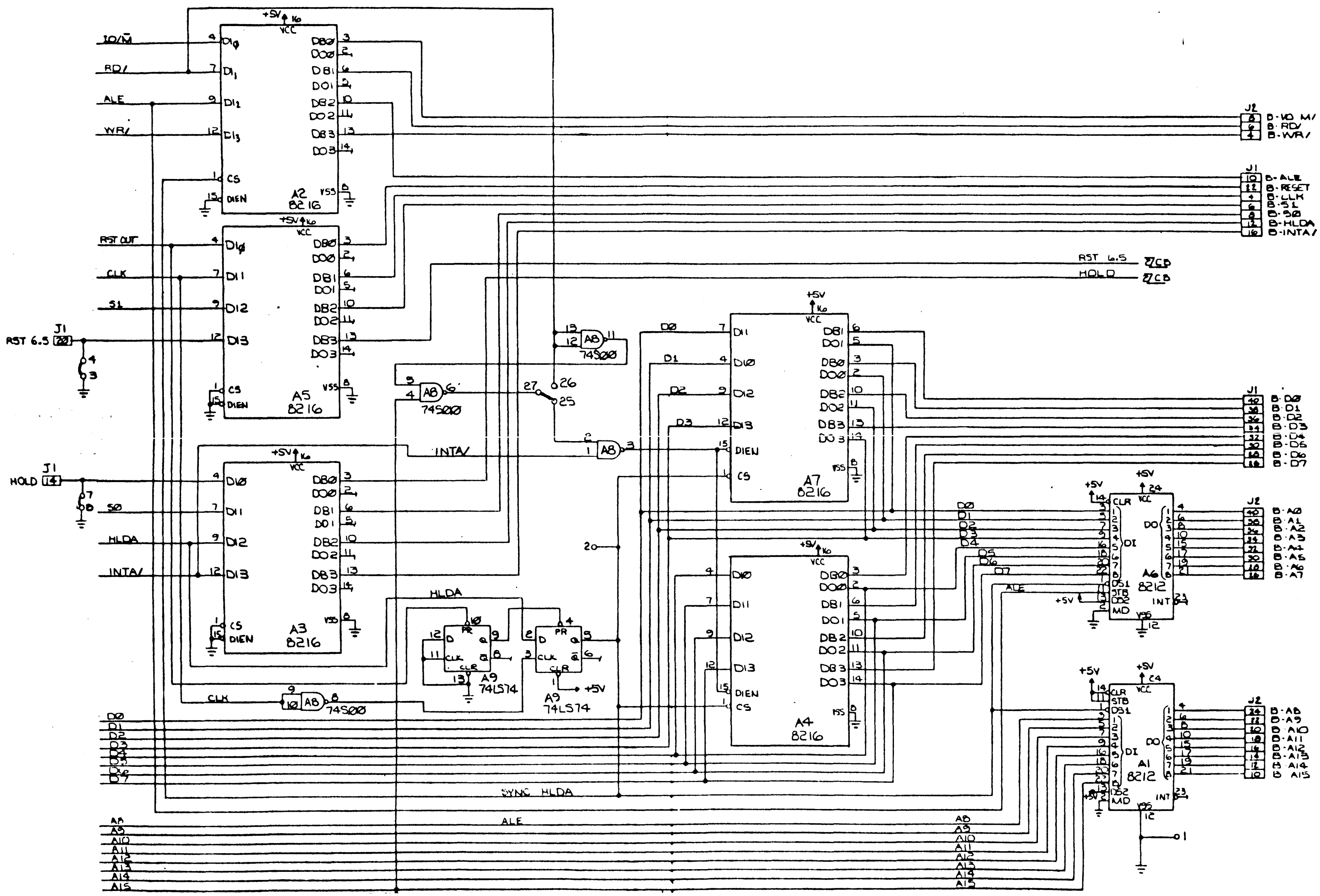

FIGURE 2.1 SDK-85 Expansion Driver Circuit Diagram

As can be observed in the circuit diagram, the address line A15 must be high (logic 1) to enable the 8216 data bus buffer/drivers. This allows the bus expansion drivers to be enabled only when the upper 32K memory locations (8000-FFFF) are addressed.

Since no external interrupt is used, the input pins for RST 6.5, INTR, and HOLD are disabled by fixing the corresponding jumpers to ground. If later developments require any of these external inputs, Chapter 3 of the SDK-85 User's Manual should be reviewed.

2.2.2 External Expansion

The external expansion circuits are located on the upper left hand side of the SDK-85 circuit board. This also identifies the wire-wrap area. Circuitry here interfaces to the basic system via connectors J1 and J2. Thus, the external expansion circuits may be divided into two categories, extended memory, and a data communication circuit. An off-board 74138 address decoder enables the applicable component. Figure 2.2 shows the external expansion circuit diagram.

The external expansion memory components include a 2716 2Kx8 EPROM and two 2114 1Kx4 static RAM chips. Like the original system, each output from the external address decoder enables a 2K block of addresses. The 2716 EPROM is addressed from 8000 to 87FF in absolute addressing. The 2114's are mapped at 9000-93FF in the lowest access range, and a duplicate resides at 9400-97FF in the highest access range. Figure 2.3 presents the SDK-85 memory map after expansion.

The data communication circuit is composed of an MC6850 ACIA

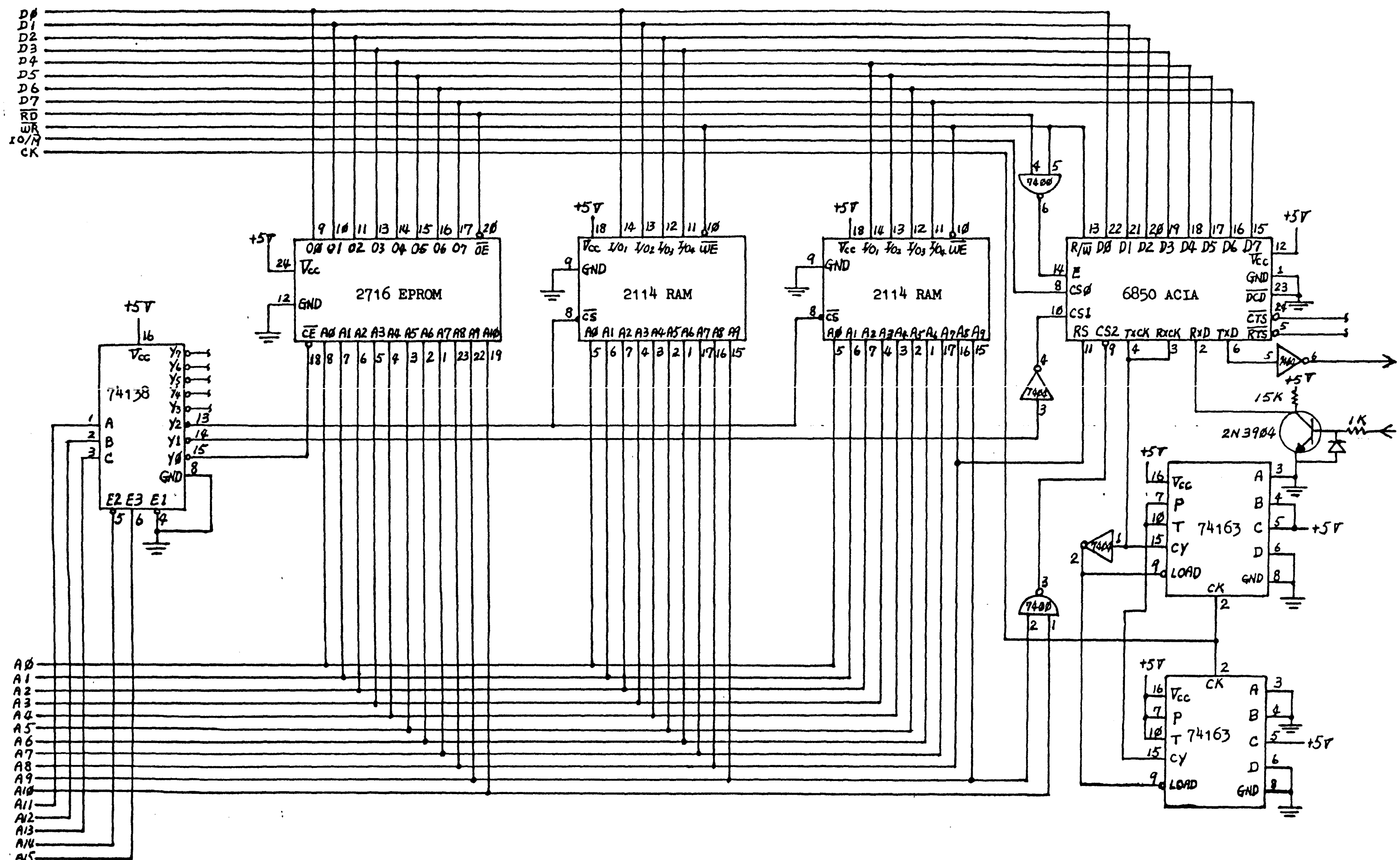

FIGURE 2.2 SDK-85 External Expansion Circuit Diagram

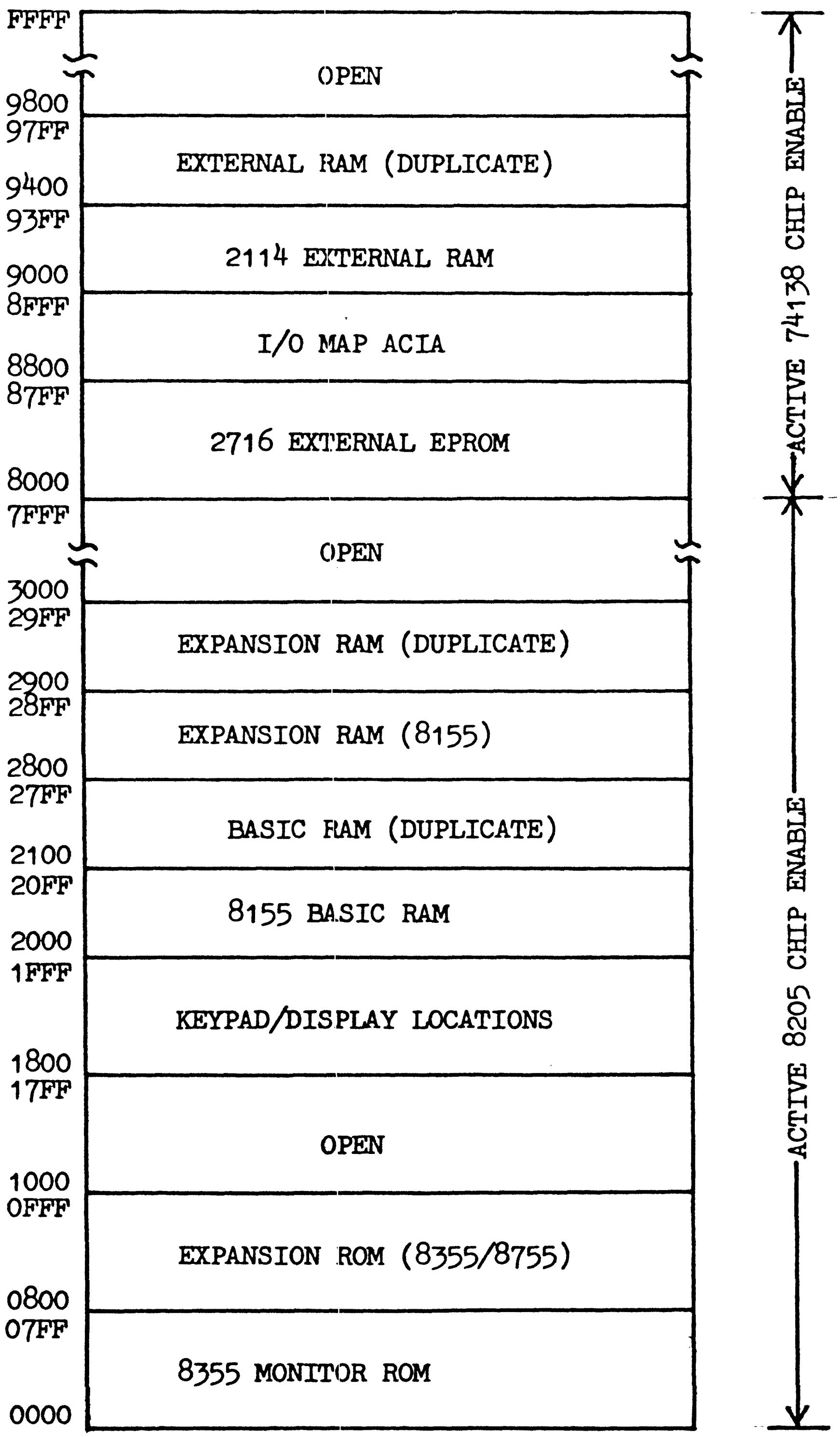

FIGURE 2.3 SDK-85 Expanded System Memory Map

(Asynchronous Communications Interface Adapter), two 74163 4-bit Presettable counters, a 7400 quad 2-input NAND gate, and a 7404 hex inverter. As noted in Figure 2.2, the 6850 ACIA is addressed by I/O mapping instead of memory mapping. Two I/O addresses, 8E and 8F, are assigned to the ACIA Control/status register and Transmit/receive register respectively. The 3MHZ system clock is divided by cascaded 74163's in order to generate a 19.23KHZ clock for the ACIA. This clock will be devided by 16, in programming the ACIA, to obtain the 1200 baud rate.

CHAPTER 3 DATA COMMUNICATION

3.1 Hardware Design

The data communication link between the SDK-85 and the OSI-C4PMF is an asynchronous, serial data handler which transmits or receives data bytes at a fixed rate of 1200 baud (1200 bits per second). Two ACIA's (Asynchronous Communication Interface Adapters), Motorola 6850's, were used to perform this task. One of 6850's was already part of the original OSI-C4PMF I/O circuitry. The other 6850 was added to the external expansion board of the SDK-85 system, and is the ACIA of interest in this section.

The added ACIA handles serial data communication at a rate of 1200 baud. This means the ACIA transmits or receives one byte of data bit by bit, as eight data bits. The 8 bits are preceded by one start bit and followed by one stop bit. Each byte requires approximately 8.33 ms to transmit all bits. This is much slower than the instruction execution time of either microcomputer system. For this reason, the handshaking between the two systems during communication can be implemented by software rather than hardware. However, before the software handshaking takes over, the hardware must be ready. Both ACIA $\overline{RTS}$ (Request-To-Send) output pins are connected to each other's $\overline{CTS}$ (Clear-To-Send) input pins in order to perform hardware handshaking for the System-ready signal. When both ACIA's are ready, the software takes over.

As mentioned in Chapter 2, the 8085 CPU, of the SDK-85 system, provides 256 bytes of I/O dedicated memory. Two of these I/O memory locations are assigned to the ACIA. Hexadecimal address 8E is the Control/Status register, and address 8F is the Transmit/Receive register. Because these memory locations can be accessed like an I/O port, they can be both written to or read from. That means each location performs the function of two registers.

Since the MC6850 was developed mainly for direct interfacing with the 6800 and 6500 series microprocessors, it is necessary to add gating for its interfacing with the 8085-based system. As pictured in Figure 2.2, the input signal (E) for enabling the I/O data buffer is given by NANDing the $\overline{RD}$ and $\overline{WR}$ output pins of the 8085 CPU to generate an active-low signal for reading from or writing to the ACIA. The R/$\overline{W}$ input pin of the ACIA is connected to the $\overline{WR}$ output pin of the 8085 to determine the direction of ACIA data flow.

The clock circuitry, as shown in Figure 2.2, is implemented by two cascaded 74163 presettable counters, which divide the SDK-85 3MHZ system clock by 156 to generate 19.23KHZ for the ACIA. This clock input will be divided by 16, in programming the ACIA, in order to get 1.2KHZ for the actual data clock.

Refer to Figure 2.2 detailing the complete data communication circuit diagram.

3.2 Software Structure

The real-time data communication software in both OSI-C4PMF and SDK-85 are written in the corresponding machine language. The user controls these machine language programs through an OSI-C4PMF BASIC language program called Extended Monitor, which is detailed in

Chapter 5. For communication structure, the OSI-C4PMF is the host system which gives a command and/or initialization information to the slave system, SDK-85.

Four commands, TRANSMIT, RECEIVE, RUN, and RESET were developed for communication between the OSI-C4PMF and the SDK-85. TRANSMIT and RECEIVE are employed to interchange data between two systems. RUN orders a specified SDK-85 program to be executed. And RESET terminates the data communication channel.

Each command is represented by an ASCII character. When the OSI-C4PMF user issues a communication command to the BASIC language program (Extended Monitor), the OSI-C4PMF transfers the execution control to the proper 6502 machine language subroutine. First, the software tests the hardwired handshaking line. A warning message will be returned to BASIC, if the SDK-85 is not ready. Otherwise the corresponding ASCII command byte is sent to SDK-85. Upon recognizing this ASCII encoded command, the SDK-85 transmits the same ASCII byte back to OSI-C4PMF for command verification. No further information is sent, unless that command is verified by OSI-C4PMF.

Except for RESET, the other three functions require the OSI-C4PMF to provide further information to the SDK-85. RUN needs the OSI-C4PMF to supply the starting address of the specified program. TRANSMIT and RECEIVE require not only the starting address for initializing a SDK-85 data location pointer, but also the length of data string for setting up a byte counter.

Both systems accumulate the checksum when each data byte is transmitted or received. After completion of data transmission, the checksum maintained by SDK-85 is sent to OSI-C4PMF for checksum verification. The error status is returned to the BASIC calling program, and translated to a proper message for prompting the user.

3.3 SDK-85 Communication Program

The algorithm for this 8085 machine code program, which accepts commands from the OSI-C4PMF host system and executes the specified command routine, can be viewed in the generalized form shown in Figure 3.1.

After turning the SDK-85 power on, hardware initialization is necessary in order to transfer control to this communication program which resides at SDK-85 starting location 8227 (hexadecimal). At the beginning of this program, the ACIA undergoes reset. This is followed by a program sequence which writes to the ACIA Control register specifying 10 bits per data byte (1 start bit + 8 data bits + 1 stop bit), divide-by-16 mode and low output state on the $\overline{RTS}$ (Request-To-Send) pin. The purpose of this low output state is to indicate that the SDK-85 is ready. Next, a small routine repetitively checks the status of the communication link between the OSI-C4PMF and the SDK-85. When the OSI-C4PMF is ready to transmit, the routine is exited.

The next step in program execution is that of waiting for an input command. This is also the re-entry point for most of the command routines when previous commands have been executed. After an input byte is received, the command recognition routine compares this input to the contents of the command table, as shown in Figure 3.2.

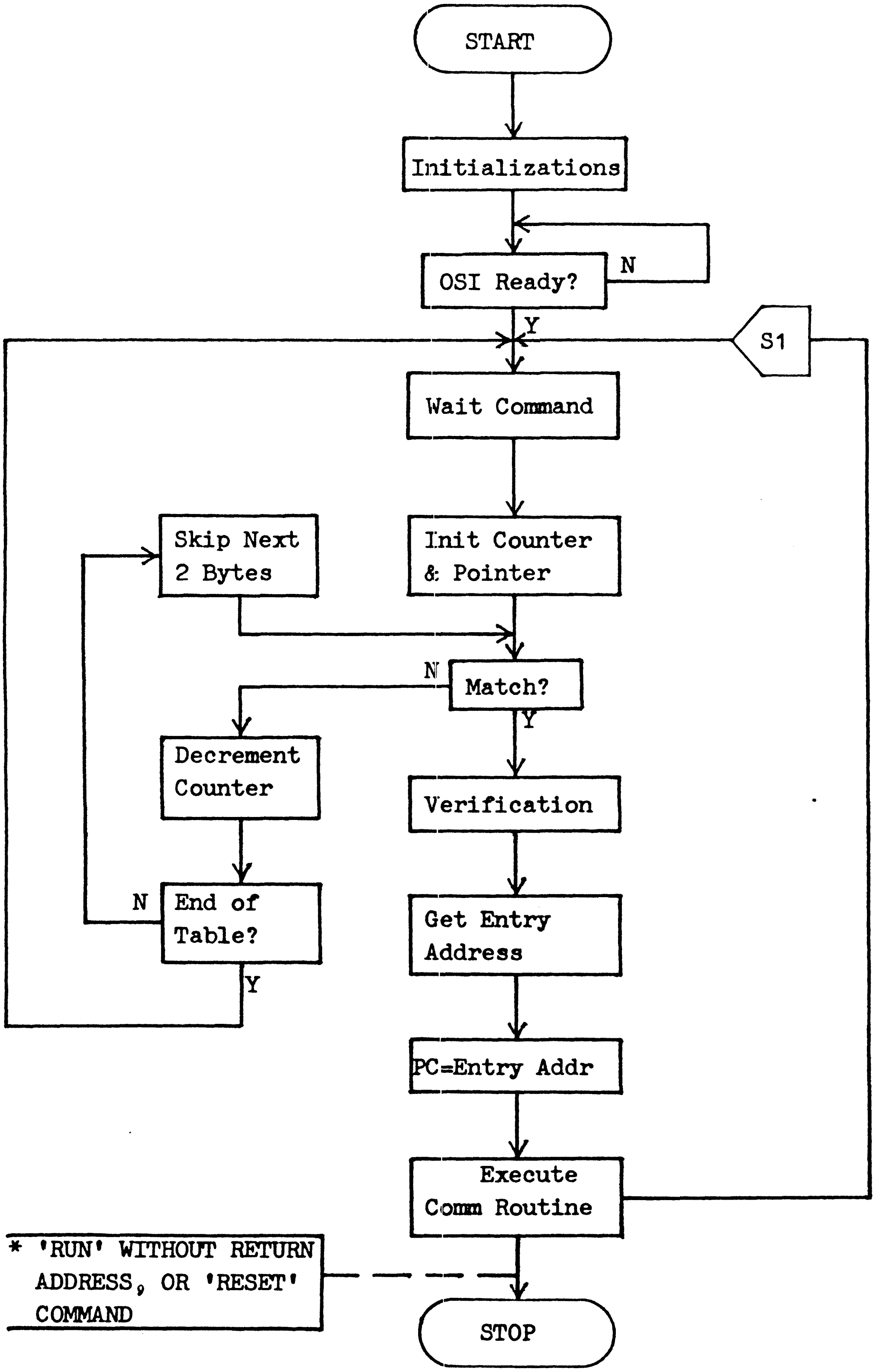

FIGURE 3.1 Flowchart for Main Program Structure of SDK-85

Address	Contents
82C9	TRANSMIT COMMAND BYTE (4F)
82CA	ENTRY ADDRESS OF ROUTINE TRANSMIT (8258)
82CB	
82CC	RECEIVE COMMAND BYTE (49)
82CD	ENTRY ADDRESS OF ROUTINE RECEIVE (827A)
82CE	
82CF	RUN COMMAND BYTE (52)
82D0	ENTRY ADDRESS OF ROUTINE RUN (828A)
82D1	
82D2	RESET COMMAND BYTE (45)
82D3	ENTRY ADDRESS OF RST 1 IN MONITOR (0008)
82D4	

FIGURE 3.2 SDK-85 Command Table Structure

If the input is identical to the command indicated by the command pointer, then the next two bytes in the table are loaded into the 8085 CPU Program Counter. These bytes form the starting address of the selected command routine. Any unrecognized input command takes the flow of execution back to the point of command entry.

In order to deal with the characteristics of the ACIA, two widely used subroutines were developed. One is called DATAIN which tests the status of RDRF (Receive Data Register Full) of the ACIA and returns with input data in the Accumulator (Register A). The other subroutine is called EMPTY which examines the status of TDRE (Transmit Data Register Empty) and returns control to the calling routine when this register is ready for the next data transmission. Figure 3.3 and 3.4 present the flowchart for these two subroutines respectively.

The RESET command causes the data communication program to transfer control back to the SDK-85 built-in monitor firmware. Since the two bytes following the RESET command byte in the table form the monitor entry location, no execution routine is developed for this command. If the communication channel is needed later, the re-entry procedures must be performed on the SDK-85 keypad.

The other three command routines are described in the sections to follow.

3.3.1 TRANSM Routine

When a TRANSMIT command is received, the execution control is transferred to this routine. TRANSM transmits a block of SDK-85

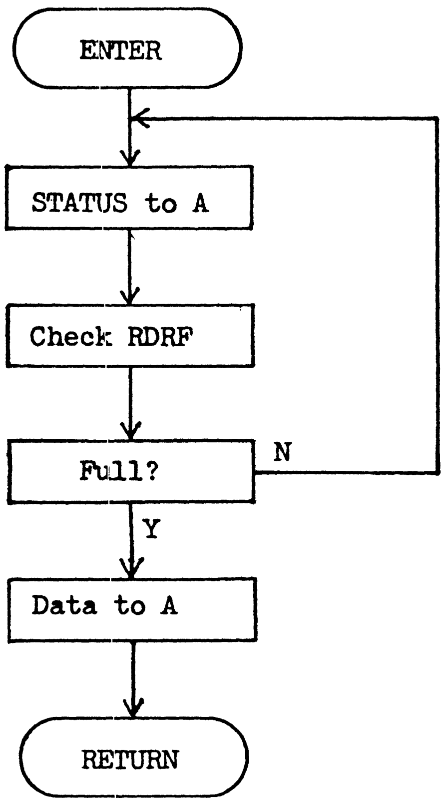

FIGURE 3.3 Flowchart for Subroutine DATAIN

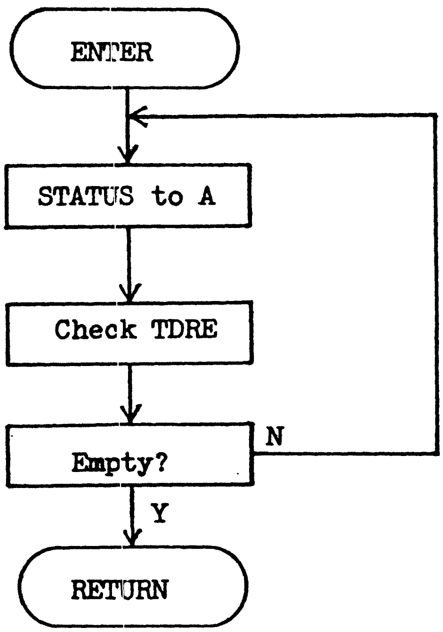

FIGURE 3.4 Flowchart for Subroutine EMPTY

memory contents to the OSI-C4PMF. The execution flowchart can be viewed in Figure 3.5.

At the beginning of this routine, the execution logic sets up an address pointer in Registers H & L and a byte counter in Registers D & E. These information are provided by the host system (OSI-C4PMF). Before transmitting the specified data block, Registers B & C are cleared for using as a checksum accumulator. Each data byte is added to the checksum after being transmitted.

The error checking procedure is entered when the byte counter reaches zero. First, the high-byte of checksum (Register B) is sent to the host system for comparison. Then the execution logic waits for the host system to send its checksum high-byte. Upon receiving a byte from the ACIA, a comparison is made to check if the two checksum high-bytes are the same. As depicted in the flowchart, the low-byte of checksum (Register C) is sent if no error on the high-byte comparison. The error checking is ended with transferring control to the main program for the next command entry.

3.3.2 RECEIV Routine

Corresponding to the RECEIVE command, this routine accepts a block of data bytes, and locates the received data to memory locations specified by the host system. Figure 3.6 presents the flowchart for this operation.

As noted in the figure, the execution flow of this routine is very similar to TRANSM routine. The difference is that instead of outputting data bytes, RECEIV inputs data bytes. The checksum

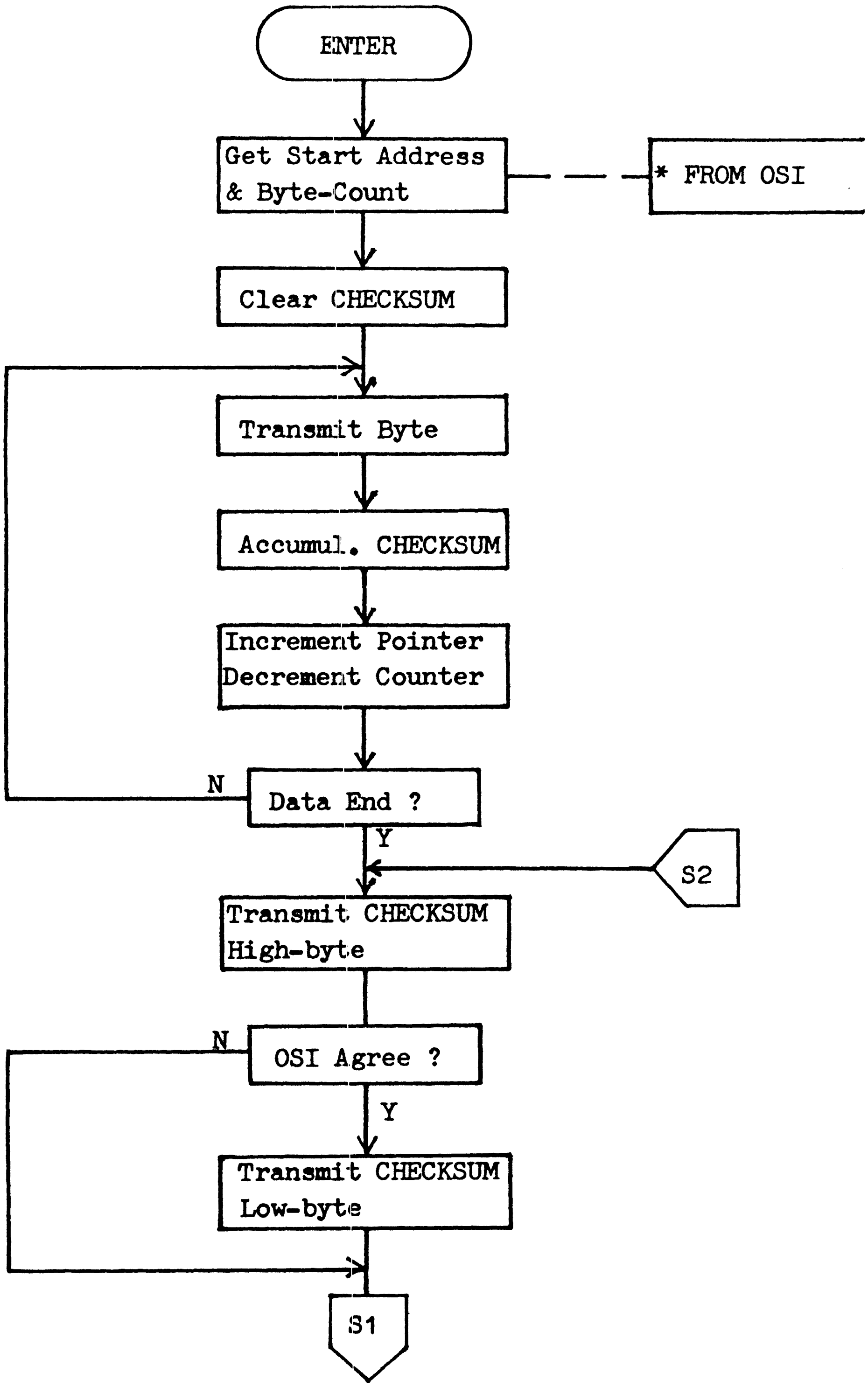

FIGURE 3.5 Flowchart for SDK-85 Routine TRANSM

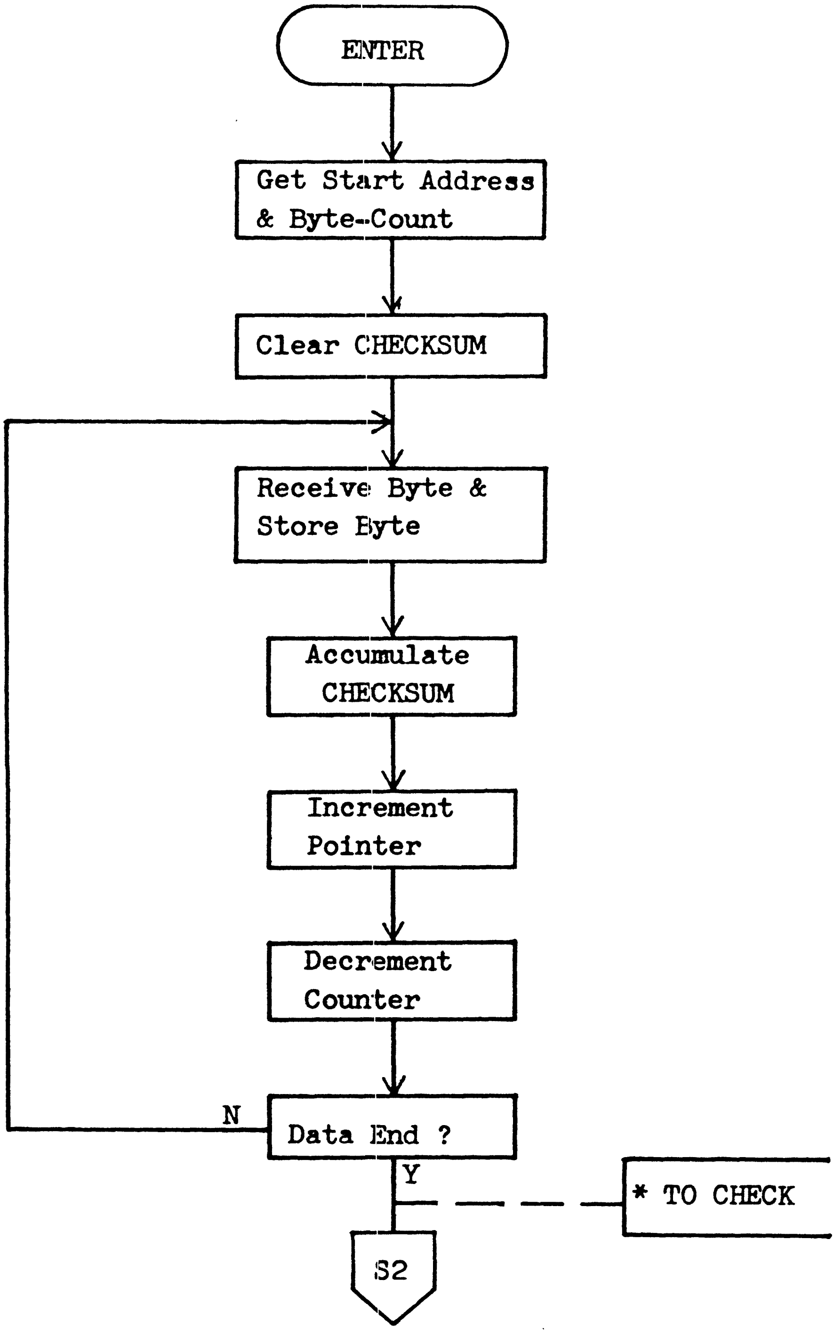

FIGURE 3.6 Flowchart for SDK-85 Routine RECEIV

checking procedures, as described in the previous subsection, are shared by both TRANSM and RECEIV routines.

3.3.3 RUN Routine

The purpose of RUN routine is to transfer execution control to the program specified by the host system. As shown in Figure 3.7, this routine is started by obtaining the starting address of the specified program from the OSI-C4PMF. Before loading the starting address to the Program Counter, the address for re-entering communication program is pushed into the stack memory.

In order to restore the communication channel, the specified program must not be a looping structure and must include an RET (return from subroutine) instruction. Otherwise, the data communication is discontinued. This makes the communication program treat the specified program as a subroutine.

3.4 OSI-C4PMF Communication Program

In this section, the communication program written in the 6502 machine language is discussed. This program, in fact, is composed of a group of assembly language subroutines and command table information. As mentioned, the BASIC program, Extended Monitor, provides mutual interchange of information between the user and these assembly language subroutines. It interacts with the user to pass the communication parameters, and the assembly language subroutines implement the real-time communication work with the SDK-85.

These machine codes are stored at the first sector of the 39th

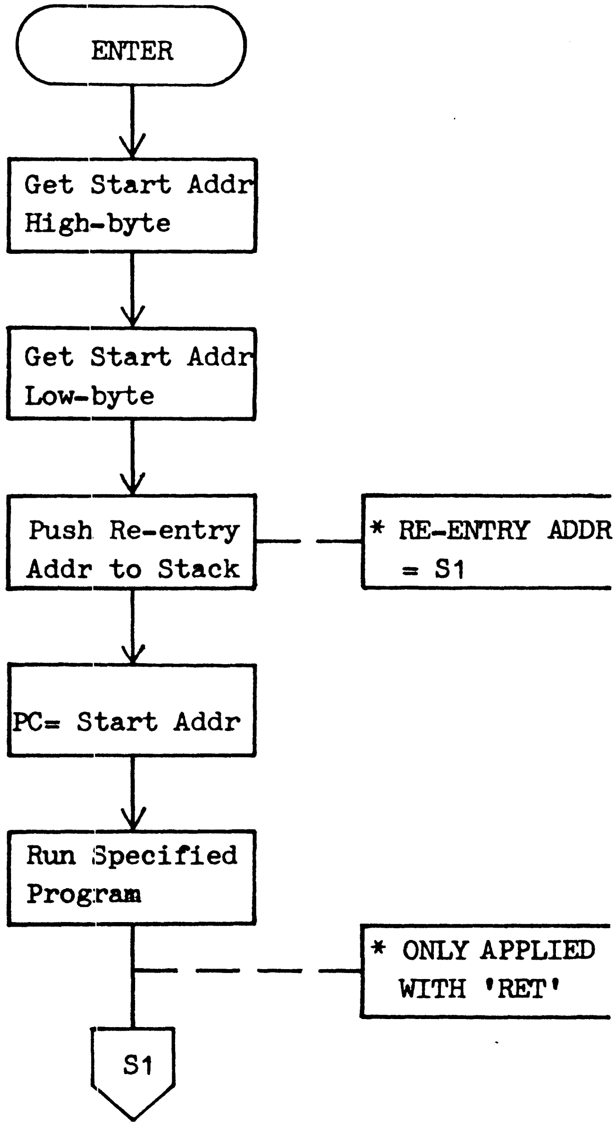

FIGURE 3.7 Flowchart for SDK-85 Routine RUN

track on disk. They are loaded to the OSI-C4PMF memory locations starting from hexadecimal address 5E00 after the Extended Monitor program is located to BASIC workspace. Figure 3.8 shows the memory map for these assembly language subroutines. As noted in the map, memory locations starting from 5EDD to 5EE2 are assigned to pass the information set up by the BASIC routine to the assembly language subroutines. Location 5EE5 is used as a message byte which contains the error status code. Upon returning to BASIC, this location is read by Extended Monitor program, and the content is interpreted as an appropriate message to inform the user. The following statements list the error status codes and the corresponding interpretations:

00 - Error free

01 - SDK-85 is not ready

02 - SDK-85 recognized a wrong command

03 - Transmission error (checksum error)

TRANSM, RECEIV, RUN, and RESET are the four major subroutines called by the corresponding command routine in BASIC. To support these major subroutines, certain housekeeping subroutines are employed. These supporting subroutines are explained in flowchart form shown in Figures 3.9, 3.10, and 3.11.

3.4.1 TRANSM Subroutine

The function of this major subroutine is to transmit a string of data bytes from the OSI-C4PMF memory locations to the SDK-85.

Label	Address	Contents	
	0000	⋮	
MPLO	009B	LOCAL MEMORY POINTER LOWBYTE	LOAD FROM IMAGES
MPHI	009C	LOCAL MEMORY POINTER HIGHBYTE	
		⋮	
	5E00	6502 ASSEMBLY LANGUAGE PROGRAM	
	5EDC		
IMLO	5EDD	IMAGE OF MPLO	SETUP BY BASIC
IMHI	5EDE	IMAGE OF MPHI	
BYCLO	5EDF	BYTE COUNTER LOWBYTE	
BYCHI	5EE0	BYTE COUNTER HIGHBYTE	
STALO	5EE1	START ADDRESS LOWBYTE	
STAHI	5EE2	START ADDRESS HIGHBYTE	
CHKLO	5EE3	CHECKSUM LOWBYTE	CLEARED BY BASIC
CHKHI	5EE4	CHECKSUM HIGHBYTE	
MSG	5EE5	MESSAGE BYTE	
CMDTB	5EE6	TRANSMIT COMMAND BYTE (4F)	
	5EE7	RECEIVE COMMAND BYTE (49)	
	5EE8	RUN COMMAND BYTE (52)	
	5EE9	RESET COMMAND BYTE (45)	
		⋮	
	5FFF		

FIGURE 3.8 OSI-C4PMF Data Communication Program Memory Map

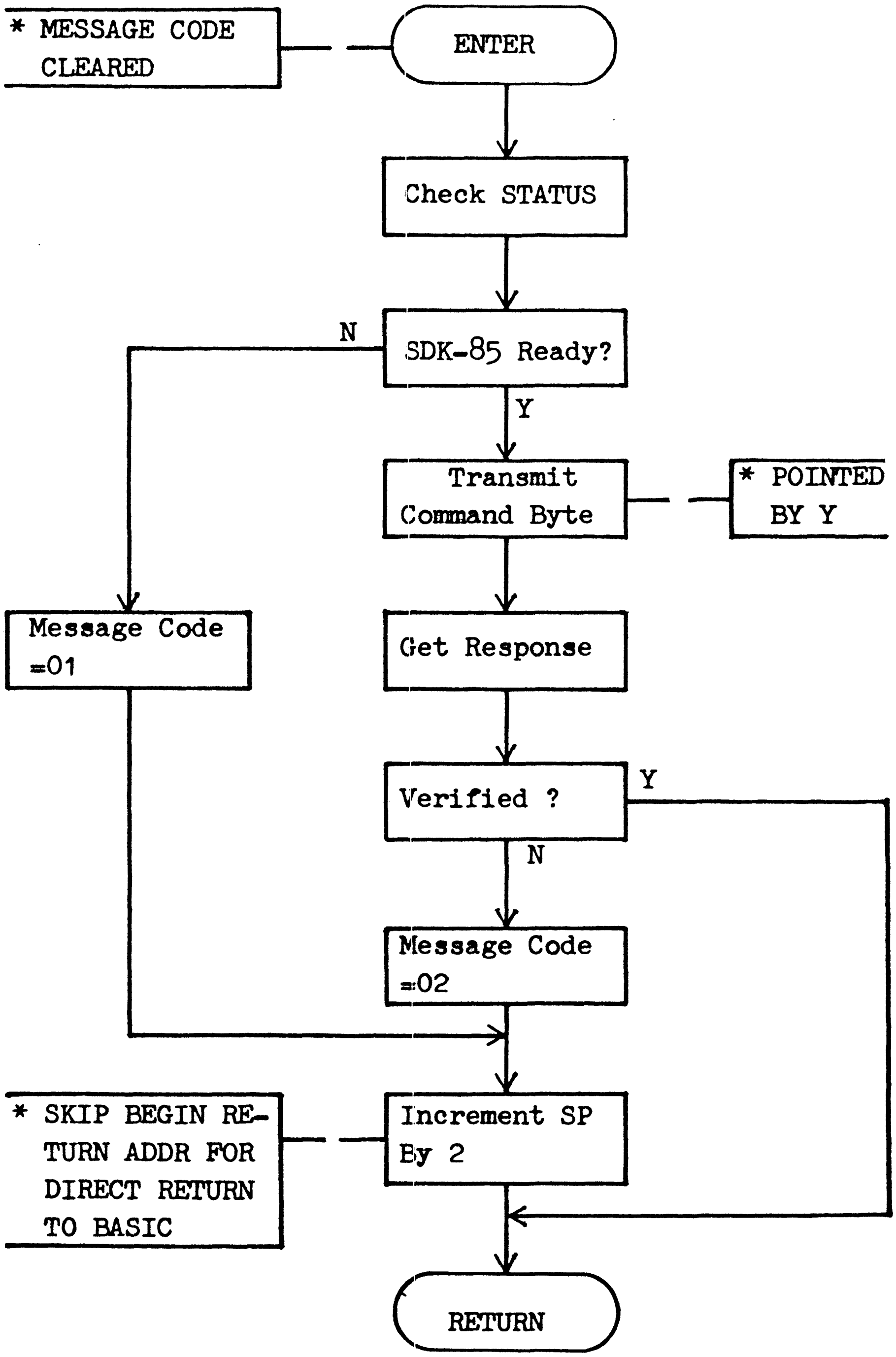

FIGURE 3.9 Flowchart for OSI-C4PMF Subroutine BEGIN

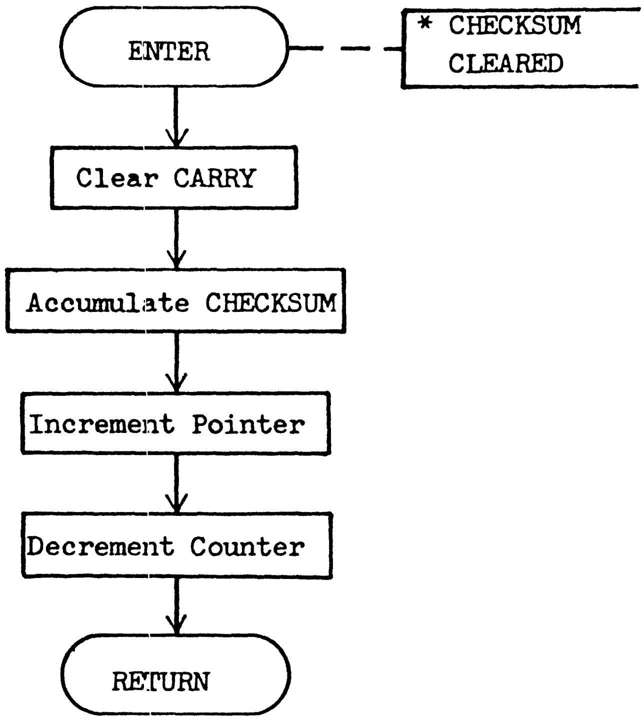

FIGURE 3.10 Flowchart for OSI-C4PMF Subroutine CHKSUM

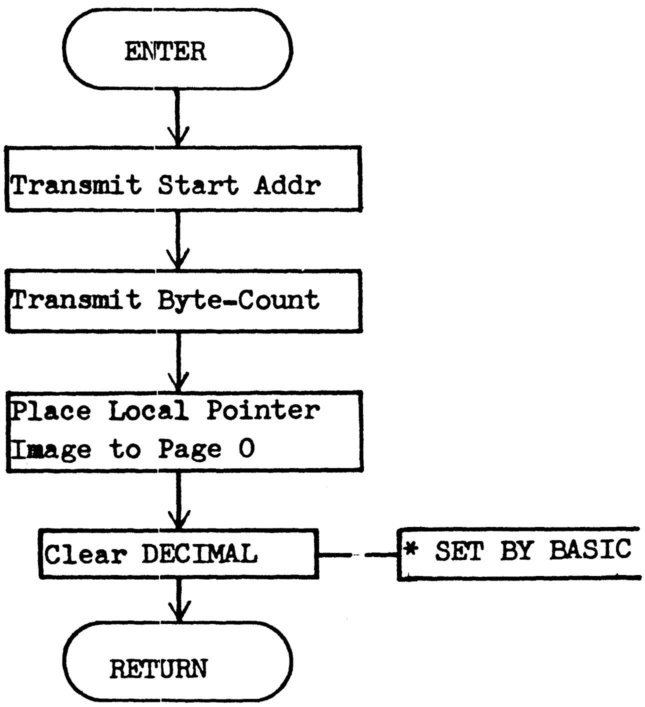

FIGURE 3.11 Flowchart for OSI-C4PMF Subroutine *SETUP*

Before data transmission begins, certain procedures are executed. First, a test on the hardwired handshaking status ($\overline{\text{CTS}}$ status) is performed to ensure the SDK-85 is in the READY state. No further procedures will be executed, if this test fails. Second, the RECEIVE command byte pointed by Register Y is transmitted to order the SDK-85 to enter the receiving mode. After the SDK-85 responded command is verified, the data string's starting location in the SDK-85 memory and the string's length are sent in sequence. Then the OSI-C4PMF local memory pointer, which marks the positions of the data string, is reflected from its image to page 0 locations in order to perform the indirect addressed data fetching. To be able to accumulate the hexadecimal checksum, the DECIMAL bit of the 6502 CPU's Status register is cleared. These procedures are implemented by calling the subroutines BEGIN and SETUP in sequence.

Upon returning from the SETUP subroutine, the data string transmission begins. When a data byte is sent to the ACIA, the subroutine CHKSUM is called to accumulate the transmitted data byte to the checksum. CHKSUM also increments the memory pointer, and decrements the byte counter. This procedure is repetitively executed until the byte counter reaches zero.

To check the data transmission error, the checking logic requires the SDK-85 to send its checksum high-byte for comparison. OSI-C4PMF then echoes its checksum high-byte to the SDK-85. If both high-bytes are the same, the comparison on the low-bytes is proceeded. As shown in Figure 3.12, any checksum mismatching leads to an error code to be loaded into the message byte location.

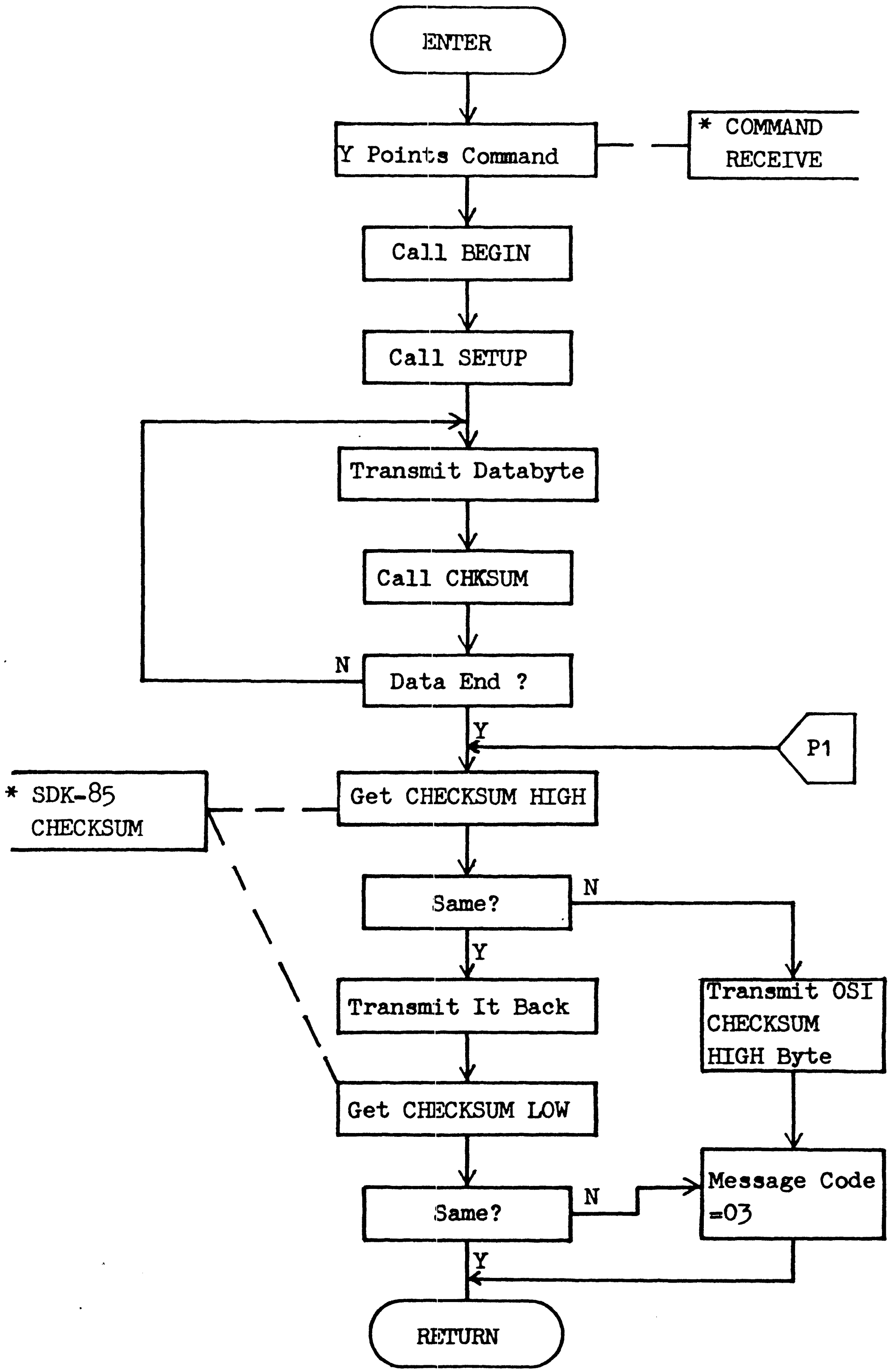

FIGURE 3.12 Flowchart for OSI-C4PMF Subroutine TRANSM

3.4.2 RECEIV Subroutine

As pictured in Figure 3.13, the structure of RECEIV is similar to TRANSM subroutine described in the previous subsection. But, unlike TRANSM, this major subroutine orders the SDK-85 to enter the transmitting mode, and receives a string of data bytes specified by BASIC from the SDK-85.

After calling the subroutines BEGIN and SETUP to send ASCII command TRANSMIT and the initialization data to the SDK-85, the execution logic starts receiving data bytes from the slave system, and allocates the received data to memory location addressed by the local memory pointer. As for TRANSM, the subroutine CHKSUM is also employed here to accumulate the checksum, and prepare for the next coming byte.

The checksum checking procedure is shared by both RECEIV and TRANSM, and is covered in the preceding subsection.

3.4.3 RUN Subroutine

This major subroutine is entered when the user orders the SDK-85 to execute a specified program.

First, the subroutine BEGIN is used for ready-checking and command transmission. Then the starting address of the user specified program is transmitted to the SDK-85 in high byte and low byte order. Figure 3.14, shows the execution sequence for this subroutine.

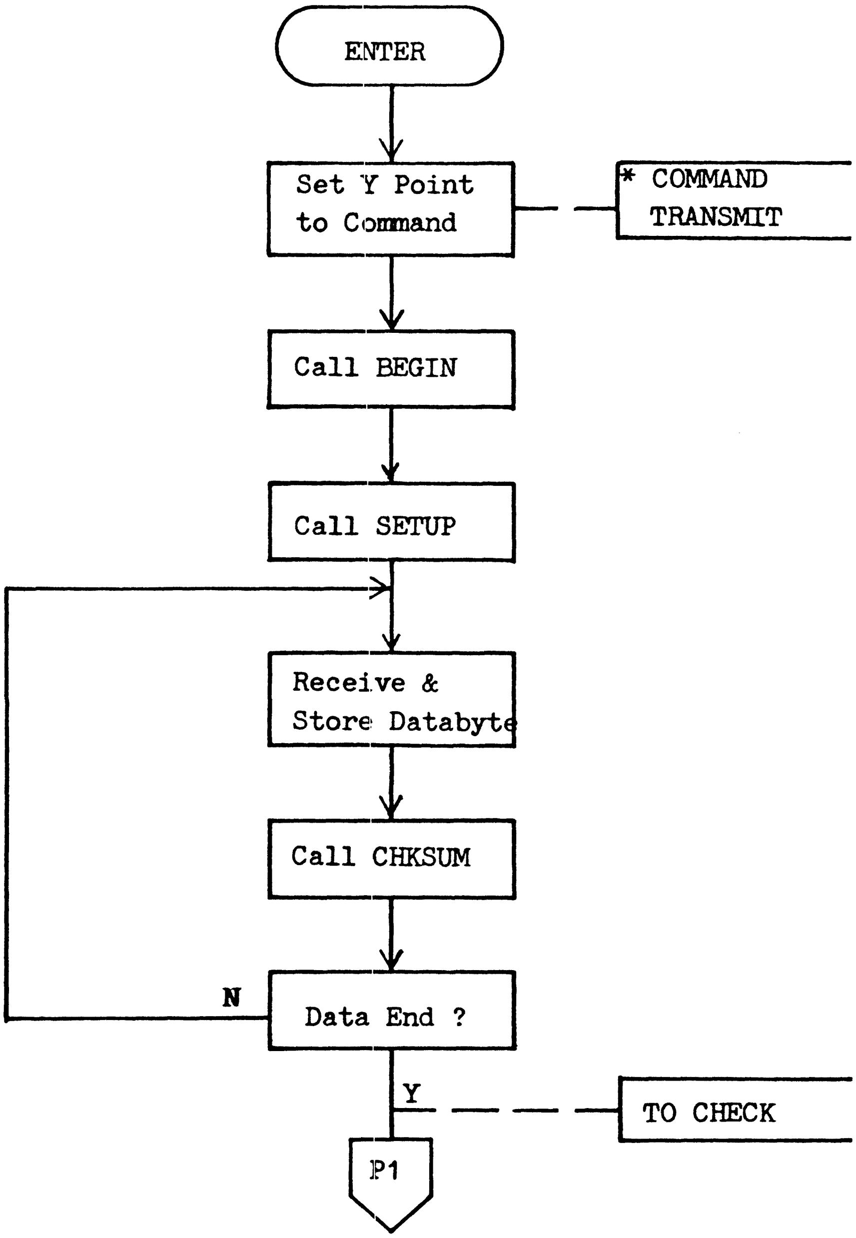

FIGURE 3.13 Flowchart for OSI-C4PMF Subroutine RECEIV

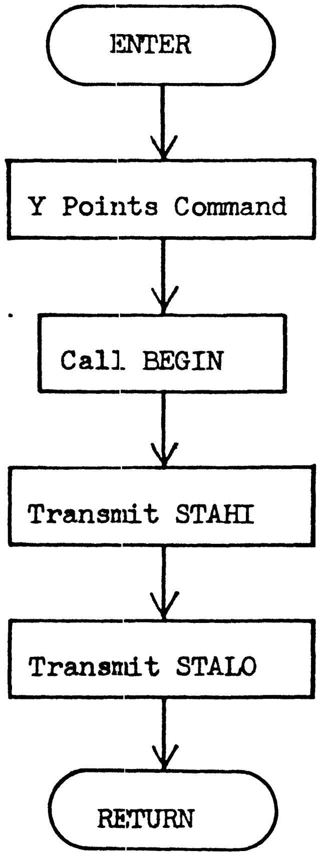

FIGURE 3.14 Flowchart for OSI-C4PMF Subroutine RUN

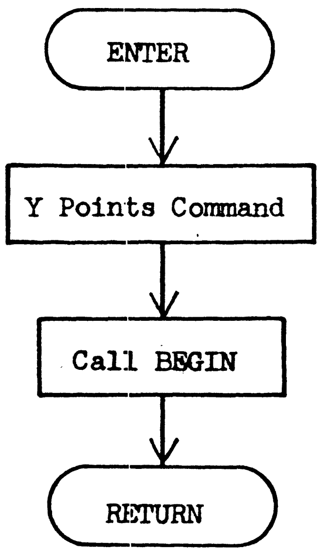

FIGURE 3.15 Flowchart for OSI-C4PMF Subroutine RESET

3.4.4 RESET Subroutine

As presented in Figure 3.15, the purpose of this subroutine is simply to transmit the RESET command to the SDK-85. After setting up the Y register to point the RESET command byte, the subroutine BEGIN is called to perform the ready-test, command transmission, and command verification.

CHAPTER 4 EXECUTIVE SYSTEM DEVELOPMENT

4.1 Disk Operating System of OSI-C4PMF

The SDK-85 development system is based upon the OSI-C4PMF (6502) microcomputer. All of the system software developed for the SDK-85 is executed by the OSI's BASIC interpreter and linked through the disk operating system (DOS).

The OS-65 DOS formats a 5 1/4" diskette to forty tracks (0-39), and eight sectors per track. Each sector holds 256 bytes. Each track accommodates 2K bytes. Therefore, a formatted diskette may store total of 80K bytes. The DOS and the system utility software occupy the first fourteen tracks (0-13) of disk. The BASIC program directory is stored at track 21. The remaining twenty five tracks can be used to save the user programs.

The OSI-C4PMF is a 24K RAM machine. Figure 4.1 shows the memory assignment of the OSI-C4PMF disk operating system. Like most of the microcomputer systems, only a small routine resides permanently in firmware for booting DOS from disk after reset. As soon as DOS acquires execution control and configures the system, it loads the BASIC program located on the 14th track of the disk into the workspace, and executes it immediately. This small greeting program can then be used to assign execution to other existing programs on the disk. This technique is referred to as an auto-run feature.

4.2 Development Software and Its Executive Program

The SDK-85 software development system is a group of BASIC

0000 00FF	6502 PAGE ZERO
0100 01FF	6502 STACK
0200 22FF	TRANSIENT PROGRAM AREA FOR USER'S LANGUAGE PROCESSOR
2300 265B	I/O HANDLERS
265C 2A4A	FLOPPY DRIVERS
2A4B 2E78	DISK OPERATING SYSTEM (DOS)
2E79 3178	PAGE 01/1 SWAP BUFFER
3179 3278	DOS EXTENSIONS
3279 327D	SOURCE FILE HEADER INFORMATION
327E 5FFF	SOURCE FILE WORKSPACE

FIGURE 4.1 OSI-C4PMF Disk Operating System Memory Map

programs designed to enhance the operation of the SDK-85 microcomputer. The developed software tools include a Text Editor, a Cross Assembler, and an Extended Monitor. The Text Editor provides the functions for editing the assembly language source file; the Cross Assembler converts the assembly language source codes to the 8080/8085 machine codes; the Extended Monitor performs the data interchanging with the SDK-85 and offers the data modifications, and the binary file maintenance capabilities. To link these BASIC programs, an executive program is also developed.

Currently, the software developed is a single-disk operation system. All the developed BASIC programs, the 6502 machine language program, and the associated reference data reside in one disk. Figure 4.2 presents the disk track assignment for the SDK-85 development system.

To take advantage of auto-execution, the greeting program is designed to be the executive program of the SDK-85 software development system. It not only provides a menu to link all development software, but also changes system configuration appropriate to the function selected by the user.

At present the menu includes the three development programs for the SDK-85, and a function FREE which releases the full workspace for user programming. Figure 4.3 presents the overall software development system structure. As noted in the flowchart, only the Assembler is able to enter the other programs without transfer through the System Executive program.

The algorithm of the System Executive program is depicted in

TRACK	USE
0-13	OS-65 DOS VERSION 3.2
14	SDK-85 DEVELOPMENT SYSTEM EXECUTIVE PROGRAM
15-18	SDK-85 EXTENDED MONITOR
19-20	TEXT FILE EDITOR
21	OS-65 DOS DIRECTORY
22-25	8080/8085 CROSS ASSEMBLER
26-28	ASSEMBLER LISTING PROGRAM
29-30	EXTENDED TEXT FILE
31	USER BINARY FILE I
32	USER BINARY FILE II
33	USER BINARY FILE III
34	USER BINARY FILE IV
35	USER BINARY FILE V
36	ASSEMBLED OBJECT CODE FILE
37-38	FIRST TEXT FILE
39	Sector 1 - 6502 MACHINE LANGUAGE SUBROUTINES & TABLE
39	Sector 2 - USER BINARY FILE DIRECTORY
39	Sector 4 - ASSEMBLER REFERENCE TABLE CONTENTS PAGE 1
39	Sector 5 - ASSEMBLER REFERENCE TABLE CONTENTS PAGE 2

FIGURE 4.2 Disk Track Use Assignment

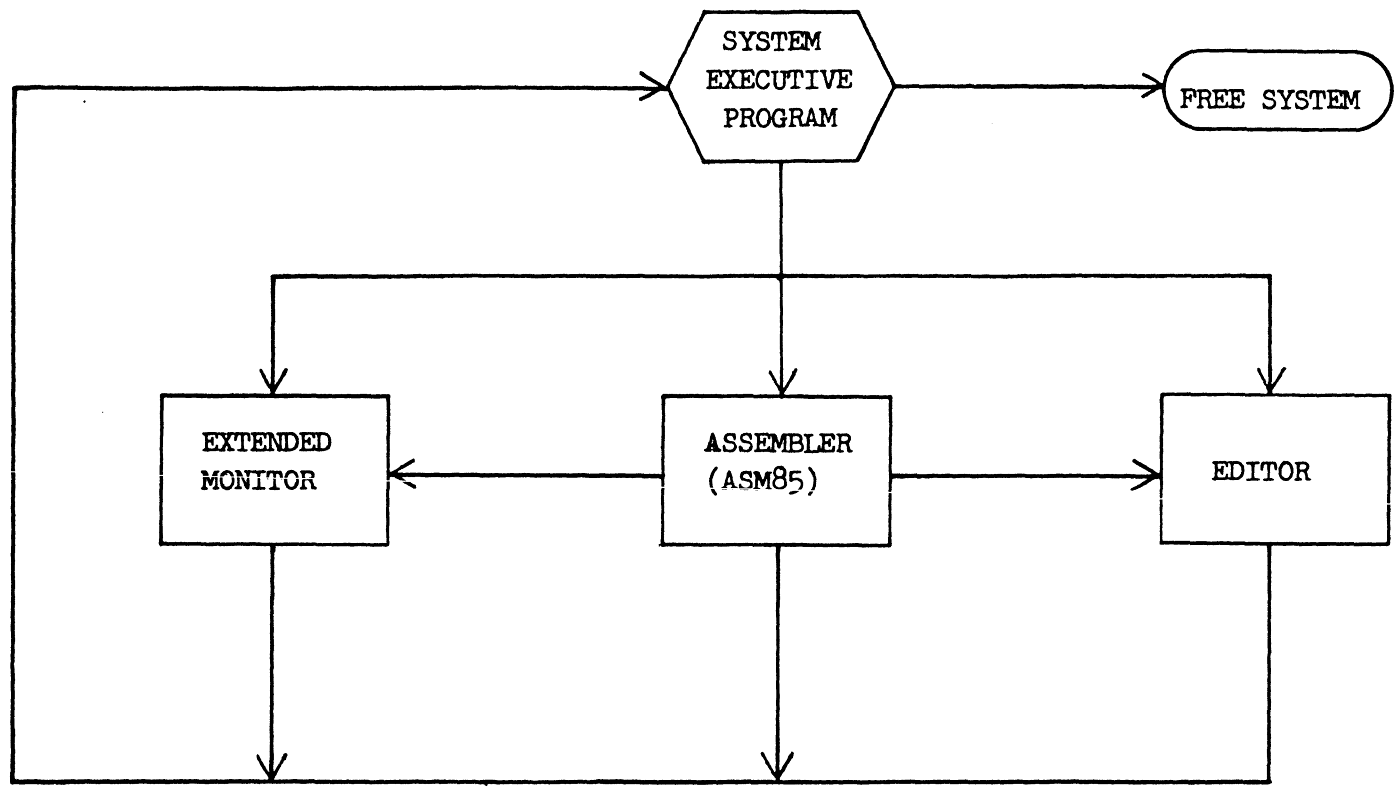

FIGURE 4.3 Overall Software Development Structure

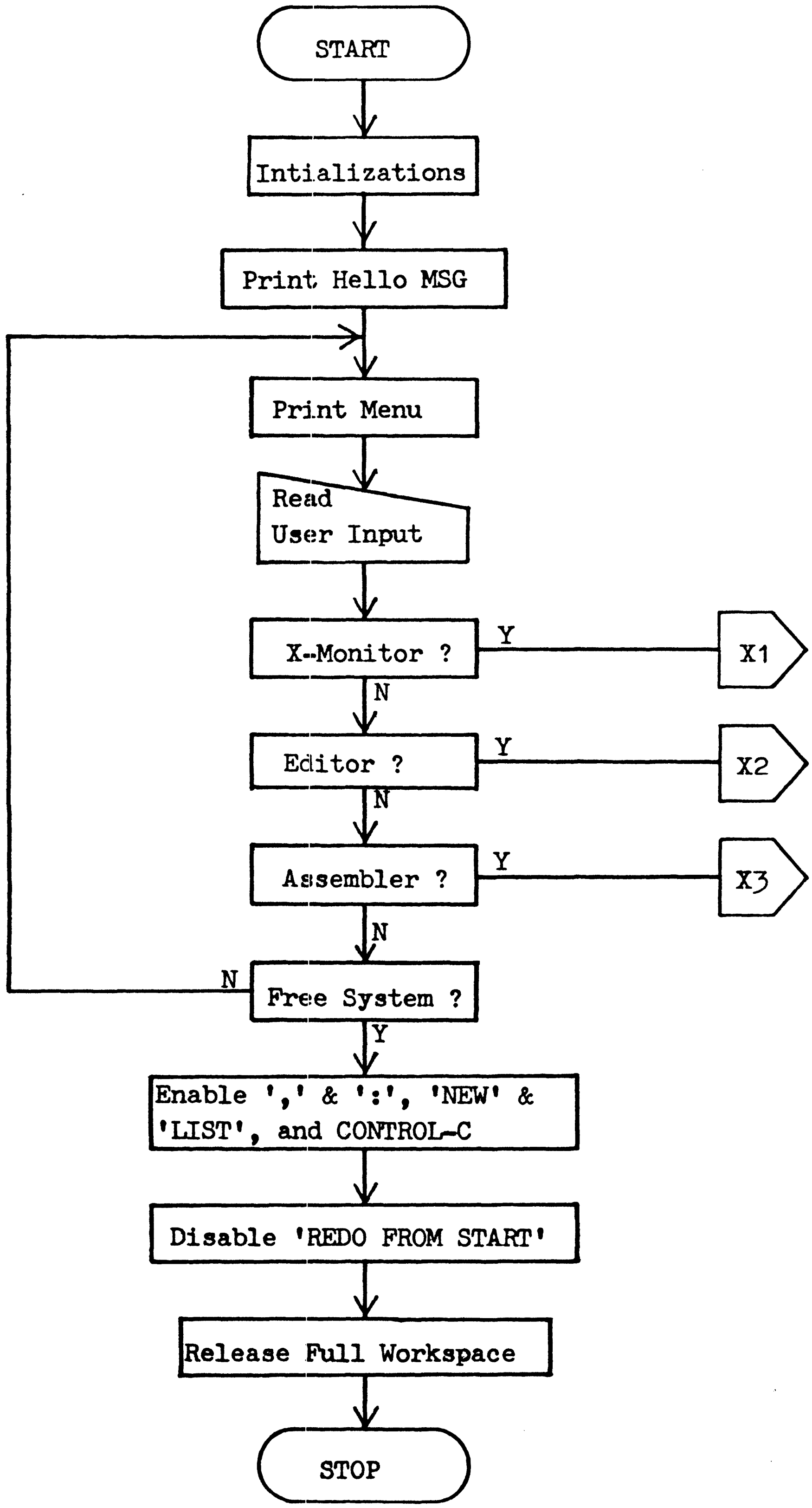

FIGURE 4.4 Flowchart for System Executive Program

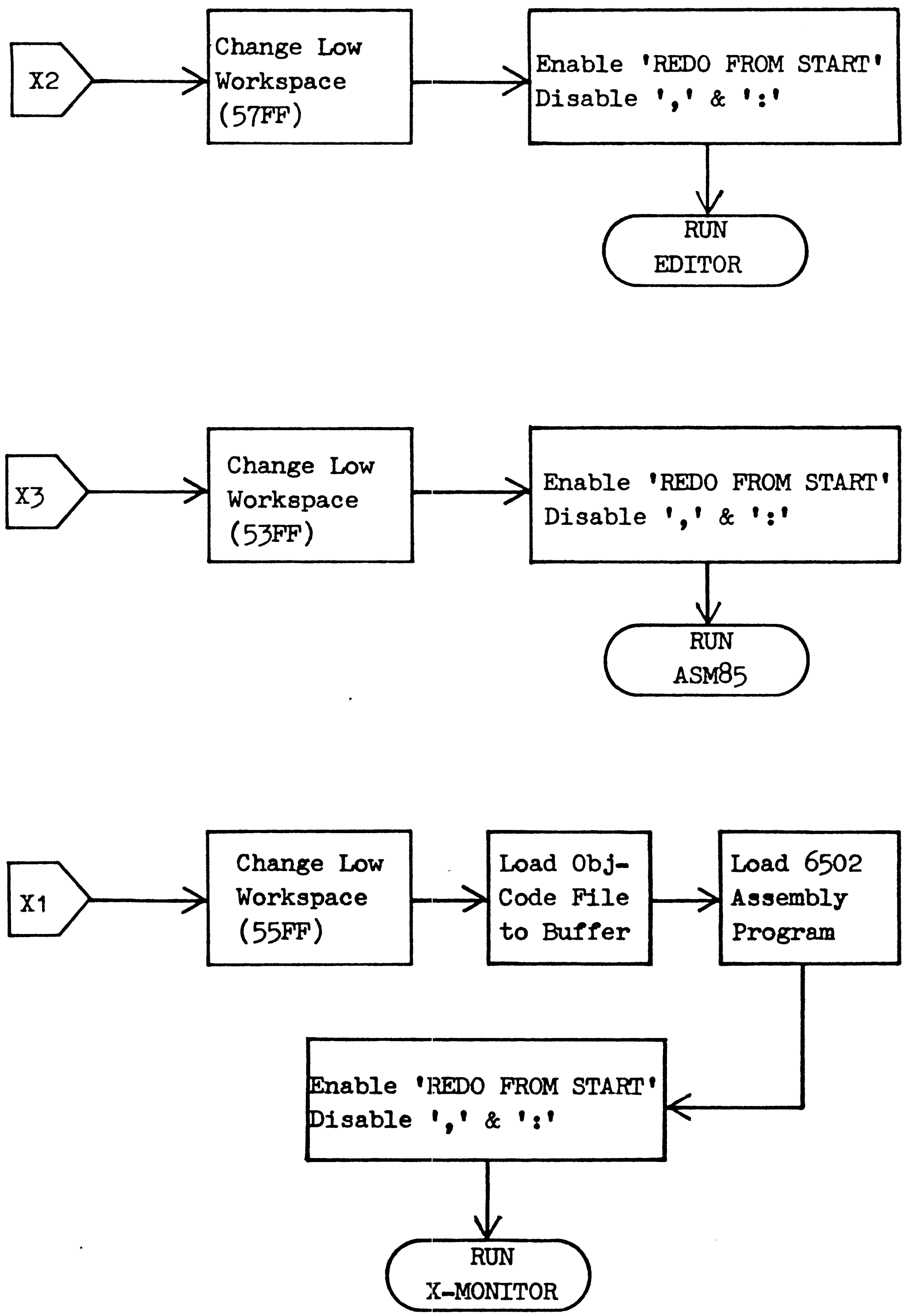

FIGURE 4.5 Flowchart for Executive Routines

Figure 4.4 and 4.5. In order to work with the DECWRITER IV printer, which may be operated only at 300/110 baud rate, the ACIA is reconfigured to the 300 baud rate. The ACIA is also used by the Extended Monitor to communicate with the SDK-85, at a 1200 baud rate.

As marked in the flowcharts, the System Executive program reconfigures certain system features for the menu selected program before execution control is transferred. In general, two major changes are made. First, the lower limit of the DOS workspace is redefined for protecting the corresponding buffer. This ensures that the DOS does not interfere with the buffer area just beneath the workspace. Second, the 'REDO FROM START' message is enabled, and the BASIC string terminators ',' & ':' are disabled. In doing so, the chance of losing execution control due to user's failure is minimized. For instance, if a null input were accidentally entered, the 'REDO FROM START' would be displayed to avoid re-entering the program.

The FREE function offers a chance to let the user to escape from the development system program. The entire workspace is assigned, the 'REDO FROM START' is enabled, the LIST & NEW commands are enabled, and the CONTROL-C function is restored. Before transferring control back to DOS, the System Executive program clears itself from the workspace. When the DOS prompt 'Ok' is displayed on the screen, the system is ready for user programming.

CHAPTER 5 EXTENDED MONITOR

5.1 Overview

This program was developed for the purpose of supporting housekeeping functions for the SDK-85 development system. It provides enhanced abilities, which are not available in the SDK-85 built-in monitor, such as disk file storage, data block move and insertion, and screen/printer display, etc. These capabilities are enabled since the Extended Monitor program is executed on the OSI-C4PMF system, rather than on the SDK-85 itself. Therefore, the most important functions are those data communication commands which can give orders to the SDK-85 for interchanging data.

Figure 5.1 presents the map of memory assignment. As may be noted, the OSI-C4PMF locations from 5600 to 5DFF act as a data buffer simulating SDK-85 memory. The first two buffer locations store the starting SDK-85 address; the next two locations store the ending SDK-85 address. The remaining bytes hold a facsimile of SDK-85 data. The first four reference addresses reflect the actual memory locations where the data block should be located in SDK-85 memory. Therefore, the 2K OSI RAM buffer contains a memory model of the SDK-85 system.

In the BASIC program, two variables ST and DN are assigned to represent, in decimal, the starting and ending memory image address values respectively. These addresses and the corresponding values may be specified by the user, or may be updated by certain command routines. The Extended Monitor program also maintains a pointer (BS)

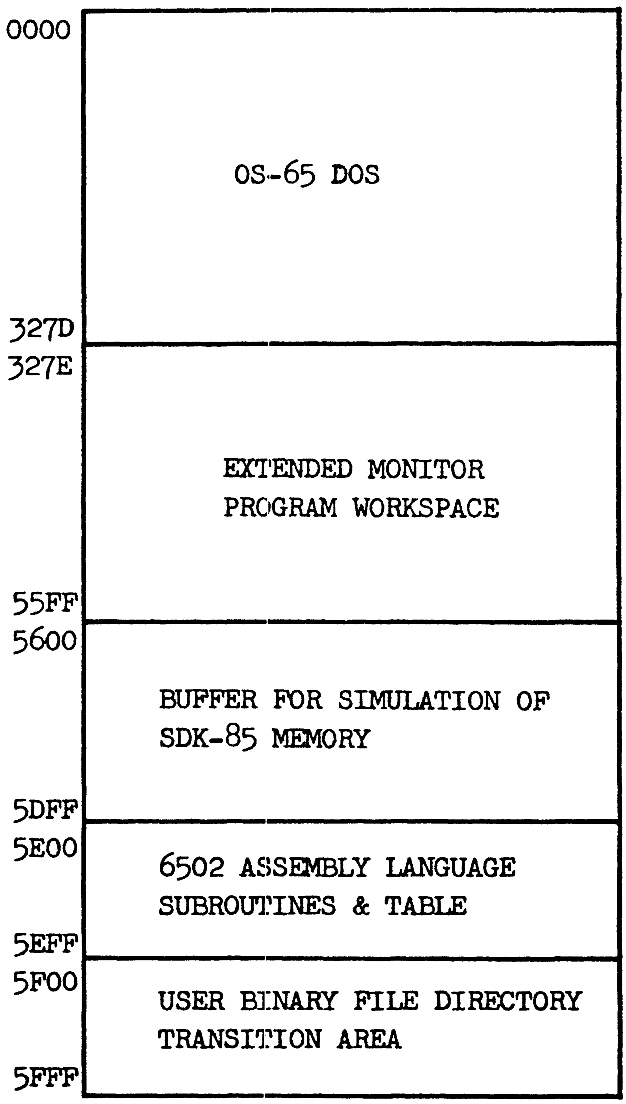

FIGURE 5.1 Memory Map for Extended Monitor

which always targets the OSI-C4PMF address of the first byte of the buffer (5600). This makes the local address (SA) of any data in the buffer obtainable by taking the difference between ST and the user specified address NS, and adding it to BS.

As listed in Figure 5.2, sixteen commands were developed to perform various tasks. These commands can be classified under the following functional groups: data communication commands, memory display & modification commands, and disk file maintenance commands. By manipulating these commands in the Extended Monitor, the user may send the object code file to the SDK-85 memory and execute it, or may get a block of data from the SDK-85 and save it as a disk file unit. The user may also modify or rearrange the current data file in the buffer area, or may display a list of contents of the file on screen or printer.

The algorithms of how to implement these commands are explained in the following sections.

5.2 Command Format

The command string should consist of the syntax field and/or the specification field. Any non-alphanumeric characters can be employed as a separator between these fields. Only if the first character of the specification field is a decimal digit, can the field separator be omitted.

In the syntax field, a command entry must be provided. Since the command logic recognizes the leftmost two characters only, a two letter abbreviation for the command is allowed. In certain command

	SYNTAX FIELD	SPECIFICATION FIELD	DESCRIPTION
DATA COMMUNICATION COMMANDS	DUmp	XXXX - YYYY (CR) XXXX (CR) (CR)	Dumps contents of XXXX through YYYY to SDK-85 Dumps contents of XXXX through end of simulated memory to SDK-85 Dumps entire contents of simulated memory to SDK-85
	GEt	XXXX - YYYY (CR) XXXX (CR) (CR)	Gets contents of XXXX through YYYY from SDK-85 Gets contents of XXXX through end of simulated memory from SDK-85 Gets entire contents of simulated memory from SDK-85
	RUn	XXXX (CR) (CR)	Orders SDK-85 to execute program starting at location XXXX Orders SDK-85 to execute program defined in simulated memory
	REset	(CR)	Orders SDK-85 to enter its system monitor
DISPLAY & MODIFICATION COMMANDS	EXam	XXXX - YYYY (CR) XXXX (CR) (CR)	Displays contents of XXXX through YYYY on screen Displays contents of XXXX through end of simulated memory on screen Displays entire contents of simulated memory on screen
	PRint	XXXX - YYYY (CR) XXXX (CR) (CR)	Prints contents of XXXX through YYYY on printer Prints contents of XXXX through end of simulated memory on printer Prints entire contents of simulated memory on printer
	SUbstitute	XXXX / DD (CR)	Substitutes the content of XXXX with hex value DD
	INsert	XXXX / D (CR)	Inserts capacity for D (0-9) bytes starting at address XXXX
	ERase	XXXX / D (CR)	Erases D (0-9) bytes starting at address XXXX
	MOve	ZZZZ = XXXX - YYYY (CR)	Moves contents of XXXX through YYYY to locations starting at ZZZZ
	SEe / SEt	(CR)	Displays current range of simulated memory / Sets new range
FILE MAINTENANCE COMMANDS	CReate	(CR)	Creates new file name in user file directory
	SAve	FILENAME (CR)	Saves current buffer contents to disk under specified file name
	LOad	FILENAME (CR)	Loads specified file from disk to buffer
	CHain	FILENAME (CR)	Chains specified file with current file in buffer
	QUit	(CR)	Exits Extended Monitor

********** NOTE: (1) XXXX, YYYY, & ZZZZ REPRESENT HEXADECIMAL ADDRESSES (2) CR - CARRIAGE RETURN

FIGURE 5.2 Command Summary for Extended Monitor strings, (eg. data communication and display commands) the specification field is optional. On the other hand, most of the modification and file maintenance commands, require user specifications. The specification field may contain up to 3 address operands, as in the MOVE command. Like the field separator, any non-alphanumeric characters can be used to separate operands.

Details of each command syntax and the requirements of the specification field are described in the following command routines, and are listed in Figure 5.2.

5.3 Main Program Structure

Whenever entering the Extended Monitor from the System Executive program or the Assembler, the object code file on track 36 is always loaded to the buffer before execution starts. In this way, the Extended Monitor may work as a Loader of the cross assembling system.

The main program structure is shown in Figure 5.3. Before accepting any command via keyboard, three procedures are processed. First, ST and DN are defined by the first four bytes of the current buffer; second, the user-defined binary file directory is loaded from disk into the last page of available RAM (5F00-5FFF) and is restored as a BASIC string array; third, the command array is defined for recognition of keyboard entries.

After the syntax field of the user input string is isolated from the specification field, the command recognition logic takes the leftmost two characters as a substring and performs comparisons with the command array. The execution logic will proceed toward the

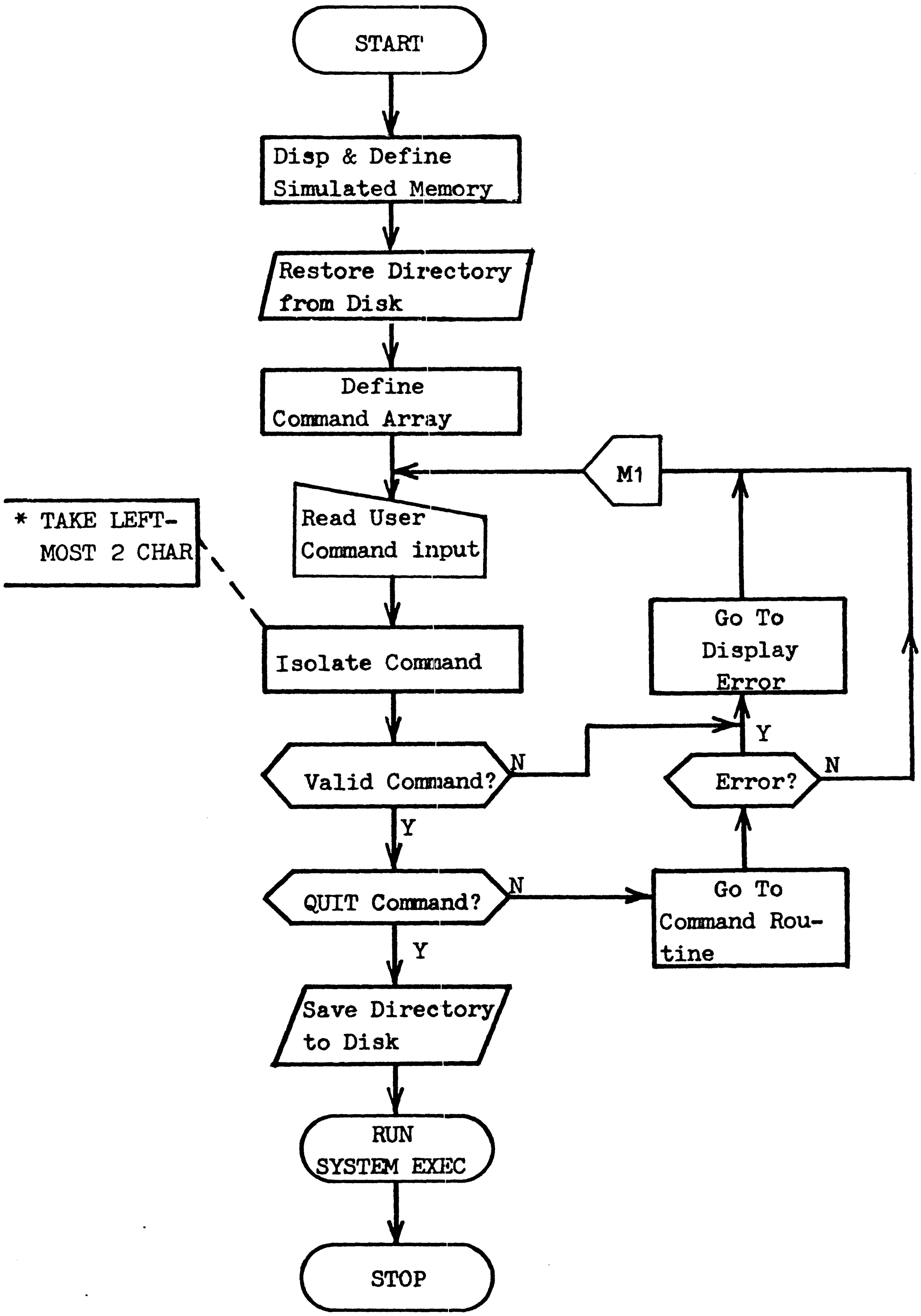

FIGURE 5.3 Main Program Structure of Extended Monitor

corresponding command routine, if a command is confirmed. Otherwise a syntax error message will be sent, and execution logic will accept a new user input.

Further scanning on the command string is performed by each command routine, when it is necessary. As described in greater detail in later sections, two scanning subroutines have been developed. PARSE is a subroutine which handles those commands with default options. If the address is not specified, PARSE designates the default condition. Otherwise, PARSE converts the entered string characters to the proper address value(s). SCAN is a subroutine called by those command routines which have no provision for default.

The only command which causes the Extended Monitor program to be terminated, is the command QUIT (abbreviated QU). This command orders the execution logic to save the current binary file directory on disk, and clears the Extended Monitor program from BASIC workspace by transferring control to the System Executive program.

5.4 Data Communication Command Routines

The most important function that the Extended Monitor provides is the ability to communicate with the SDK-85 motherboard. Each of the data communication command routines sets up the necessary information, then transfers control to a common routine, called LINK. LINK calls the specified assembly language subroutine which implements the command function by interacting with the SDK-85. The assembly language subroutines are described in section 3.4.

5.4.1 DUMP Routine

DUMP is a BASIC routine which operates with the assembly language subroutine TRANSM, to transfer a block of data in the simulated memory to the SDK-85. DUMP functions as a Loader for the Assembler.

The user may or may not enter address specifications following the command field. If an address specification is issued, then the starting address must be included. The ending address may be omitted. The subroutine PARSE will replace the excluded address with the corresponding default address value.

After the DUMP routine collects the necessary information, the execution logic will be routed to the routine LINK, in order to associate the assembly language subroutine, TRANSM, with the BASIC DUMP routine. If a transmission error occurs, unlike other error procedures, the execution logic may be ordered to retransmit the data block at the user's request. For this reason, the specified variable values remain valid, after the DUMP command is executed, until they are redefined.

Figure 5.4 depicts the flowchart of this routine.

5.4.2 GET Routine

The program logic of the GET command routine is very similar to the DUMP routine described in the previous subsection. However, the purpose of this routine is to get a block of data from the SDK-85 and to allocate that data to the corresponding locations in the buffer area. The GET function may be seen as the inverse of the DUMP function. Figure 5.5 presents the execution flowchart for this

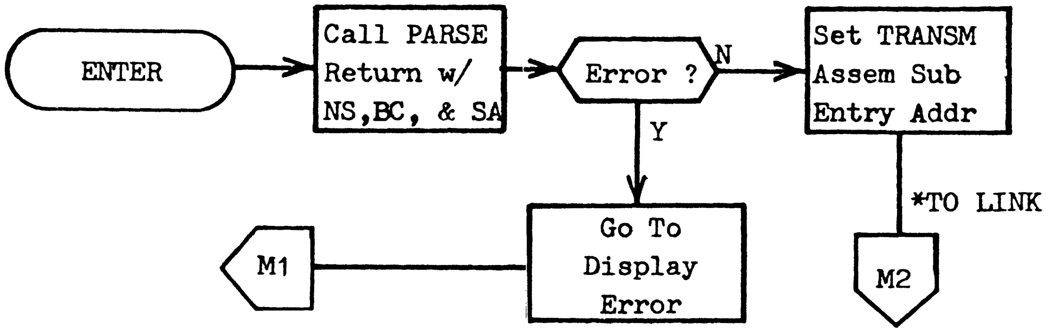

FIGURE 5.4 Flowchart for Routine DUMP

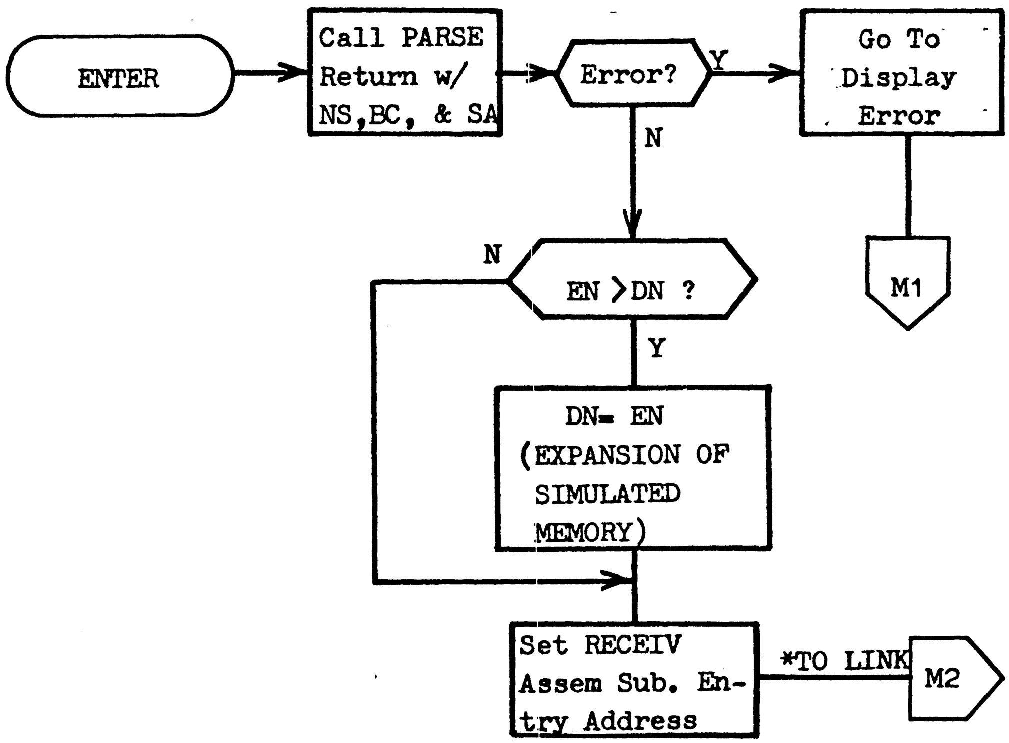

FIGURE 5.5 Flowchart for Routine GET

routine.

The BASIC variable, ST, and its associated hexadecimal value in the first two bytes of buffer RAM are initialized prior to user command entry. These values define the start of a 2K block of simulated SDK-85 memory. The address specifications of the GET command can not alter ST or its hexadecimal equivalent. This means that GET can only operate within the 2K buffer boundary. If the starting and ending addresses designated in the GET instruction, fall within this 2K range, then the corresponding SDK-85 data is loaded into the buffer displaced, if necessary, from the start of the buffer. To incorporate this additional data, the end of data record must be indicated.

Thus, if the value of the last address specification (EN) is greater than the current ending address (DN) of the simulated SDK-85 memory, then the value of EN replaces DN and the hexadecimal value of DN in bytes 3 & 4 of the buffer are likewise converted.

5.4.3 RUN Routine

This command routine can be used to order the SDK-85 to execute any specified program residing in the memory of the SDK-85. The user may or may not give the starting location of that program. If there is no address field following the syntax field, NS will default to the current starting address (ST) of simulated SDK-85 memory.

As cautioned in Chapter 3, if there is no RET instruction at the end of the 8085 program or the SDK-85 program itself is terminated in an infinite looping structure, then the OSI-C4PMF system loses control of the SDK-85. In this case, a manual reset and initialization on the SDK-85 is necessary if the communication channel is to be restored.

The program sequence of this routine is reproduced in Figure 5.6.

```
700 REM  RUN Command Routine Entry
710 GOSUB 20100 : REM  Call GETNS
715 ON CHK GOTO 30000, 30050, 30100, 30300 : REM  Check error
718 IF J-(K+3)<>0 GOTO 30000 : REM  Extra specification
720 LO=71 : REM  Set assembly subroutine RUN entry address
725 GOTO 11500 : REM  Go to LINK routine
```

FIGURE 5.6 Execution Sequence of Routine RUN

5.4.4 RESET Routine

This command performs a soft-reset function on the SDK-85. In other words, the OSI-C4PMF releases its control of the SDK-85 and lets the SDK-85 ROM monitor program take over.

Unlike other commands, there should be no address following the syntax field. Figure 5.7 duplicates the program procedures of the RUN routine.

```
750 REM  RUN Command Routine Entry
760 LO=92 : REM  Set assembly subroutine RESET entry address
765 GOTO 11640 : REM  Go to LINK
```

FIGURE 5.7 Execution Sequence of Routine RESET

5.4.5 LINK Routine

Unlike the previous routines, this routine is not a direct command procedure. It is used to link all data communication commands and the corresponding assembly language subroutines

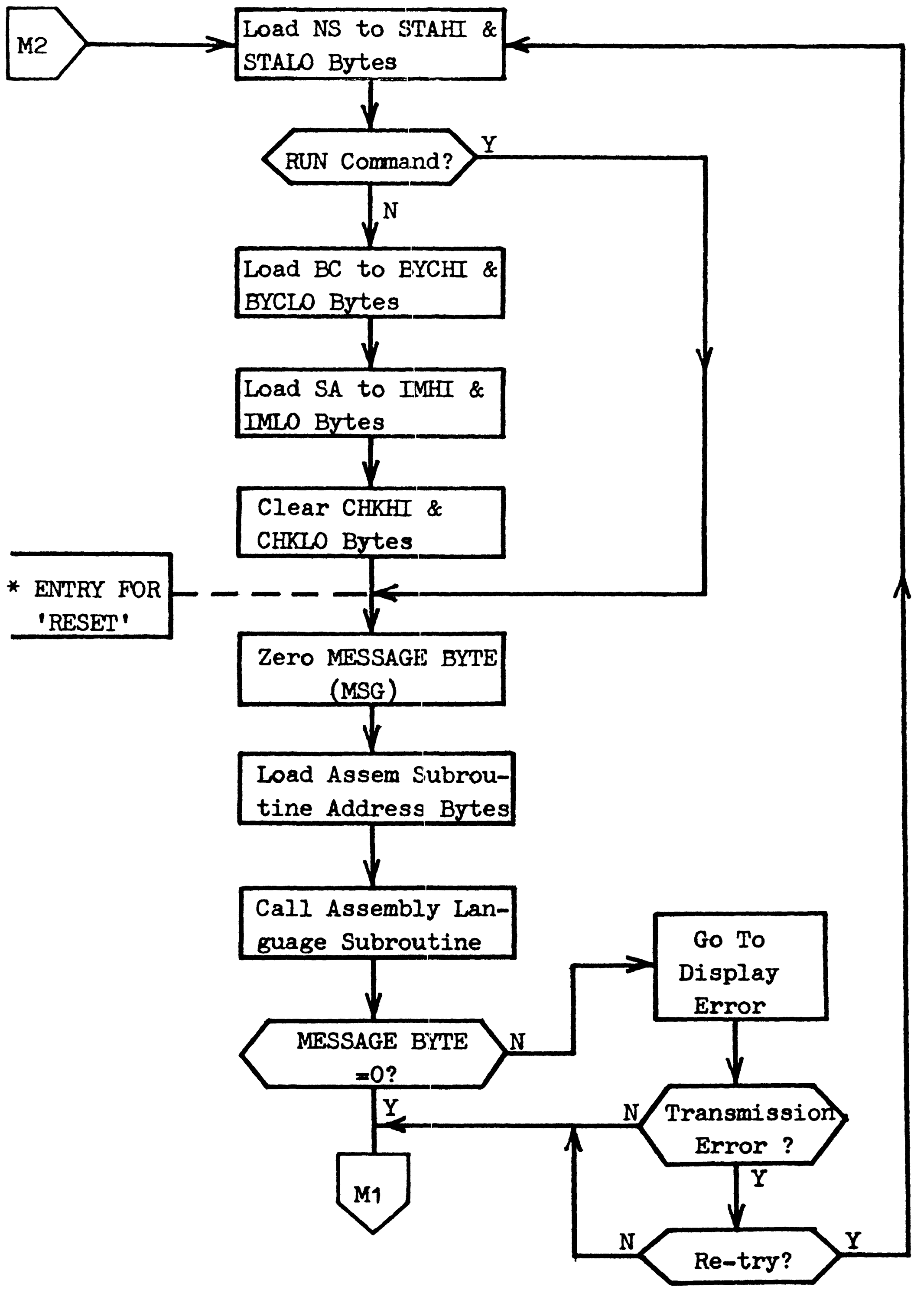

FIGURE 5.8 Flowchart for Routine LINK

together. The aforementioned command routines set up the necessary address values. Then LINK is entered to allocate those values to the appropriate memory locations before calling the assembly language subroutines. LINK also checks communication error status by examining the message byte, after returning from the assembly language subroutine.

Figure 5.8 presents the algorithm of this routine. As may be noted, the RESET entry is different than the entry location of other commands.

5.5 Display & Modification Command Routines

Seven commands are classified in this family. They are EXAM, PRINT, SUBSTITUTE, INSERT, ERASE, MOVE, and SEE/SET. The common characteristic of these commands is that they can be used to display/print the contents of the simulated SDK-85 memory, or modify the layout of the current buffer.

5.5.1 EXAM and PRINT Routines

Although EXAM and PRINT are two independent commands, they share the same procedures to perform the displaying task. The EXAM command allows the user to examine a block of data on the screen, and the PRINT command prints the data on the serial printer. However, the PRINT command has an extra feature which the EXAM command does not. This is the ability of allowing the user to add a title line before data printout. Both commands use the same displaying form, an example of which is shown in Figure 5.9.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0010           AC CD 00 2B 30 49 FF DE C6 E8 20 12 AD
0020 01 39 BD B2 F0 4F EA 60 7F 03 56 8A 9D CD 37 FA
0030 DE BD F6
```

FIGURE 5.9 An Example for Displaying Form

Like the data communication commands, the user may or may not specify the first and last displaying addresses. The subroutine PARSE is again used here to return the appropriate starting address and the byte-count, or error code.

The subroutine DISPLAY is called to exhibit data on the screen or printer. DISPLAY collects 16 bytes of data in a string, and sends the string to either the screen or printer by checking a display flag. As illustrated in the example of Figure 5.9, the first row indicates the least significant digit of the hexadecimal address. These digits, 0 to F, form the columns of a matrix. The matrix rows begin with an address value which is a multiple of sixteen. The data dump is accomplished by displaying blanks until the data starting address is hit.

Upon returning from the subroutine DISPLAY, the user may request the execution logic to display the next 256 bytes of data by simply typing "Y" when interrogated by the OSI-C4PMF.

Figure 5.10 explains the algorithm in flowchart form.

5.5.2 SUBSTITUTE Routine

This function allows the user to change the contents of the buffer area.

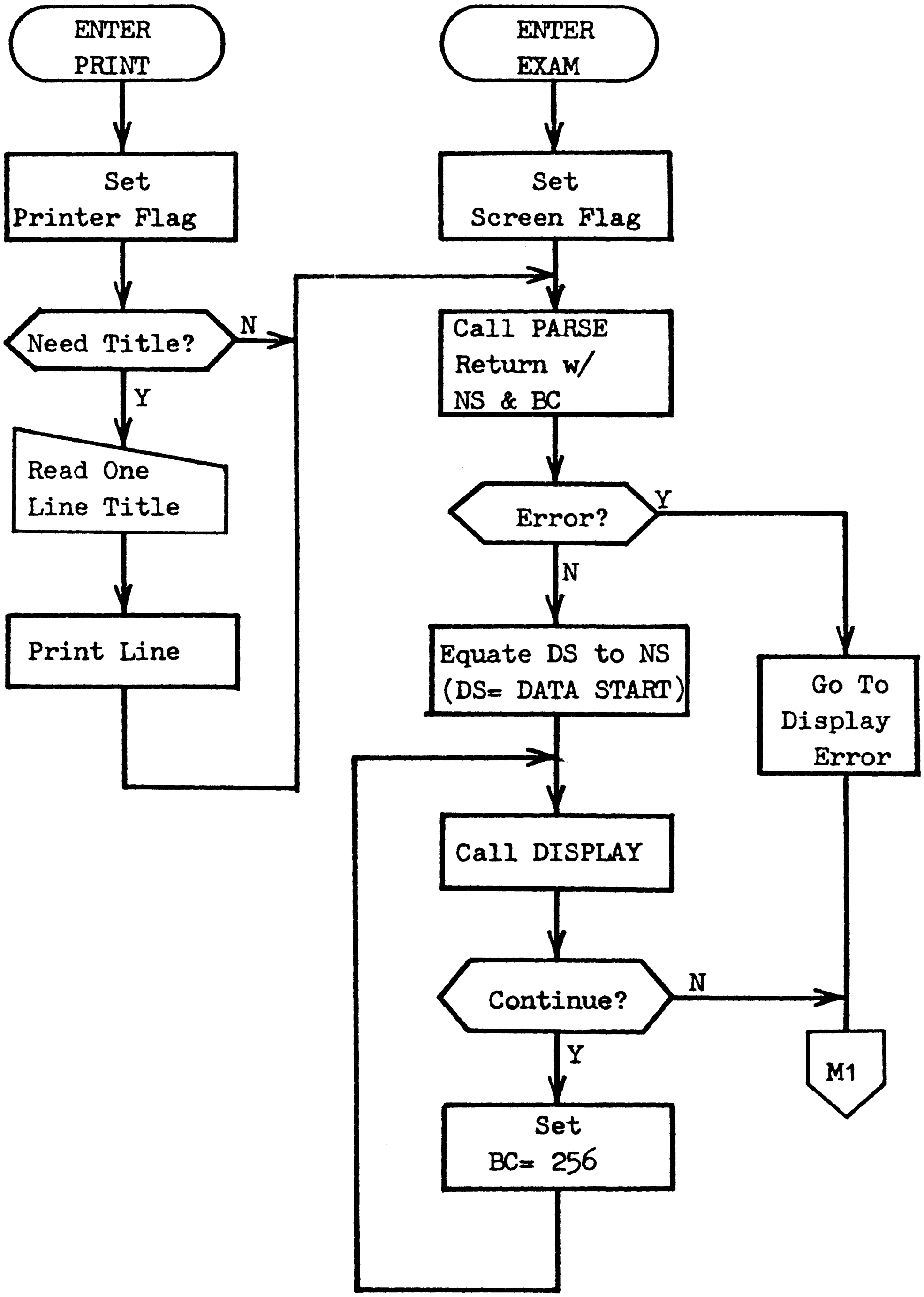

FIGURE 5.10 Flowchart for Routine EXAM and PRINT

Once the user specifies the location and the new contents to be entered, the subroutine SCAN is called to check if there are any errors on the entered values. After the task of changing is performed, the program logic compares the address of the altered byte with the current ending address value of simulated SDK-85 memory. If the changed location exceeds the current end of simulated SDK-85 memory, the pseudo SDK-85 memory is expanded to include that byte. This performs an automatic change & increment function for convenient buffer operation.

The routine is designed so that the user may change the contents of the next buffer location by simply entering the new data value in hexadecimal when prompted by the execution logic. This sequence of events continues until the bottom of the buffer is reached or any non-hex digit is entered.

Figure 5.11 shows the flowchart for this operation.

5.5.3 INSERT Routine

The flowchart of this command routine is presented in Figure 5.12. The starting address where the data is to be inserted and a single decimal digit which indicates the number of inserted bytes, must be provided by the user. An error message is generated if this insertion would increase the size of the buffer over the 2K capacity or if the number of bytes is greater than nine. Therefore, this function allows the user to insert a maximum of nine bytes.

The actual action taken by the execution logic is to move the data block which follows the insertion point down D bytes. D is a

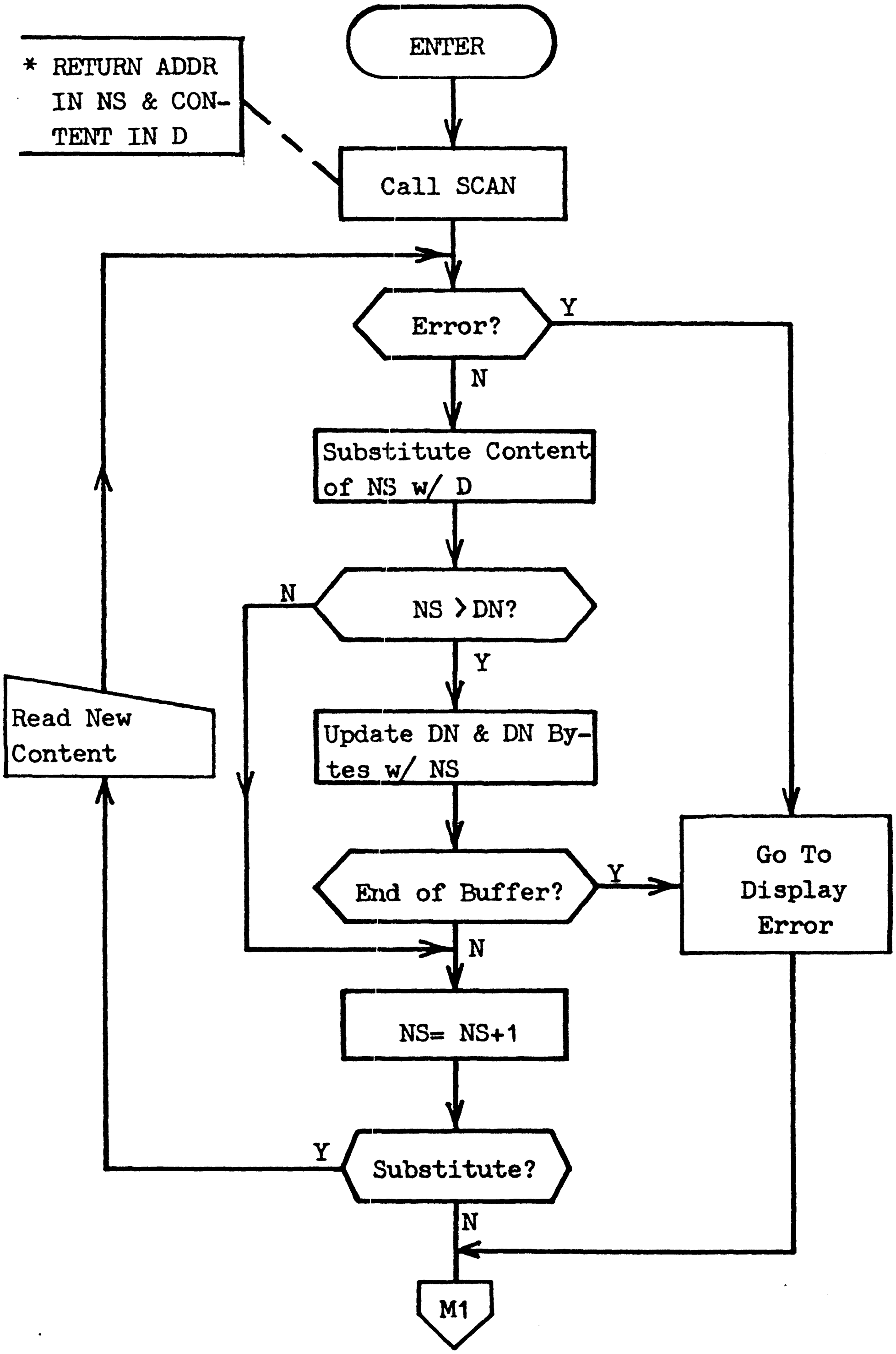

FIGURE 5.11 Flowchart for Routine SUBSTITUTE

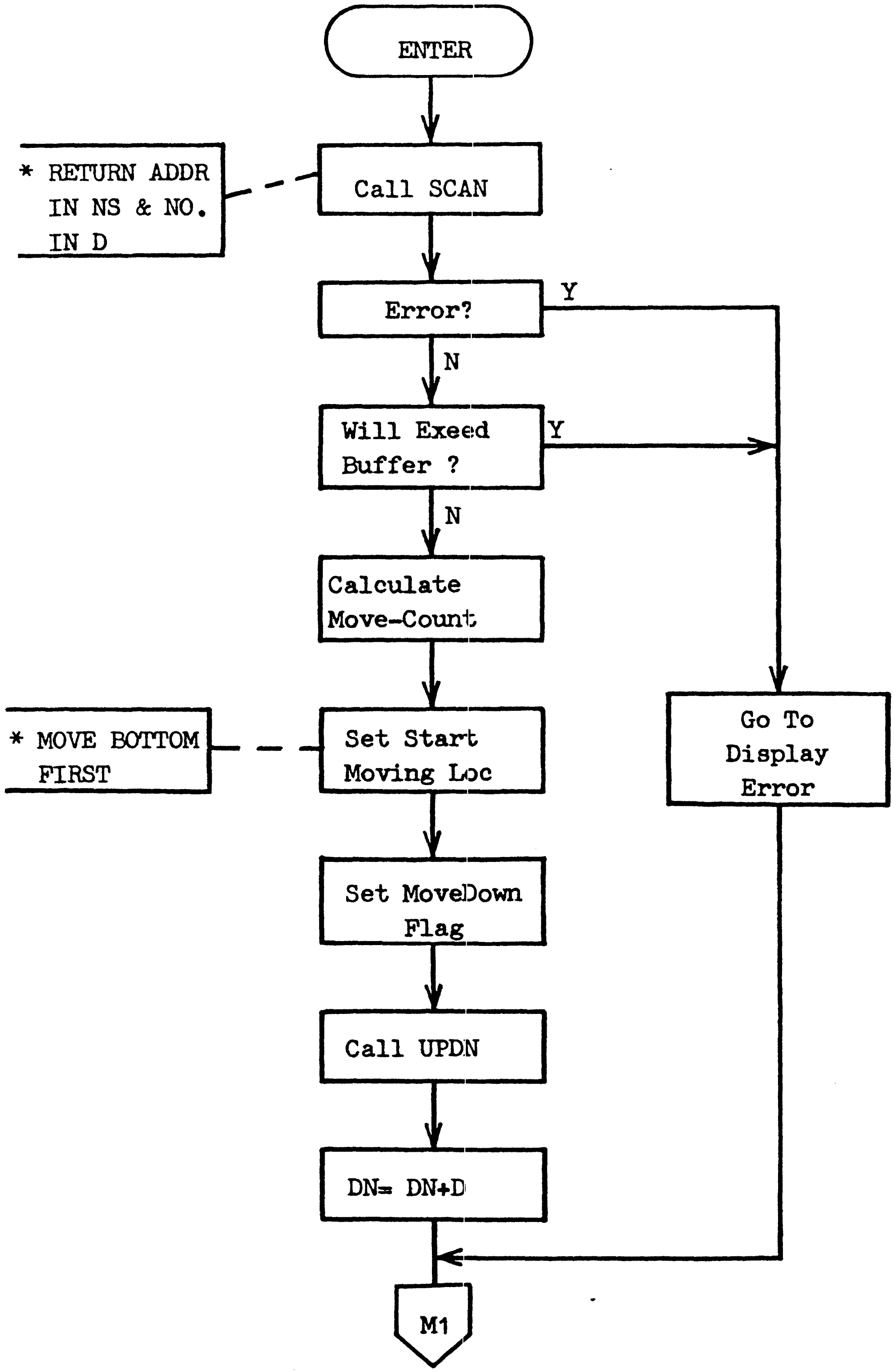

FIGURE 5.12 Flowchart for Routine INSERT

variable in the range of 0 to 9. As noted in Figure 5.12, the execution logic sets a flag and then calls a subroutine UPDN to perform the block move task. Moving the bottom of the block first prevents loss of data due to overwriting. The ending address of simulated SDK-85 memory is extended to appropriate new location.

It should be noted that the contents of the locations where the user intends to make insertions remains unchanged. The user must use the SUBSTITUTE command, which is described in the previous subsection, to enter new data to those locations.

5.5.4 ERASE Routine

This function allows the user to erase a number of bytes from simulated SDK-85 memory. The number of addresses cannot be greater than nine, and the starting location must be valid in the current buffer range. Otherwise the execution logic will refuse to perform this operation, and an error message will be generated.

Unlike the INSERT command, the program sets an UP flag before calling the UPDN subroutine. The address of the first byte to be moved is set to the location just beyond the last byte to be erased. Then UPDN moves the data block, starting at the first address to be moved through the end of simulated memory. The data block is in this way, shifted up D locations. As for INSERT, D is the variable containing the number of bytes to be erased. Before this routine is terminated, the ending address of the memory image is updated with the result of DN minus D bytes.

A generalized flowchart for this routine may be reviewed in

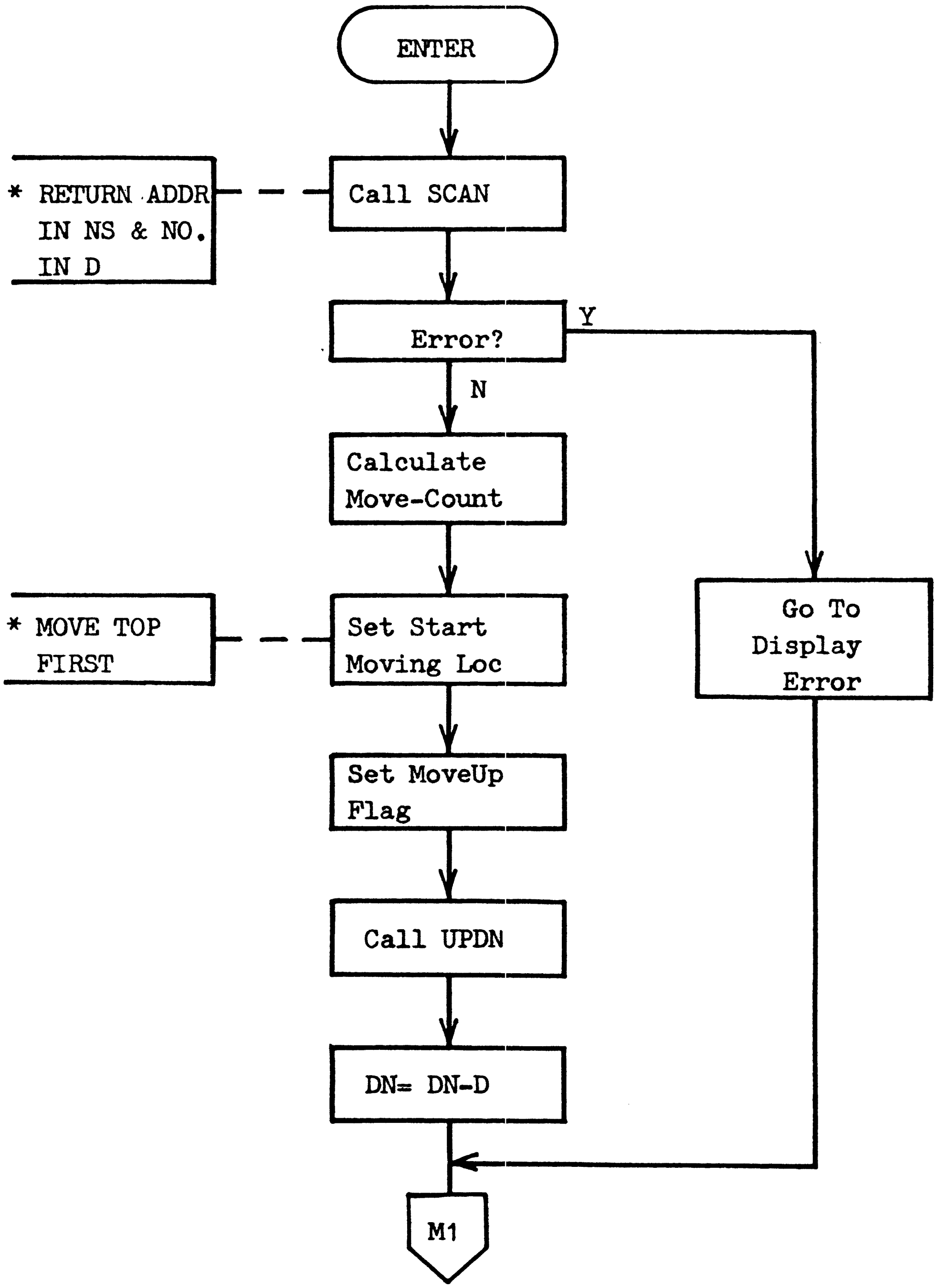

FIGURE 5.13 Flowchart for Routine ERASE

Figure 5.13.

5.5.5 MOVE Routine

To perform this function which can relocate a data block anywhere in the buffer area, three address operands must be provided in the specification field. The first is the destination starting location of the data block. The next two operands represent the source starting and ending address of that data block respectively.

Figure 5.14 shows the flowchart of this routine. Upon entering this routine, the subroutine GETNS is called to isolate the first operand and return the destination starting address in the BASIC variable NS. Since GETNS is then used to fetch the starting address of the data block, it is necessary to equate MS to NS. GETNS is called by the subroutine SCAN which reads the next two operands, and returns the source starting and ending addresses in NS and EN.

The execution logic examines these three address values to determine the direction of movement. If the function desired to move down, the program logic will also determine the end of the data block to prevent over-expansion (2K maximum). Like INSERT and ERASE, the UPDN subroutine is employed to perform data block movement.

In the case of downward data block movement, the ending address of simulated SDK-85 memory is updated, if the data block move increases simulated memory size. In moving data upward, the size of simulated memory generally remains the same. It may only be reduced if the user sets the source ending address equal to the current end of simulated SDK-85 memory.

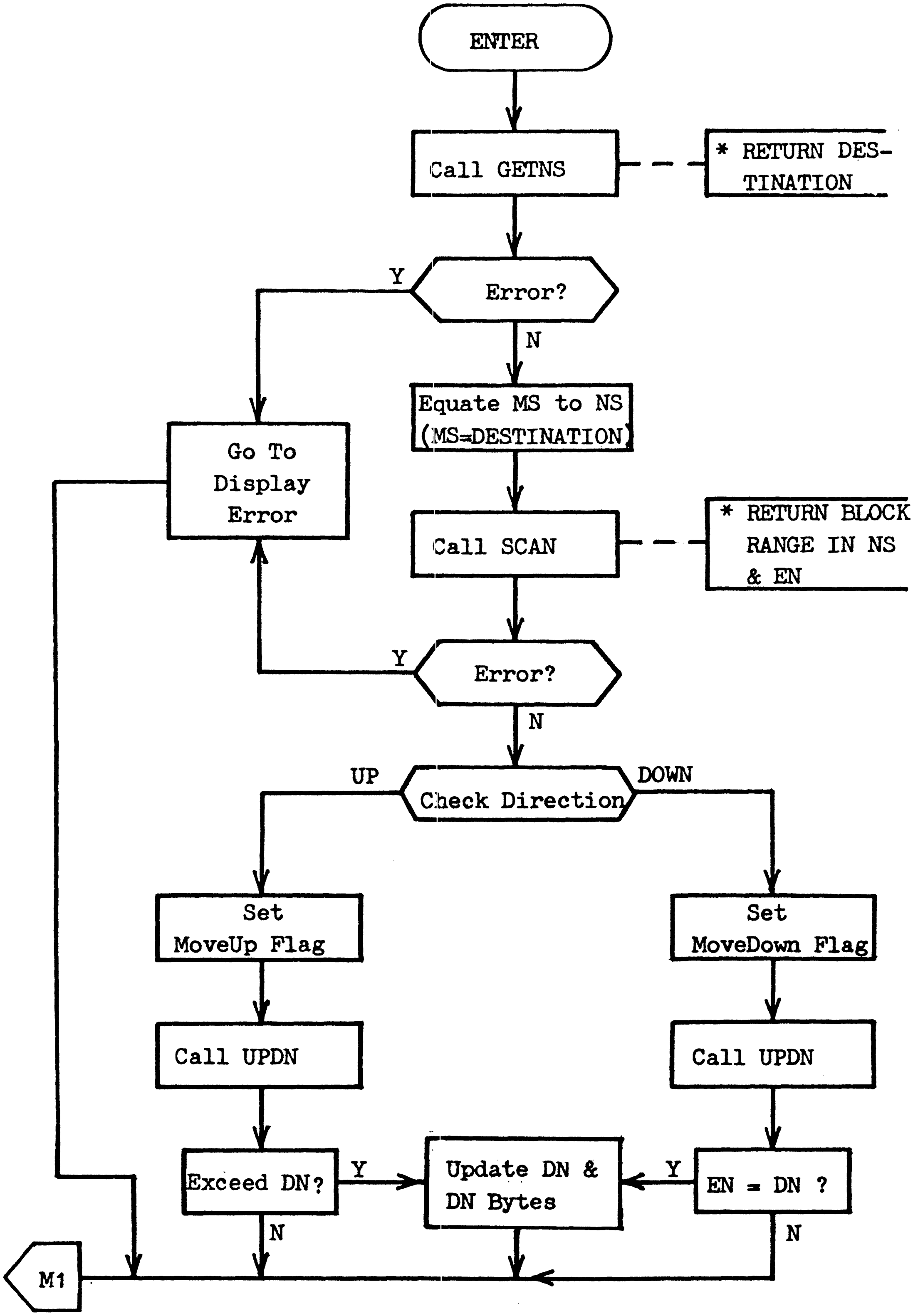

FIGURE 5.14 Flowchart for Routine MOVE

5.5.6 SEE/SET Routine

The SEE/SET function allows the user to examine or define the range of simulated SDK-85 memory. This function contrasts with the automatic ranging which occurs as a result of previously discussed commands. The SEE/SET command has no specification field. In this way, the user may view the current range of simulated memory without affecting the established limits. The user may set a new boundary under the direction of software logic. No change is made unless the user input is a hexadecimal address.

The flowchart of this routine is presented in Figure 5.15. As illustrated, the routine is begun by calling the subroutine SHOW to display the current limits, in hexadecimal, on the screen. The execution logic interrogates the user on whether to change the upper boundary. The user may enter a new address in four hexadecimal digits or may simply enter an "N" to escape this change. The lower boundary procedure operates in a similar manner. Again, the user may enter a new address or may avoid change by typing "N". Next, the error detection procedure begins. If the simulated SDK-85 addresses exceed a 2K range, or the ending address precedes the starting address, or if any invalid hexadecimal digit is entered, an error message is displayed.

It should be noted that the SEE/SET operation not only changes the decimal variables maintained in BASIC workspace, but also alters the corresponding hexadecimal bytes in the first four locations of the buffer.

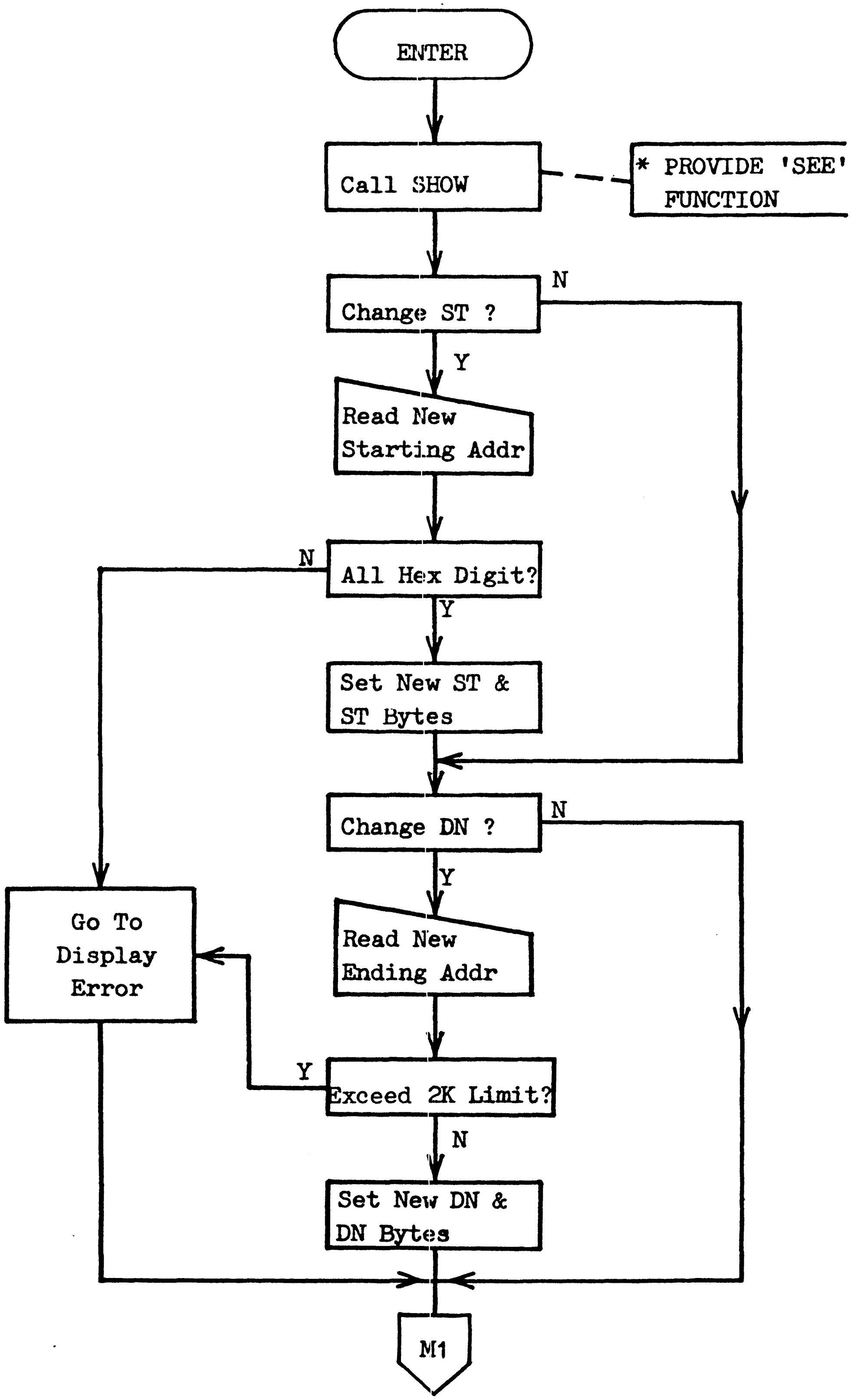

FIGURE 5.15 Flowchart for Routine SEE/SET

5.6 File Maintenance Command Routines

The Extended Monitor allows the user to manage five binary files. Track 31 to track 35 are reserved for these files. Each file occupies one track on the disk. A directory is maintained by the Extended Monitor program to provide records to file maintenance commands.

As mentioned before, the user file directory is recovered from sector 2 of track 39 when the Extended Monitor program initializes the system. The directory is composed of two arrays, F$(X) and P(X). F$(X) holds the file names of each track, and P(X) records the integer number of pages (sectors) occupied by the corresponding file. The directory may be updated by certain file maintenance commands, and is saved back to its disk location before exiting the Extended Monitor.

5.6.1 SAVE Routine

This command routine allows the user to save the current file in the buffer onto the disk with a defined file name in the specification field.

The routine starts by calling the subroutine GETFILE which checks the user input file name with the directory contents, and returns with a file index number in variable X. Only five tracks have been assigned for file storage. If the returned value in X is greater than 5, then the routine is terminated and an undefined file error message is displayed. Otherwise the subroutine CALCPAGE is

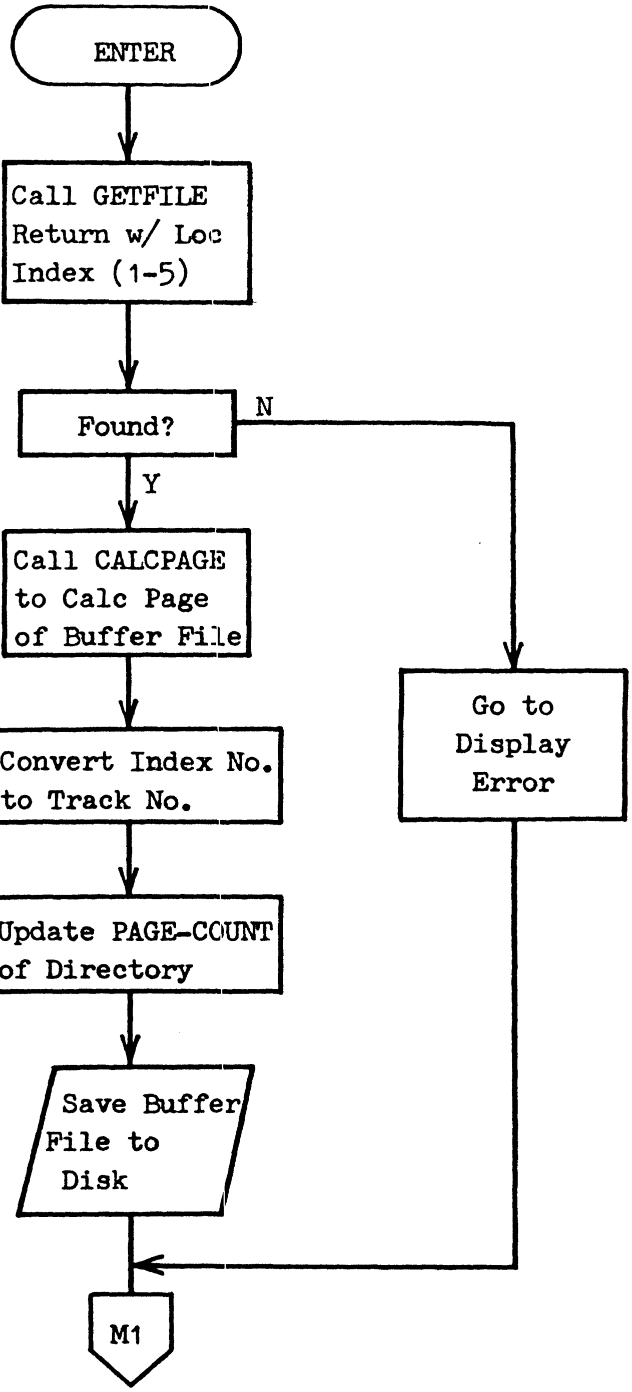

FIGURE 5.16 Flowchart for Routine SAVE

called for calculation of the page-count (P) of the current file in the buffer. Page-count determines the integer size of the file to be saved.

Since the five file tracks are located from tracks 31 to 35, the appropriate track position can be obtained by adding the index value to the base value 30. Before saving to disk, the corresponding page-count in the directory is updated with the value in P.

Figure 5.16 presents the flowchart for this command routine.

5.6.2 LOAD Routine

Retrieving a file from one of the file tracks and loading it into the buffer, is the purpose of the LOAD routine. As with SAVE, the user input file name must be defined prior to its designation in the specification field.

After the file name has been verified, the base value, 30, is added to the index number. This track number is converted to a string variable to be used in a DOS load statement of the BASIC routine. As depicted in Figure 5.17, the range of the simulated SDK-85 memory is redefined by the contents of the first four locations of the buffer. This is accomplished by calling the SHOW subroutine after loading. SHOW will also displaying the new simulated memory limits for the user's reference.

5.6.3 CHAIN Routine

The CHAIN routine was developed to combine two files into a single file space within the confines of the 2K buffer. In order to

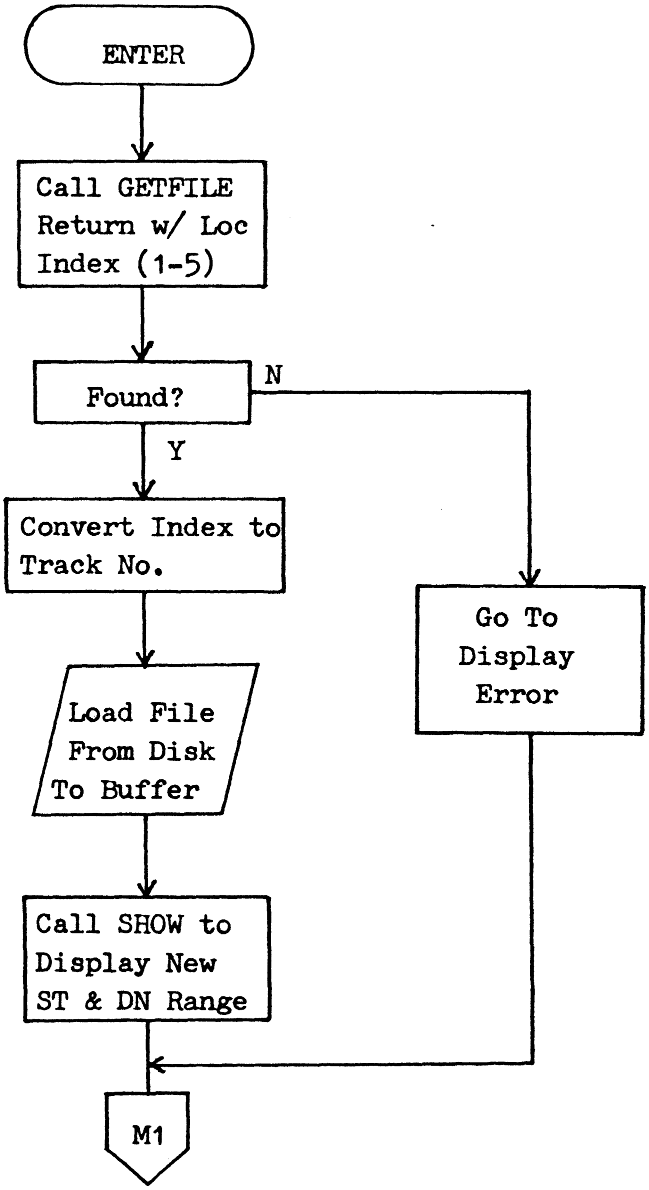

FIGURE 5.17 Flowchart for Routine LOAD

successively join two files, the file which is to come first must reside within the buffer before the CHAIN command is issued. The file described in the specification field is the remaining file.

As mentioned earlier, the user file directory is composed of a file name array F$(X) and a file page-count array P(X). The latter indicates the integer number of pages in each file. Since the size of the buffer is limited to eight pages, routine logic determines the total number of pages in the combined file, to prevent exceeding the lower limit of the buffer. This procedure, as shown in Figure 5.18, is implemented by adding the page-count of the current file in the buffer to the page-count of the disk file to be chained. The files are not joined if the sum is greater than 8 pages. The CHAIN operation transfers the disk file to the location following the end of the file residing in the buffer.

The ending address of simulated SDK-85 memory is increased to include the added file. The first four bytes of the added file are removed by a deleting process.

5.6.4 CREATE Routine

To create, rename, or check the filenames of the user directory are the purposes of this command routine. As for SEE/SET, no specification field is allowed. The routine logic instructs the user to enter filenames.

Figure 5.19 shows the execution sequence of this routine. As illustrated, the algorithm starts by displaying the current directory on the screen. The user must then confirm the intention to generate

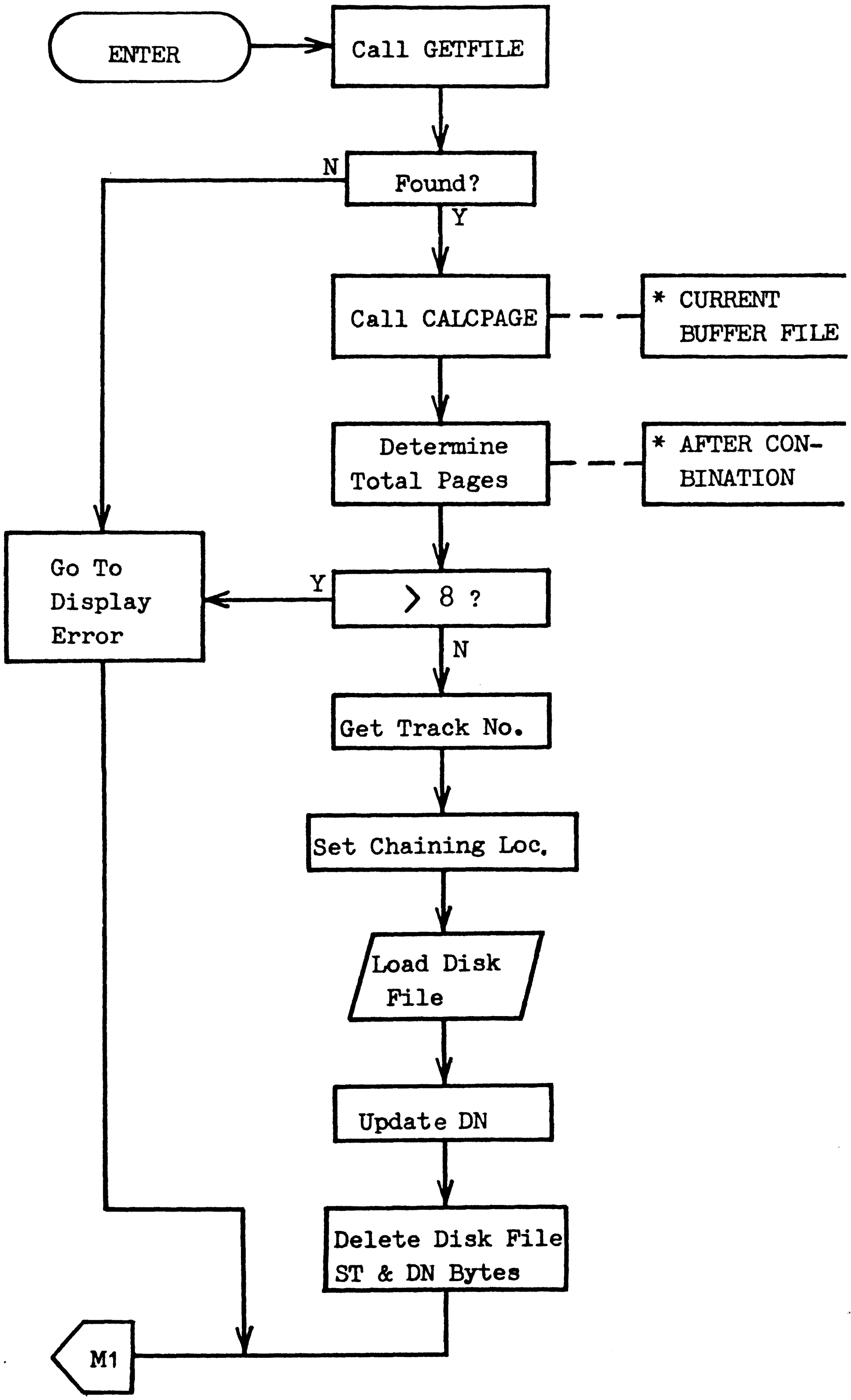

FIGURE 5.18 Flowchart for Routine CHAIN

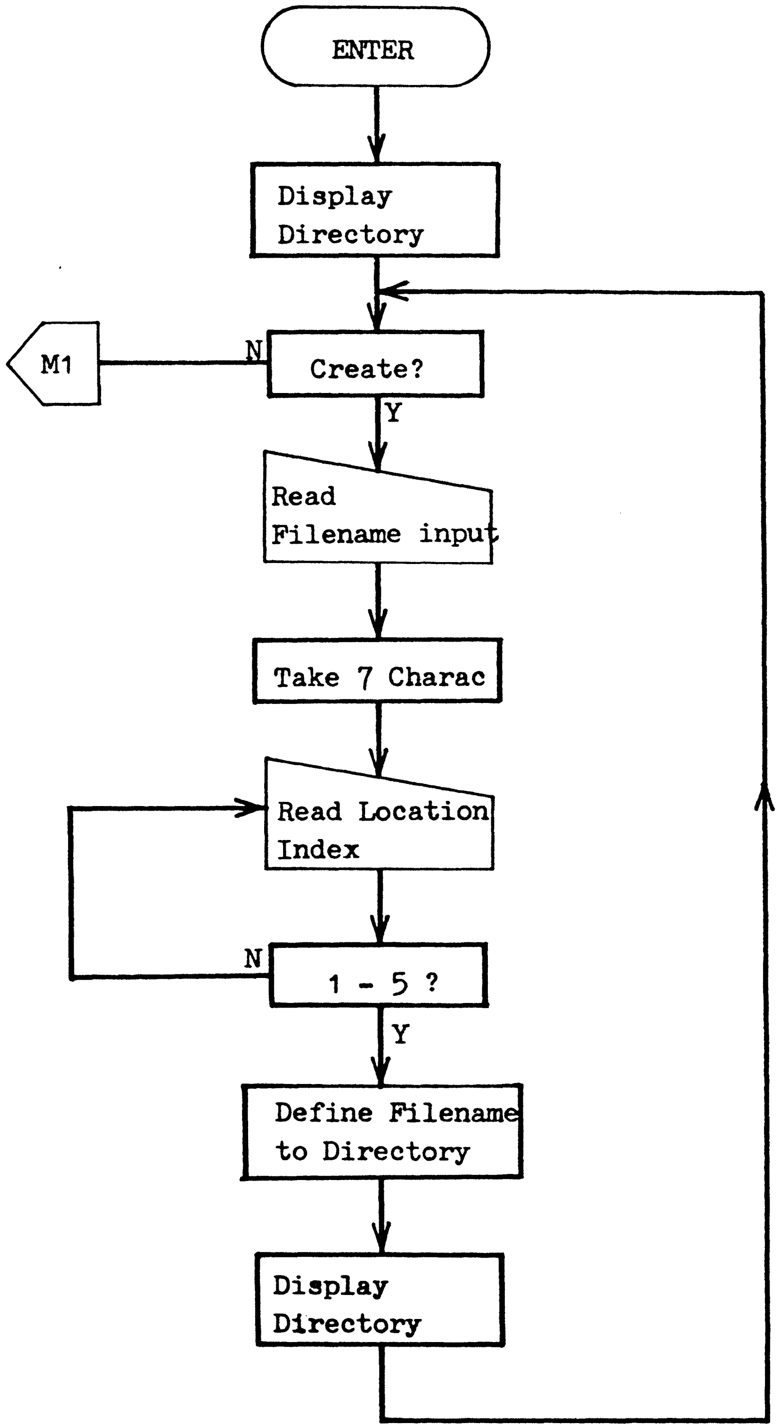

FIGURE 5.19 Flowchart for Routine CREATE

a new filename. Otherwise the routine will be terminated. This gives user an opportunity to simply review the directory without changing it.

The filename creation procedures may be divided into three parts. First, the first 7 characters are read from the user console as a filename. Second, the user is asked to enter the location index (1-5). Third, the entered filename is allocated to the array position pointed to by the location index.

After these steps are completed, the updated directory is displayed on the screen. The user may create another filename or exit this routine when the routine raises the question on the screen.

5.7 Subroutines

The execution procedures of major subroutines are explained in flowchart form in the next few pages. From Figure 5.20 to Figure 5.25, the following subroutines are depicted:

PARSE - Interprets the specification field or defines default value(s)

SCAN - Reads the specification field without assigning default value

DISPLAY - Exhibits data block from NS through EN on screen or printer

SHOW - Defines ST & DN from the first four buffer bytes and displays their hexadecimal values

GETFILE - Gets the designated file location index

CALCPAGE - Calculates the integer number of pages that the file in the buffer occupies

Those subroutines which are not listed above can be reviewed in the Extended Monitor program listing in the Appendix.

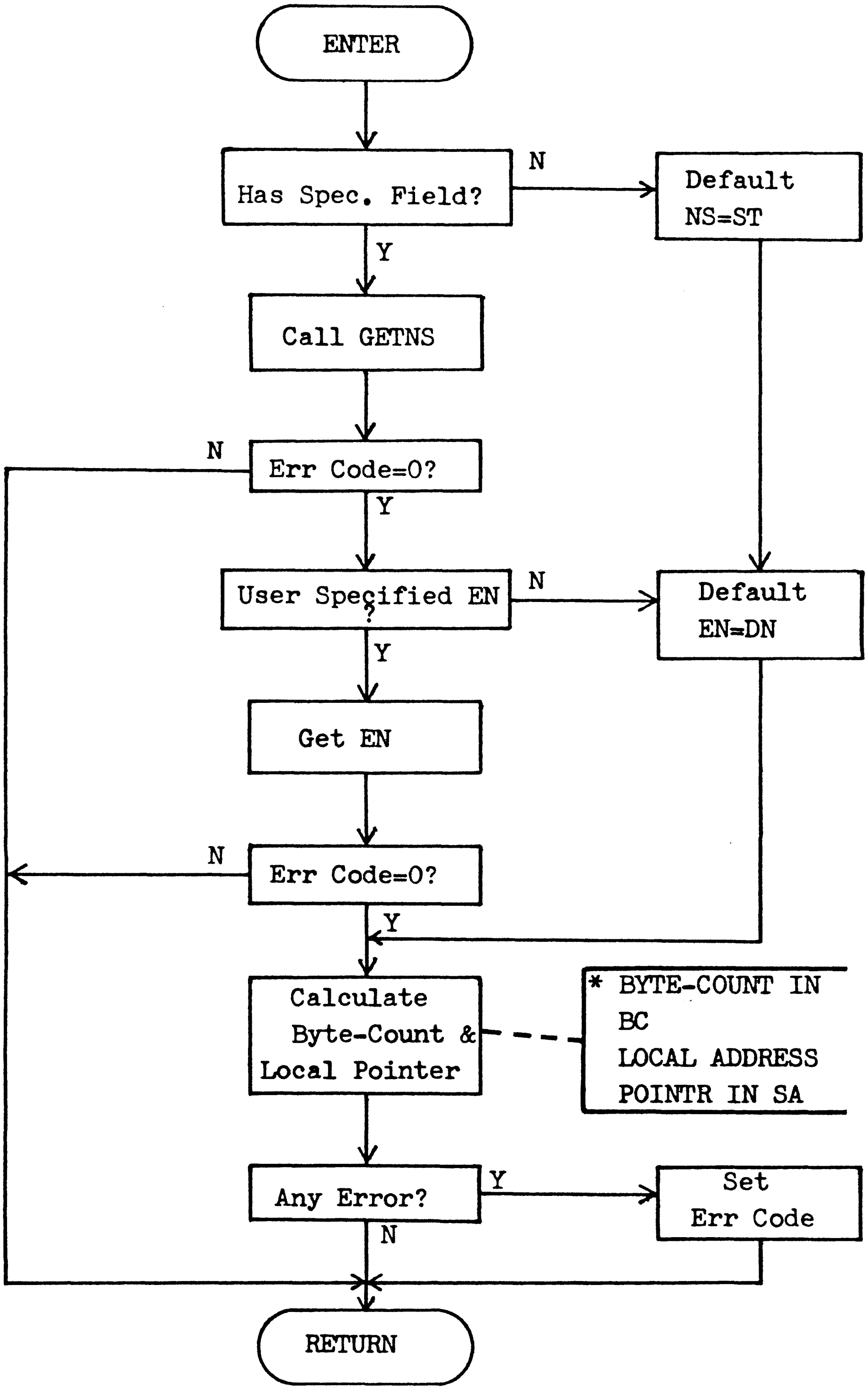

FIGURE 5.20 Flowchart for Subroutine PARSE

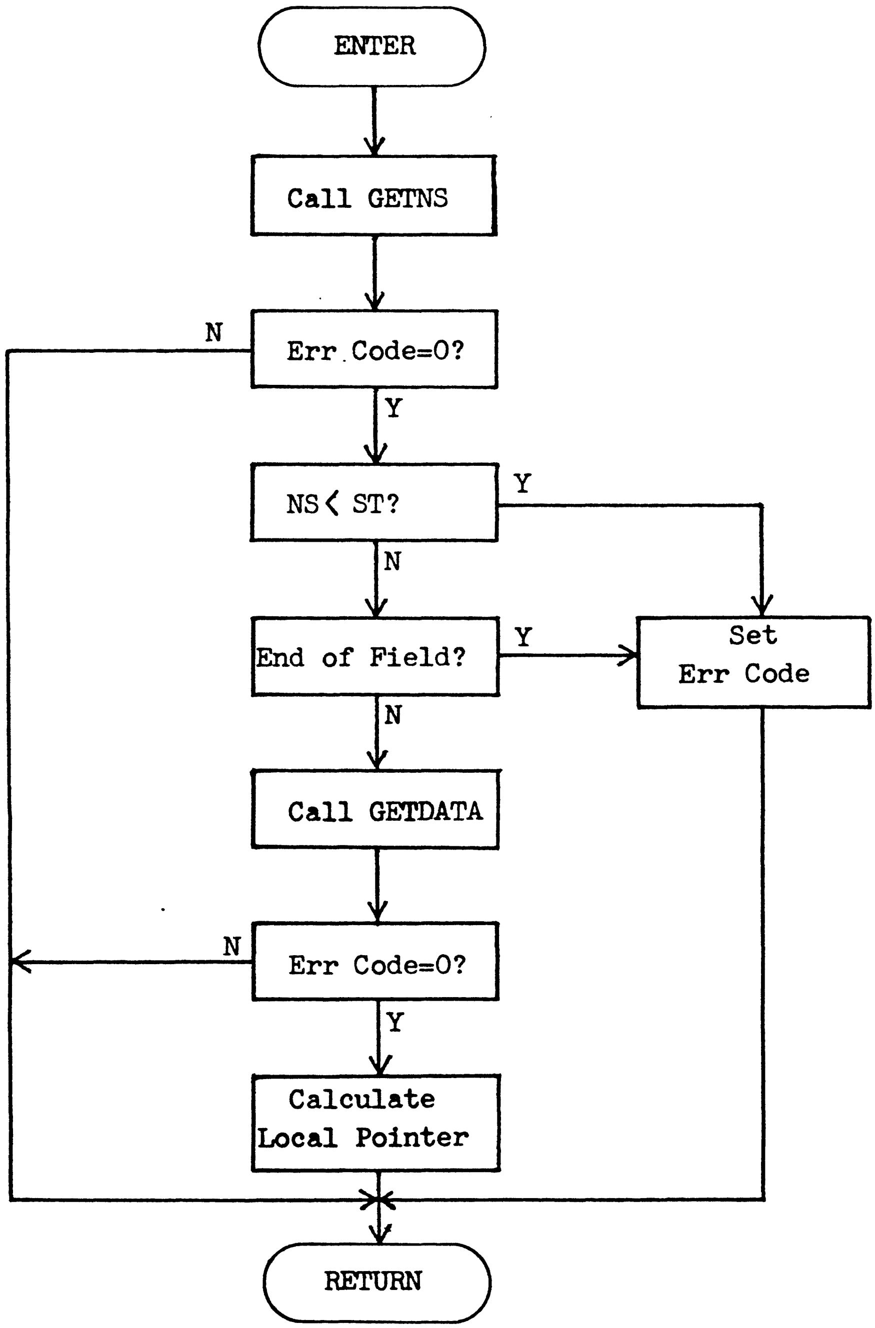

FIGURE 5.21 Flowchart for Subroutine SCAN

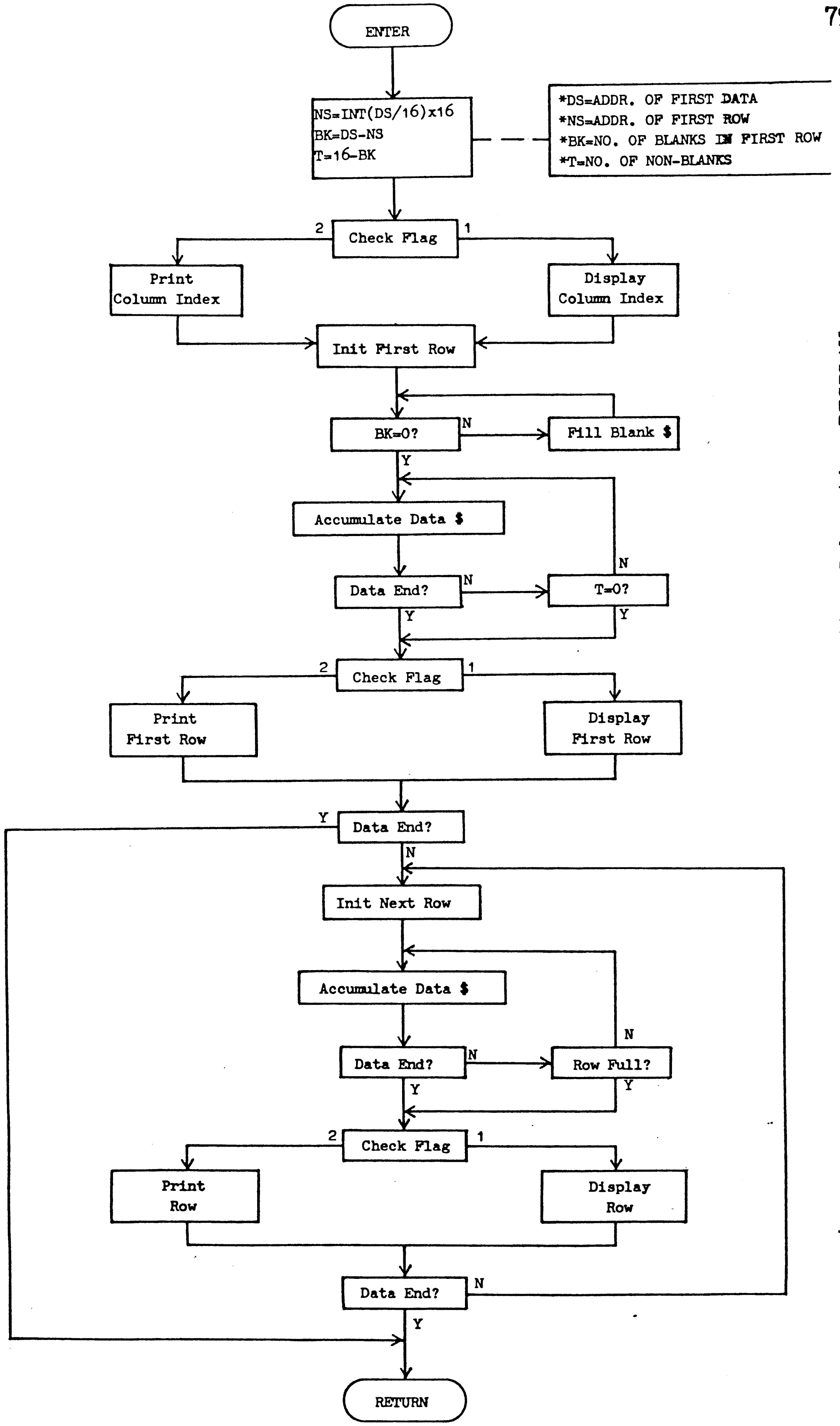

FIGURE 5.22 Flowchart for Extended Monitor Subroutine DISPLAY

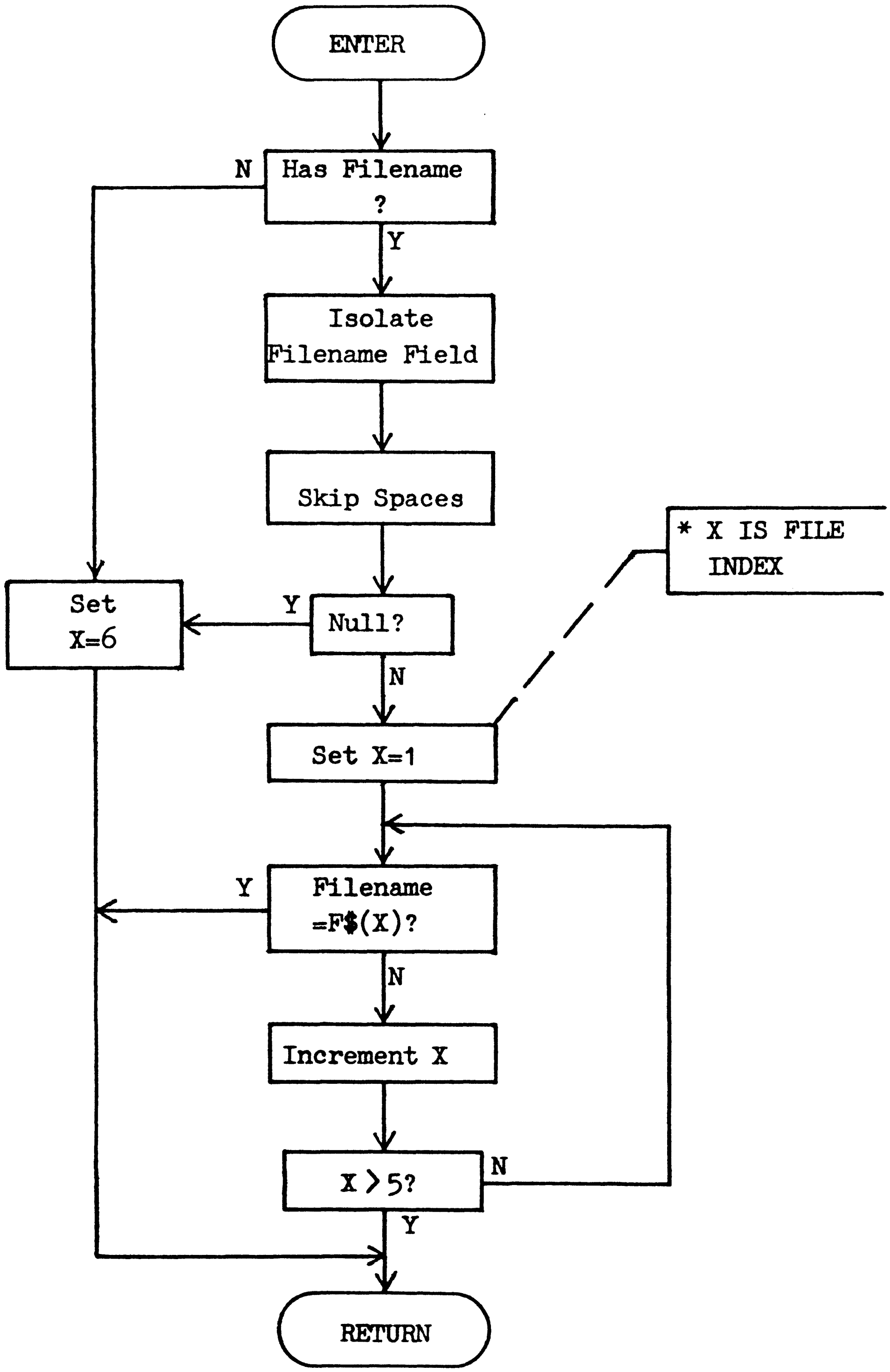

FIGURE 5.23 Flowchart for Subroutine GETFILE

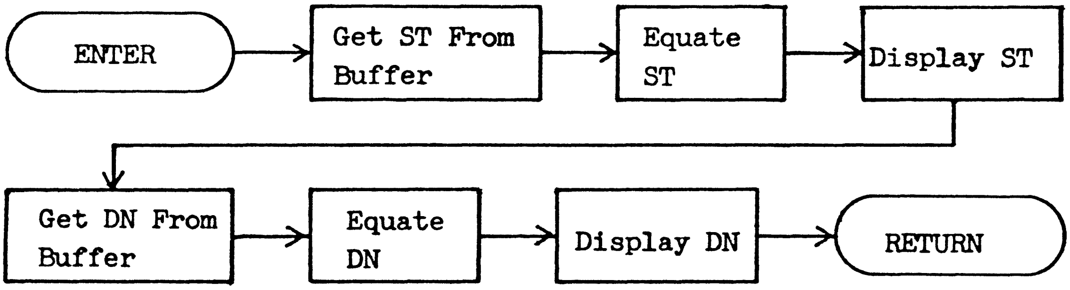

FIGURE 5.24 Flowchart for Subroutine SHOW

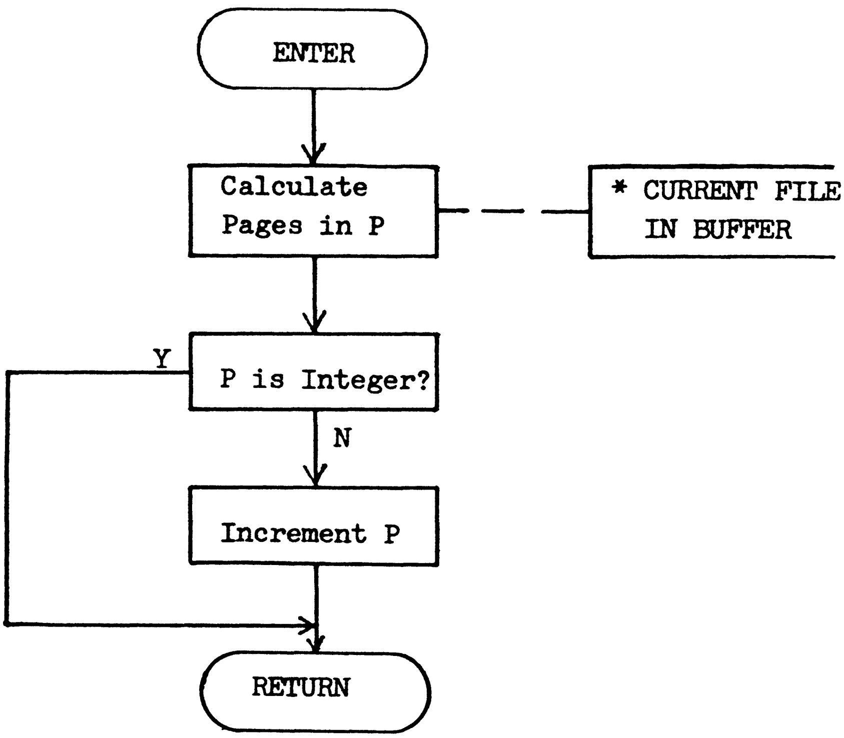

FIGURE 5.25 Flowchart for Subroutine CALCPAGE

CHAPTER 6 TEXT EDITOR

6.1 General Description

This Editor is developed for the purpose of editing the text file of the assembly language source program. It is written in BASIC language, and stored on disk under the file name, EDIT. It is loaded into BASIC workspace by the proper menu selection in the System Executive program or the error exit of the Assembler.

A 2K buffer is protected by limiting the lowest BASIC workspace to hexadecimal location 57FF. This buffer is used as the source file I/O transition area for saving to or loading from disk. Due to the restriction of limited memory, the maximum file capacity at a time maintained in the workspace by the Editor is 4K characters (bytes) or 280 source lines. Four tracks are available to accommodate the source files. Every two tracks contain a total of 4K bytes. Therefore, one file may occupy two tracks, and the Editor may manage two files. One is called First file. Another one is called Extended file. The First file uses disk tracks 37 and 38. The Extended file uses tracks 29 and 30. A file mode flag maintained by the Editor guides the disk accessing logic to either the First file tracks or the Extended file tracks. This flag defaults to flag the First file mode by the Editor initialization procedures, and may be varied by the proper commands. Although the Editor manages these two 4K-files as two independent files, lacking an END directive at the end of the First file will cause the Assembler to see the Extended file as an extension of the First file. This makes the Editor impose a maximum capacity upon the source file of 8K bytes or 560 source lines. It should be noted that the Extended file cannot be assembled individually.

Each of the entered source lines is maintained by a BASIC string array element. Every line must be started by a decimal line number. This line number is used as an index reference to locate the entered line to the proper array element position. Once a new line is entered, the program logic sorts all lines in sequence by comparing the line numbers. Therefore, no insertion command is needed. The use of line numbers is modeled after the BASIC programming language.

In order to store more characters in the limited memory space, every entered line is rearranged by a shrinking procedure before the input logic prompts the user to enter a new line. The shrinking procedure scans the entered line, and replaces the encountered multiple-space with one space character followed by a letter character (A-Z) as the repeat-count. For example, a source line is entered as below ('*' represents space):

```
10*******LDX*H,2000H
```

After completing the shrinking process, the appearance of this line is shown as below:

```
10*GLDX*AH,2000H
```

The letters G and A represent the repeat-counts for seven spaces and one space respectively. Therefore, the maximum allowed spaces between any non-space characters is limited to twenty six which is the total number of alphabetic letters. The displaying/printing commands recover each of the specified lines back to its original form without changing the shrunken form.

A string array variable, I$(X), is assigned to accommodate the entered source lines. A numerical array variable, I(X), stores the corresponding line numbers. The Editor program maintains a Line-count in variable I and a Data-count in variable C. The Line-count records the number of lines in the current file. The Data-count indicates the total bytes occupied by the current file. Since one byte is reserved for the file ending mark used in filing/retrieving procedure, the upper-limit for the Data-count is 4095 bytes (4096=4K). After shrinking an entered line, the program logic accumulates the length of this shrunken line and one extra byte into Data-count. The extra one byte is reserved for the character-count (length) of that line while dumping the file to disk. When the Data-count indicates that the current file has overflowed (greater than 4095 bytes), the program logic adjusts the size of the file by deleting the highest-numbered line until the Data-count is reduced under the limit (less than or equal to 4095 bytes).

Figure 6.1 lists all of the Editor command syntax and their corresponding operations.

6.2 Main Program Structure

The Editor program is started by setting the File mode flag to the First file mode. Unless the user issues an EXTEND command to alter the file mode, the disk accessing logic is always led to those tracks (tracks 37 & 38) where the First file resides.

After the command array is defined, the Line-count and

COMMAND SYNTAX	DESCRIPTION
New	Clears entered lines & enters First file mode
Extend	Clears entered lines & enters Extended file mode
Input	Inputs source lines containing line numbers
File	Files entered lines to disk
Call	Calls file from disk
List	Lists all lines of file on screen
XX	Lists line XX on screen
XX-	Lists lines XX through end of file on screen
-XX	Lists from start of file through line XX
XX-YY	Lists lines XX through YY on screen
Print	Prints all lines of file to printer
XX	Prints line XX to printer
XX-	Prints lines XX through end of file to printer
-XX	Prints from start of file through line XX
XX-YY	Prints lines XX through YY to printer
Delete XX	Deletes line XX from file
XX-	Deletes lines XX through end of file
-XX	Deletes from start of file through line XX
XX-YY	Deletes lines XX through YY from file
Quit	Exits Editor

** NOTE: XX & YY ARE LINE NUMBERS IN DECIMAL.
COMMANDS MAY BE ABBREVIATED BY FIRST INITIAL.

FIGURE 6.1 Command Summary for Editor

Data-count are both initialized to zero. Then the execution logic prompts the user to enter a command input. As shown in the command summary (Figure 6.1), a one letter abbreviation for the command is acceptable. If the leftmost character of the entered string is not a letter character, a syntax error message is sent, and the execution logic requests the user to re-enter a command. Otherwise, this isolated letter is compared with the entries of the command array. The execution logic will proceed toward the corresponding command routine, if a command is confirmed. Otherwise, the syntax error message will be displayed, and execution logic will accept a new user input.

Like the Extended Monitor, the QUIT command causes the Editor program to be terminated. As depicted in Figure 6.2, when this command is confirmed, the execution logic clears the Editor program from BASIC workspace by transferring control to the System Executive program.

Other commands are divided into two groups, the file mode related commands and the non-file mode related commands, as discussed in the following sections.

6.3 Non-File Mode Related Command Routines

Four commands are classified under this group. They are INPUT, LIST, PRINT, and DELETE. The common characteristic of these commands is that the algorithms are independent of the File mode flag.

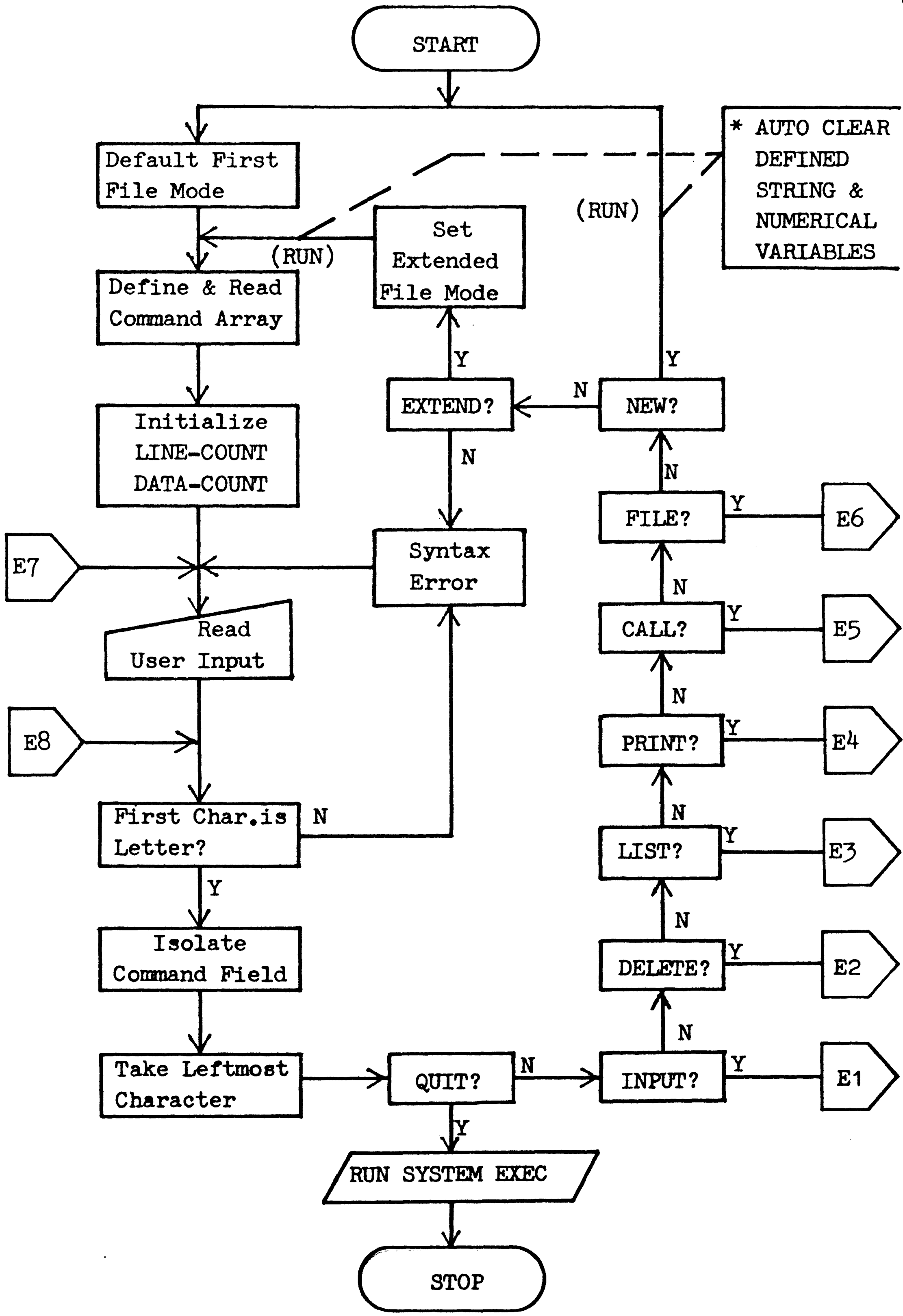

FIGURE 6.2 Flowchart for Editor Main Program Structure

6.3.1 INPUT Routine

This function allows the user to enter the source file. The execution logic sends the question mark to prompt the user to enter source file line by line. Each of the entered lines must be started by a non-zero number digit (1-9). The user may input those source lines in a random numbered sequence. This routine will place each entered line in the proper location by comparing this line number with other line numbers. This routine is exited when the user inputs a non-number led string or the file reaches its maximum limit (280 lines or 4095 bytes).

As may be viewed in Figure 6.3, the INPUT routine is started by checking the Line-count. If the Line-count records 280 lines already, a file-full message is sent and execution logic is routed to wait a new command input. Otherwise, the routine execution proceeds to accept a new line input. A question mark displayed on the screen indicates that the execution logic is ready to receive a new line. The user may order the Editor to implement other functions by simply typing the proper command instead of number-led line. Upon receiving the user entered string (A$), the execution logic tests the leftmost character of this string to determine whether it is a source line. If the leftmost character is not a non-zero decimal digit (1-9), the execution logic exits this routine, and routes to the command recognition procedure. If the test verifies that the input is a source line, the line array pointer, X, is defined by I+1. The entered line then is read into the line array position pointed by X. As aforementioned, this new entry must be rearranged by a shrinking

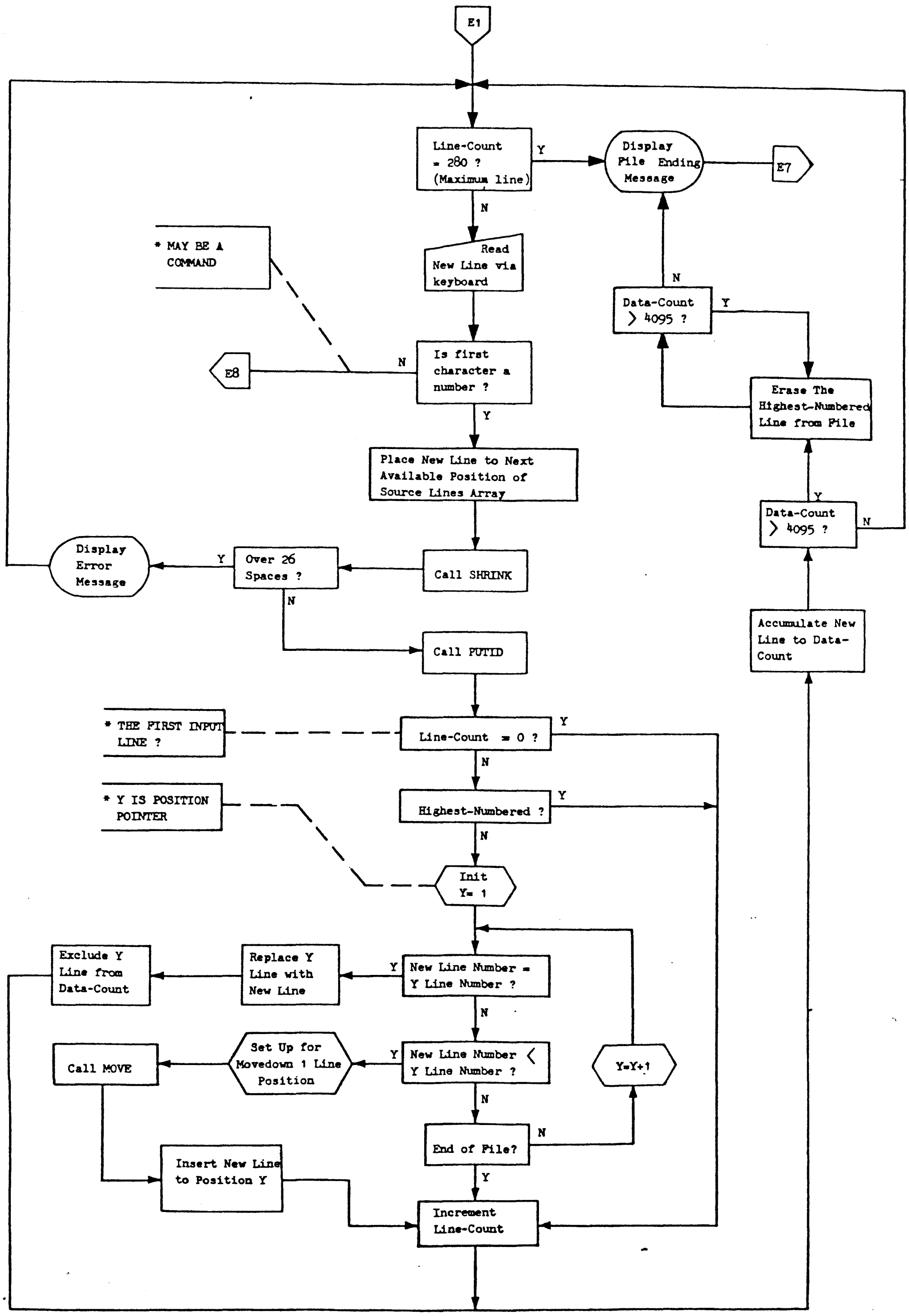

FIGURE 6.3 Flowchart for Routine INPUT

process. To do this, a subroutine SHRINK is called. SHRINK returns the shrunken line and its new character-count (length). Upon returning from this subroutine, the error flag is checked. If the error flag indicates that there is a violation on the space limit (twenty six spaces), then an error message is sent and the execution logic is led back to the beginning of this routine to request the user to re-enter a line. After the logic confirms that the entry is a valid line, another subroutine PUTID is called to collect and place the line number in the line number array position pointed by X. Then, the execution logic enters the sorting procedures.

If there is only one line in the source line array or the new entry has the highest line number, then the file is already in sequence. Otherwise, the further evaluation is proceeded. A FOR ... NEXT loop is applied to compare the line number of the new entry to other entries in the line number array. If the comparison logic detects the new line number is equal to the number in position Y, then the line in position Y is replaced by the new entry. If the new number is smaller than the number in position Y, then those lines starting from position Y through the end of file are repositioned by moving them down one line position, and the new entry is inserted to position Y.

After sorting all lines in sequence, the Line-count is increased to include the new entry. The character-count of the new entry is accumulated into the Data-count. If the Data-count indicates that the total characters is not over 4095 bytes yet, the execution logic routes to the beginning step of this routine. Otherwise, this overflowed file is adjusted by deleting the highest-numbered line. This adjustment is performed until the Data-count is reduced below the boundary. Then the execution logic sends a file-full message, and exits this routine.

6.3.2 LIST and PRINT Routines

The only difference between these two commands is the displaying destination. The LIST command sets the screen flag (F=1) which leads the displaying logic to screen. The PRINT command sets the printer flag (F=2), before routing to share the rest of the program statements with the LIST command.

In both cases, a specification field following the command syntax is optional. This specification field is used to enter the line specifications, in which a dash mark is the separator between the start line and end line. The user may specify both the start line and end line, or specify either one, or omit this field. The execution logic will replace the excluded specification with the corresponding default value.

As depicted in Figure 6.4, if there is no file established in the workspace, the execution control is simply transferred to the command recognition procedure to wait a new command entry. Otherwise, the execution logic proceeds toward the examination of the specification field. If there is no line specification, the displaying range defaults to the whole file. Otherwise, the subroutine STEND is called to scan the specification field, and return the displaying range. After checking the error flag returned by STEND and confirming no

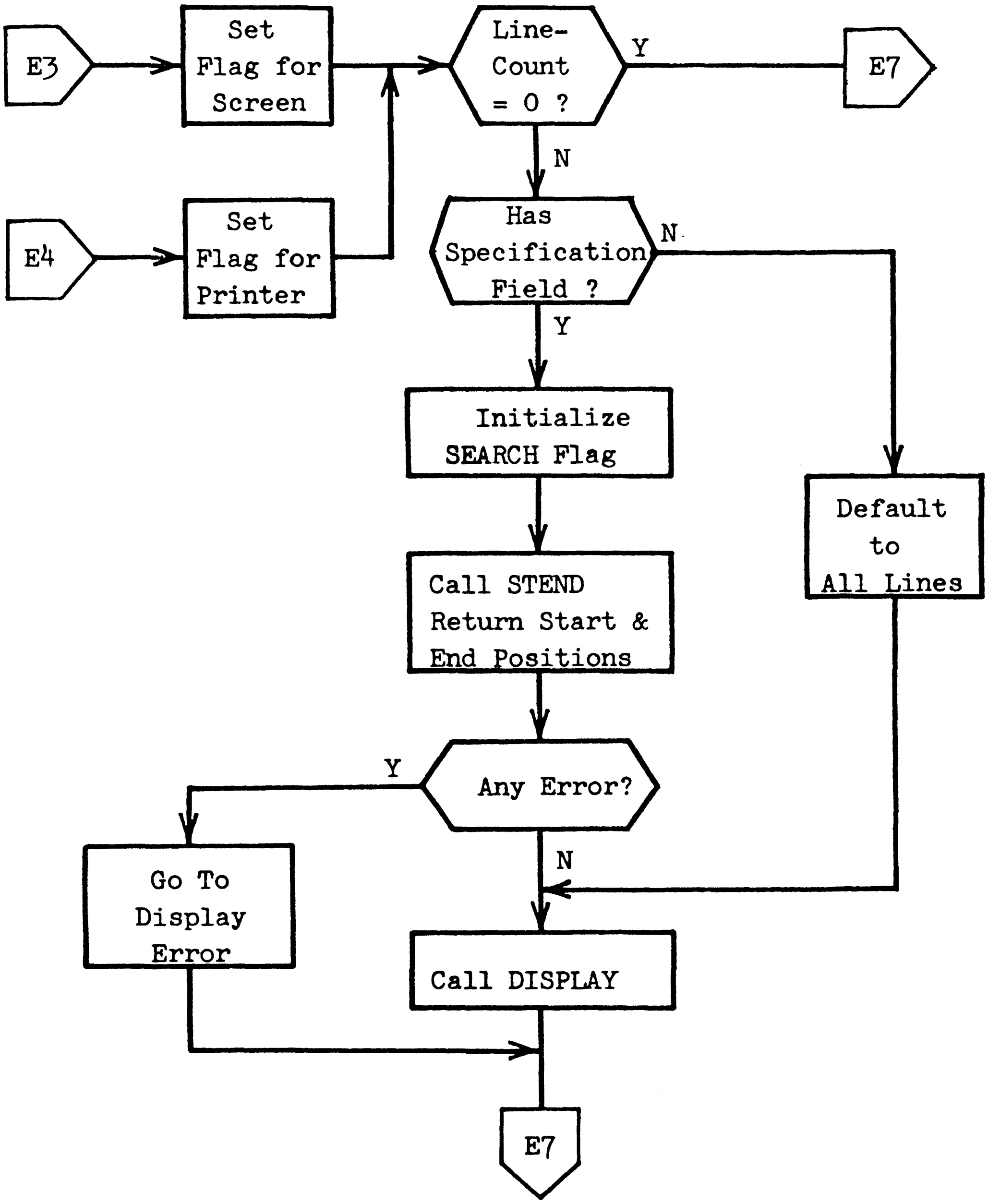

FIGURE 6.4 Flowchart for Routine LIST and PRINT

error, the subroutine DISPLAY is called to and send the designated lines to either the screen or the printer.

6.3.3 DELETE Routine

This command routine performs the deletion of a block of specified source lines. Unlike LIST and PRINT, this routine requires the presence of the specification field. As listed in the command summary (Figure 6.1), the user may specify both start line and end line, or specify either one. A dash mark is also used here to separate these two line specifications. If one of the line specifications is absent, the corresponding default value is used.

Figure 6.5 illustrates the execution sequence in flowchart form. As may be noted, if the execution logic detects a null specification field, this routine is exited. When the presence of the specification field is confirmed, the subroutine STEND is called to evaluate this field and return the deleting range. The specified lines must be existed in the file. Otherwise a 'NOT IN THE LISTING' message is sent.

The deletion work is accomplished by three procedures. First, those lines to be deleted are excluded from the Line-count and Data-count. Second, those lines, starting from the line just beyond the last line to be deleted through the end of file, are moved to new positions starting from where the first deleting line resided. The last procedure clears the useless array entries for faster DOS execution.

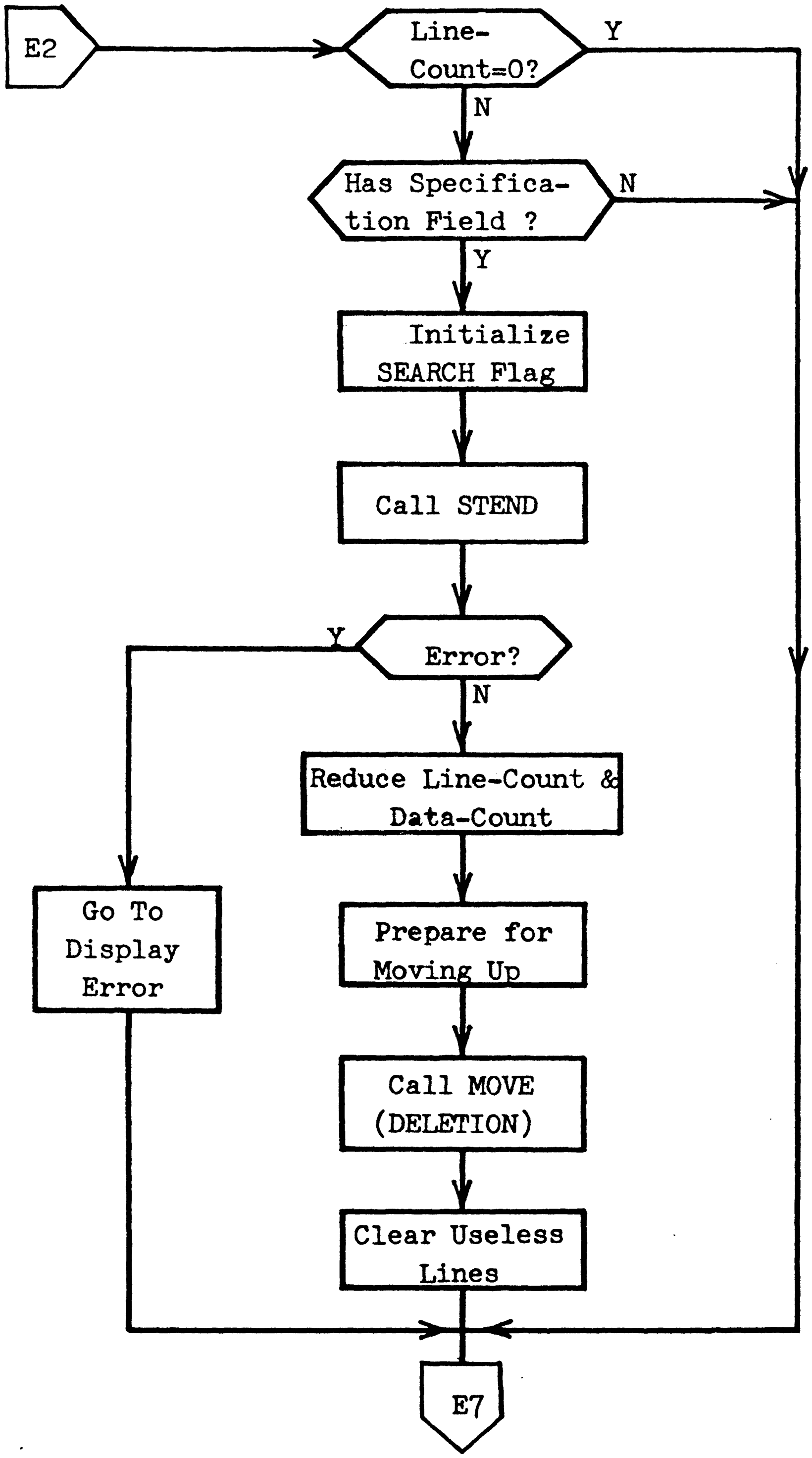

FIGURE 6.5 Flowchart for Routine DELETE

6.4 File Mode Related Command Routines

NEW, EXTEND, FILE, and CALL are the four members of this command group. The first two commands will change the File mode flag. The other two take the File mode flag as reference during operation.

6.4.1 NEW Routine

This command clears any entered lines and its corresponding arrays from memory, so that the user can have the full space to input a new file. The File mode flag is set to the First file mode. This command can be employed to clear the Extended file.

In order to destroy all of the defined variables, a very simple scheme is applied. According to the characteristics of the BASIC command, RUN, all of the established string arrays and numerical variables will be set to null by issuing this command. Therefore, this routine simply re-runs the main program statements starting from setting the First file mode, as shown in the main program flowchart (Figure 6.2).

6.4.2 EXTEND Routine

When this command is issued, the current file in memory is cleared, and the Editor enters the Extended file mode.

This operation starts with setting the File mode flag to indicate the Extended file. Then, as for NEW, the BASIC command, RUN, is executed to set all of the arrays and variables to null. As mentioned, the NEW command can be used to exit the Extended file mode.

6.4.3 FILE Routine

The FILE command routine dumps the current file to transition buffer, and saves this ASCII file on the disk. The execution logic will check the File mode flag to determine whether to use the First file tracks (tracks 37 & 38) or the Extended file tracks (tracks 29 & 30).

The capacity of the transition buffer is only 2K bytes (one track). Each file may occupy 2 tracks. Therefore, the execution logic checks to see if the buffer is full, after dumping a byte. Once the buffer address pointer (BA) exceeds its limit, the contents of this buffer is saved to the first track of the corresponding file. The rest of the file then is dumped and saved to the second track.

As explained in Figure 6.6, the dumping procedure for each of lines is started by storing the character-count (length) of that line to the buffer location pointed by BA. Then a FOR ... NEXT loop converts each of the characters to ASCII representation, and dumps this ASCII byte to the buffer. As noted, after dumping a byte, the buffer address pointer is checked. If the contents of BA indicates that the buffer is full, the execution logic checks the File mode flag and saves the current buffer data to the proper first track (track 37 or track 29). Before re-starting the dumping process, the buffer address pointer is initialized, and a track pointer (T) is increased to indicate the first track has been used already.

After having all of the source lines dumped, a null ASCII byte is placed as file end mark. Next, the execution logic examines the

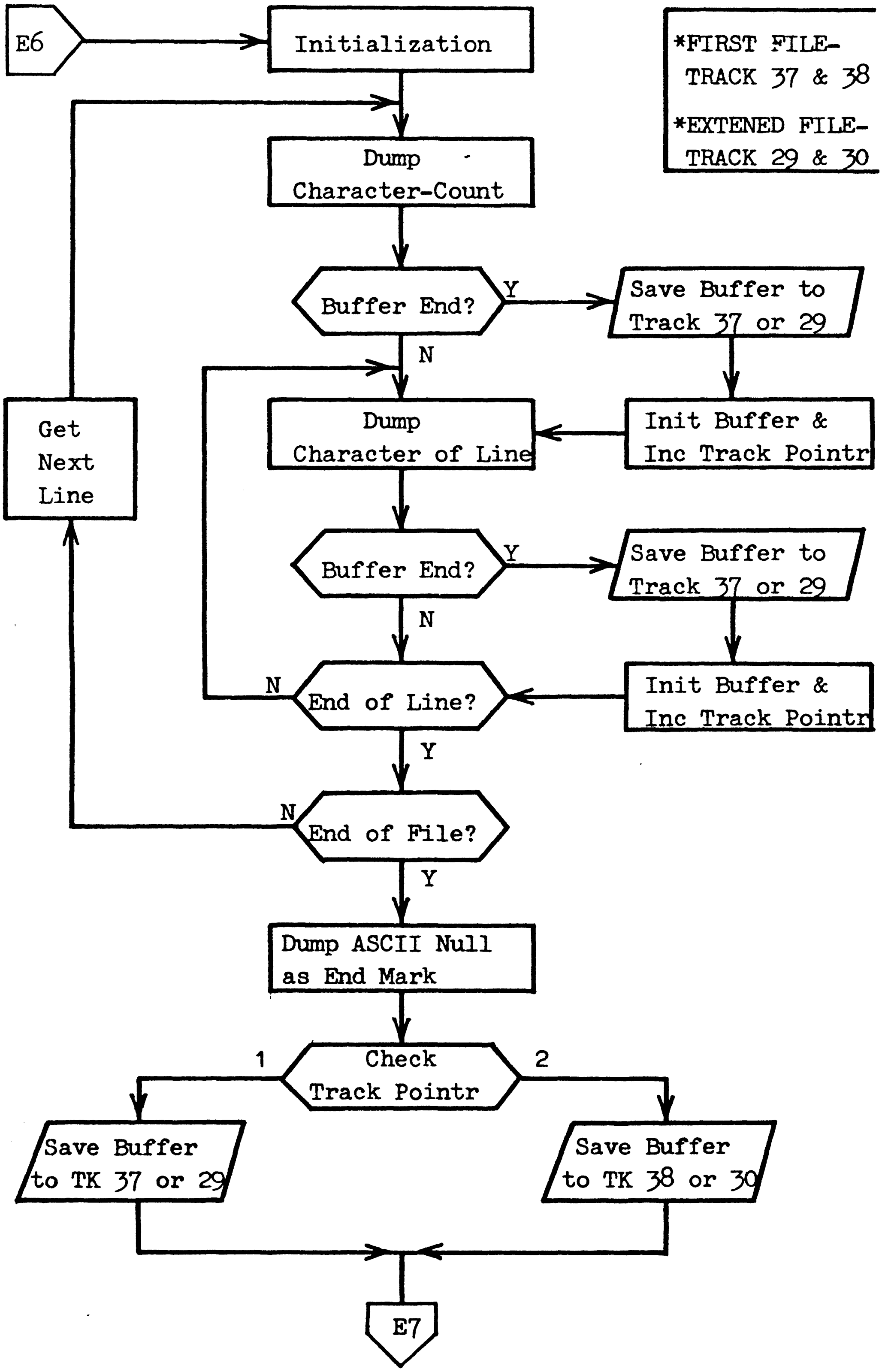

FIGURE 6.6 Flowchart for Routine FILE

contents of the track pointer. If the track pointer records that the first track is not available, the logic stores the buffer contents in the second track of the corresponding file designated by the File mode flag. Otherwise, the File mode flag guides the disk accessing logic to place the buffer data to either of the first tracks.

6.4.4 CALL Routine

This command is the inverse of the FILE function. It retrieves the First file or Extended file from disk, and reconstructs that file in the workspace. As with FILE, the current setting of the File mode flag designates which file to be retrieved.

The calling procedure is executed following the reverse order of filing. As described in the FILE routine, the length of each line is stored before dumping that line, and the last character in the file is a null ASCII byte. Therefore, after the retrieving logic loads the proper track contents to the buffer, the first byte obtained from the buffer must be the character-count of the first line. If the character-count is an ASCII null, this marks the end of the file. A non-zero character-count sets up a FOR ... NEXT loop to recover the succeeding characters of that line. After recovering a line, the execution increments the Line-count, accumulates the Data-count, and calls the PUTID subroutine to collect that line number. This process is repeated until the execution logic reads a zero character-count which marks the end of the file.

As with the FILE routine, a file may occupy more than one track of data. After reading a byte from the buffer, the execution logic

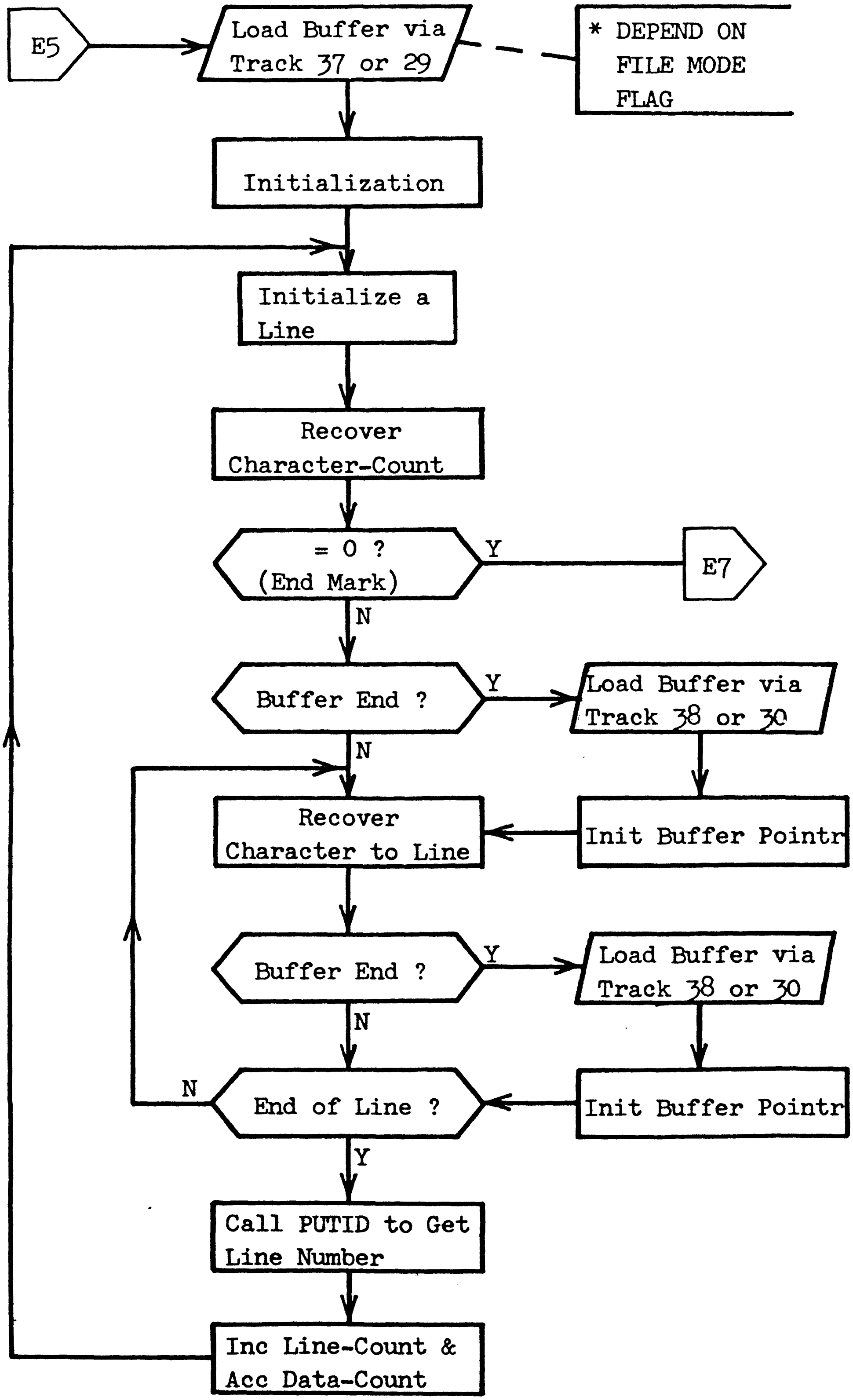

FIGURE 6.7 Flowchart for Routine CALL

checks the buffer address pointer. If the pointer indicates that the end of buffer has been reached, then new buffer contents are loaded from the second track of the corresponding file.

A flowchart for this operation is shown in Figure 6.7.

6.5 Subroutines

This section presents the execution flowcharts for certain important subroutines. In Figure 6.8 to Figure 6.13, the following subroutines are explained:

SHRINK -

Scans the source line pointed by X, and replaces the encountered spaces with one space character followed by an alphabetic character as repeat-count

RECOVER -

Recovers the source line pointed by X to its original form in T$ without changing its shrunken form

PUTID -

Puts the line number of the source line specified by X into the corresponding location of line number array

DISPLAY -

Recovers and sends a block of source lines starting from array location S through E to screen or printer

STEND -

Interprets the specification field or defines default value (s) in S and E

GETPOSITION -

Searches the location of the specified line number in the line number array and returns the appropriate location in X or T

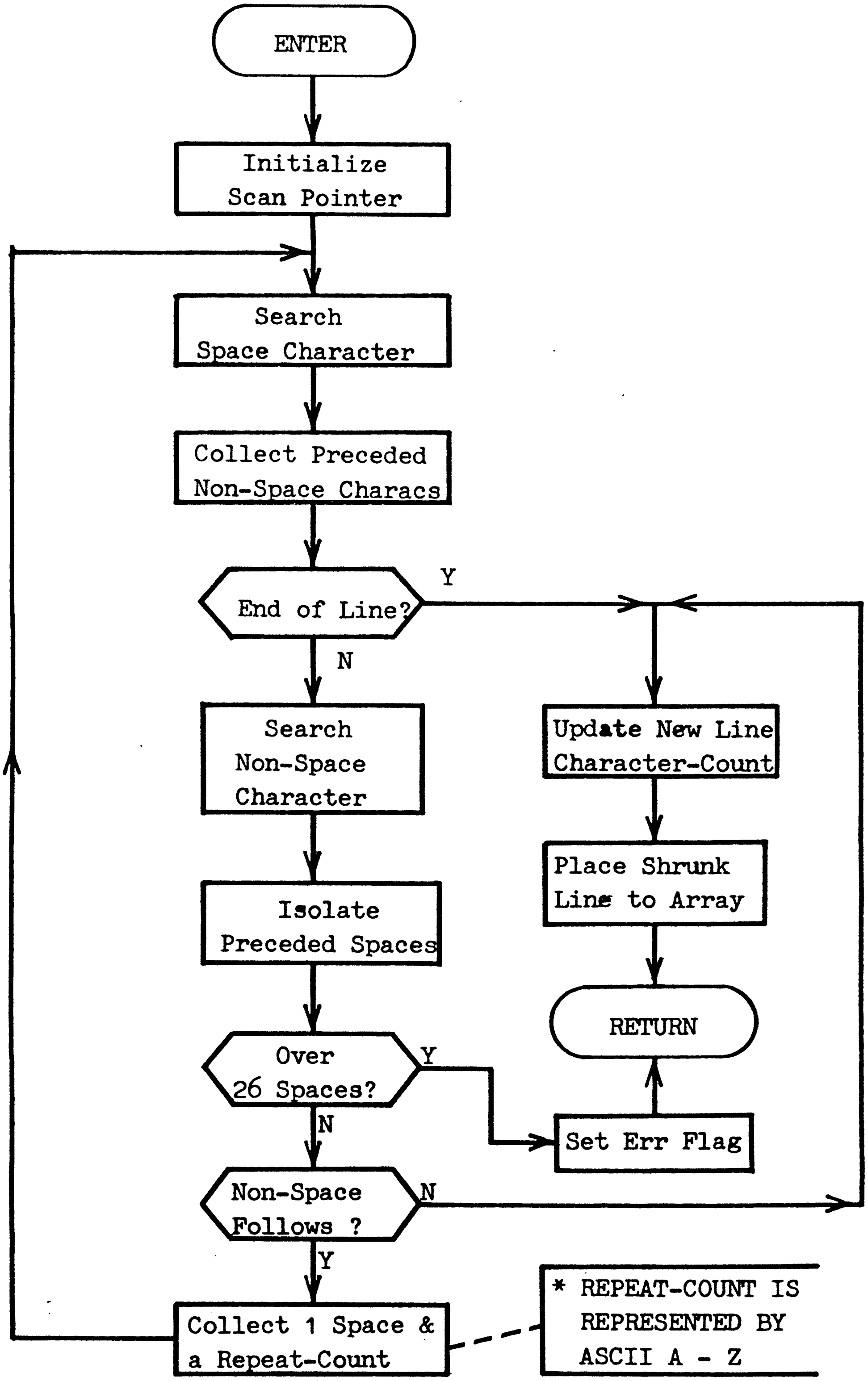

FIGURE 6.8 Flowchart for Subroutine SHRINK

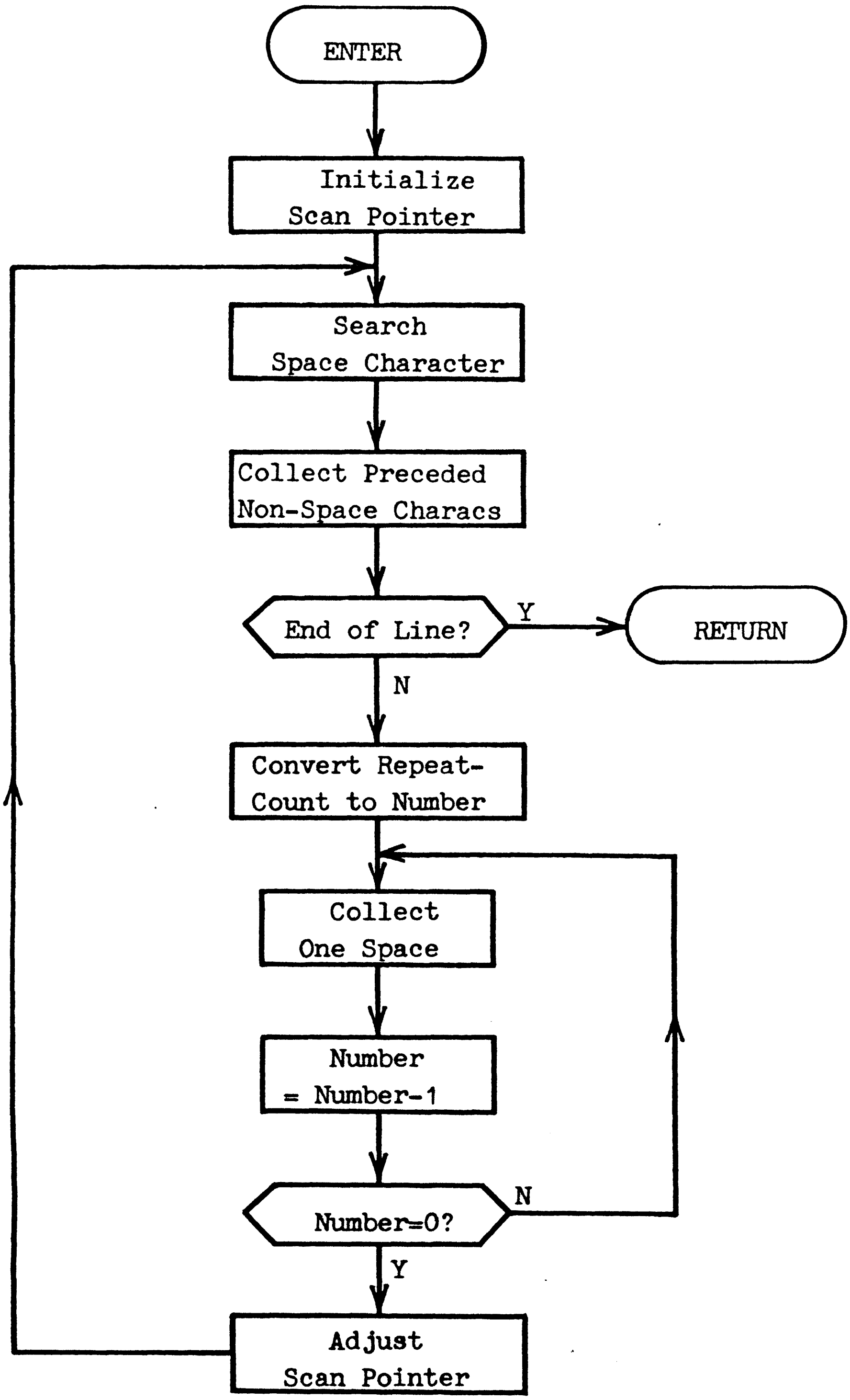

FIGURE 6.9 Flowchart for Subroutine RECOVER

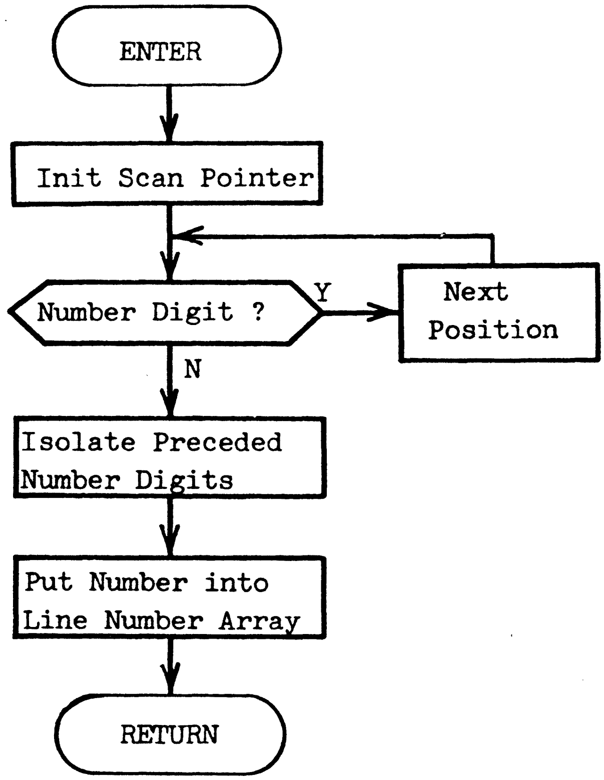

FIGURE 6.10 Flowchart for Subroutine PUTID

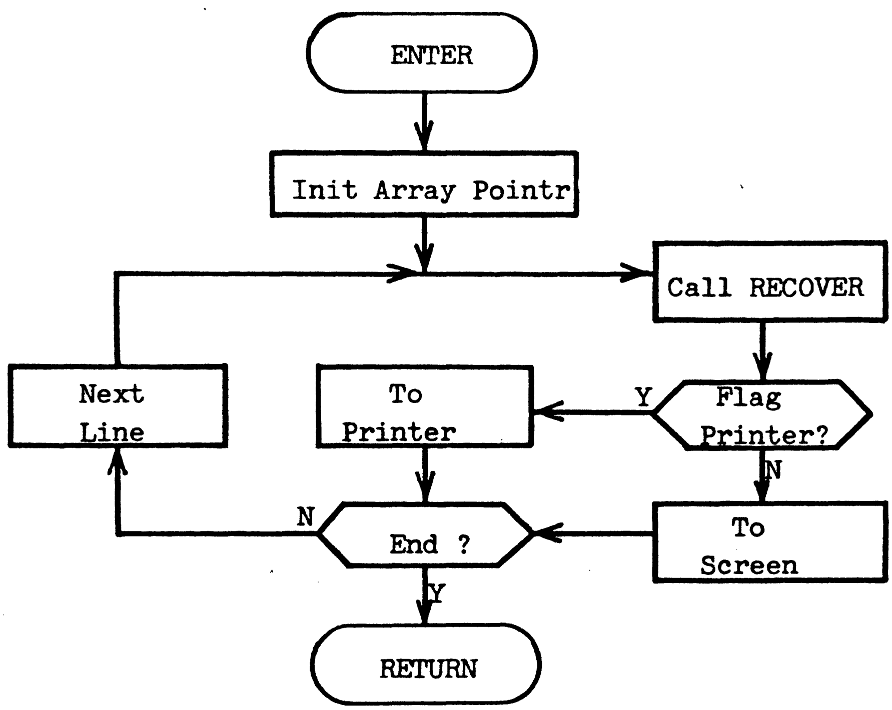

FIGURE 6.11 Flowchart for Subroutine DISPLAY

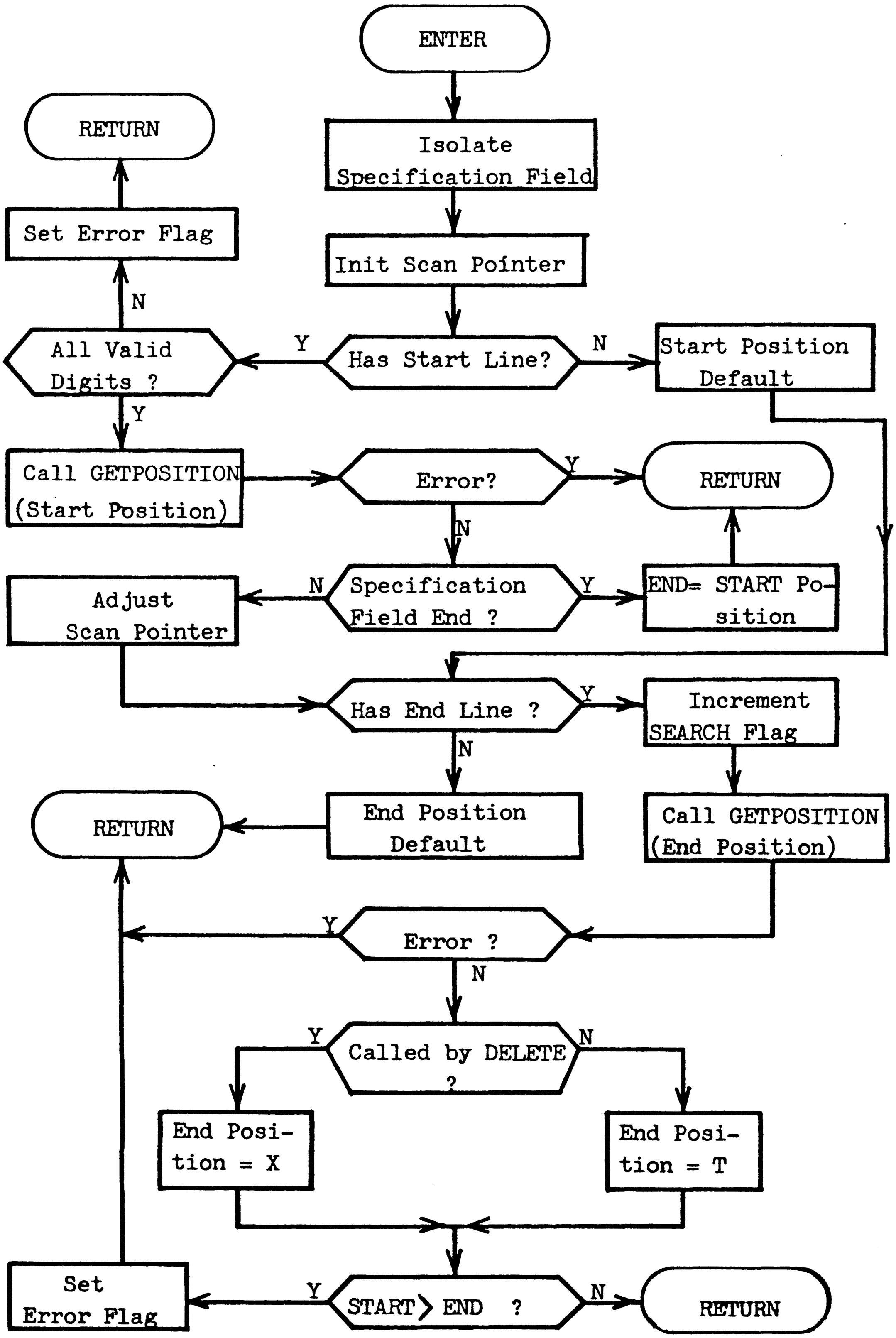

FIGURE 6.12 Flowchart for Subroutine STEND

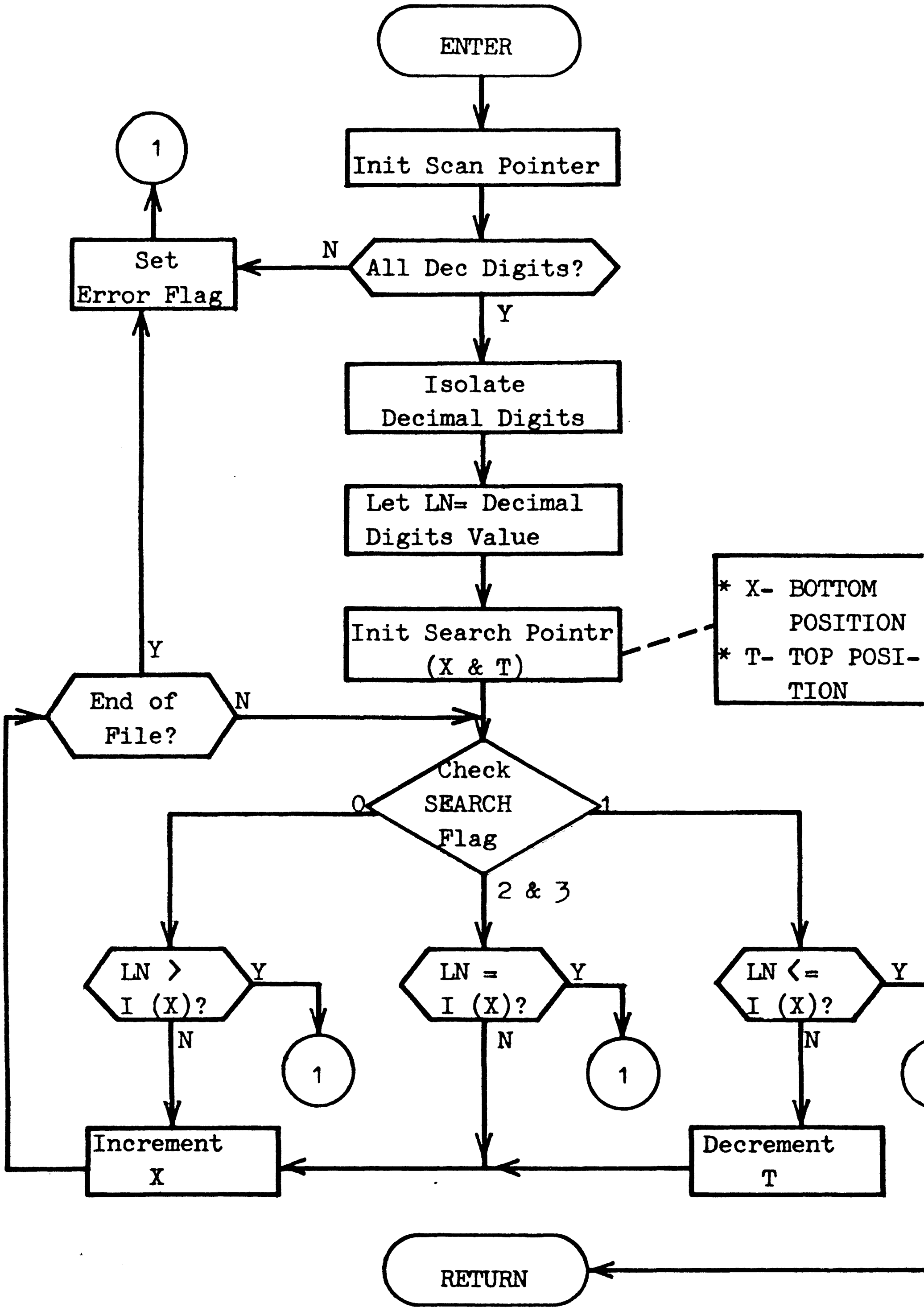

FIGURE 6.13 Flowchart for Subroutine GETPOSITION

CHAPTER 7 8080/8085 CROSS ASSEMBLER

7.1 Overview

To develop an assembly language processing program is the main purpose of this thesis. This Assembler program performs the clerical task of translating the 8080/8085 assembly language source program into the binary (machine) code language which can be executed by the 8080/8085-based microprocessor systems.

7.1.1 System Description

This Assembler program is written in BASIC language, and is stored on disk under the file name ASM85. It is loaded into BASIC workspace, and executed by the proper menu selection in the System Executive program.

Figure 7.1 presents the workspace memory assignment while executing the Assembler. A 2K buffer is reserved as a transition area for the source file created by the Editor. The buffer assigned for the object codes has a maximum capacity of 1K bytes. The Editor imposes a maximum capacity upon the source file of 8K bytes or 560 lines. A typical 8080/8085 assembly language source line includes line number, operation code field, and the spaces between them. If an average line occupies sixteen memory locations, then the source file comprises 512 lines. Assuming each line generates two bytes of object code, then a 1K buffer for the assembled code is adequate.

Before assembling the source file, the 1K buffer region is used as a temporary work area for the recovery of all reference tables

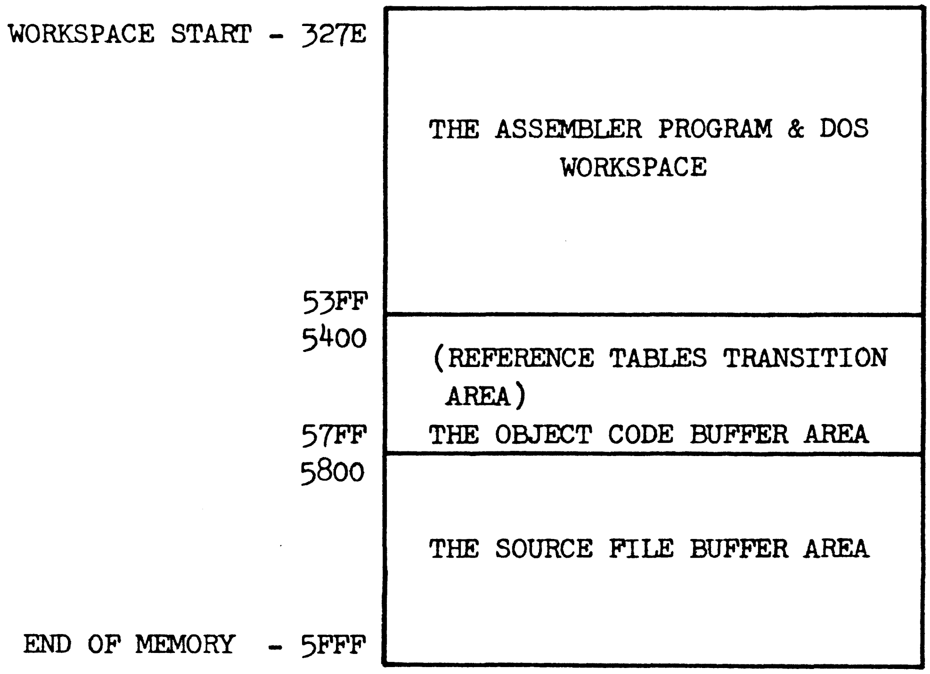

FIGURE 7.1 The Assembler Workspace Memory Map

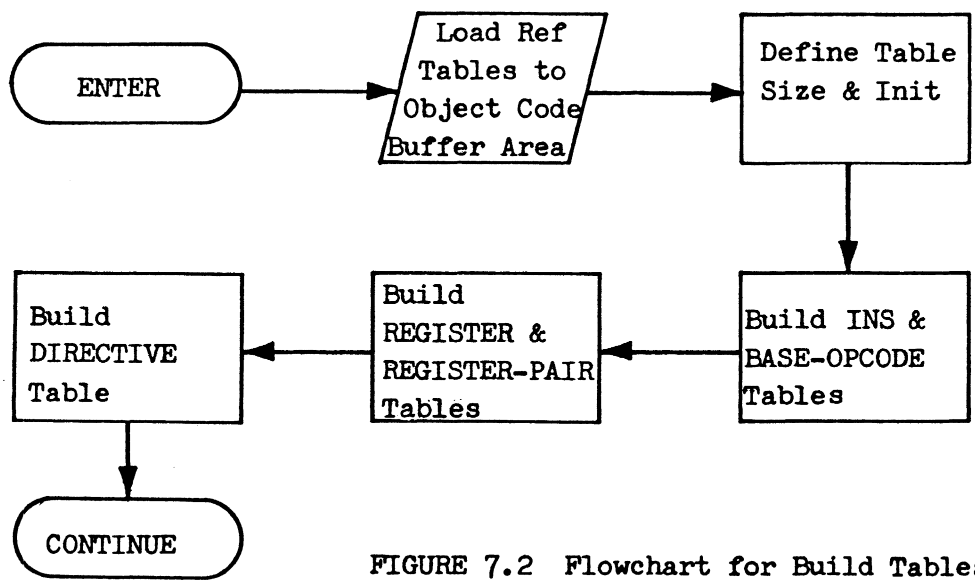

FIGURE 7.2 Flowchart for Build Tables

from the disk. Necessary records include the instruction table, the base-opcode table, the directive table, the register table, and the register-pair table. All table information is stored permanently on disk, and is transposed to corresponding arrays in the BASIC workspace. The reference table contents occupy disk sectors 4 & 5 of track 39. Except for the base-opcodes, all table entries on the disk are stored in ASCII form. ASCII null characters are used as separators between elements on the disk. For the instruction mnemonic record, disk storage consists of instruction characters followed by a separator and a corresponding base-opcode. Figure 7.2 shows the flowchart of the table construction sequence at the beginning of the Assembler main program.

7.1.2 Design Background

The reference tables could have been generated directly in the BASIC program by reading table entries from DATA statements, rather than recovering information from disk. However, the DATA statement occupies BASIC workspace even after the data has been read. Since the reference tables are large, considerable workspace memory can be saved by fetching the information from disk. After transferring the table data into the 1K object code buffer, the data is converted into BASIC string arrays in workspace memory. The contents of the 1K buffer are later overwritten during object code generation.

Other features of the Assembler have been designed in such a way as to minimize the requirements for BASIC workspace memory. Only one source line at a time is recovered from the source file buffer and operated on by the Assembler. All consecutive sequences of ASCII blanks in the source line are reduced to one space when brought into workspace. The comment field is not recovered into workspace.

Because the Assembler main program supports no comment field, it does not output the listing file. Instead, a subprogram SCRIBE is accessed to perform the listing task. Unless the source language file released by the Editor is 100% error free, the Assembler does not let the user select the listing function. Every detected error is sent to either screen or printer in error-code form. The meanings of the error codes are listed in the Appendix.

Like most of the assemblers written for microcomputers, a two-pass scheme is applied. In the first pass, the assembler simply collects and defines all symbols. In the second pass, it replaces the references with the actual definitions. Since a source file is physically read twice, and much time is consumed in the BASIC language interpreter, the assembling speed is slow.

7.1.3 Syntax Format

Many assemblers use fixed format. Some assemblers require that each field of a line start in a specific column. An example of this might be when there is no label field, the first column must be a blank. Another instance is when the operation code (mnemonic) field must start in the 7th column. The fixed formats are often a nuisance to users. Thus, for convenience, the design of this Assembler adopts a free format where the fields may appear anywhere in the line. To avoid confusion, it is required that the user retrain from using labels which are the same as instructions or directives.

The field assignment, like all assemblers, may consist of a label field (optional), an operation code (instruction or directive) field, an address field (conditional), and a comment field (optional). Each field must be separated by a proper delimiter. Figure 7.1 presents the standard Intel 8080/8085 assembler delimiters.

```
      :  - AFTER LABEL FIELD
'SPACE'  - BETWEEN OPERATION CODE AND ADDRESS
      ,  - BETWEEN OPERANDS IN THE ADDRESS FIELD
      ;  - BEFORE COMMENT
```

FIGURE 7.3 The Standard 8080/8085 Assembler Delimiters

For more flexibility to the user, this Assembler allows the first three delimiters shown in Figure 7.3 to be interchangable in all fields. Only the semicolon is always used to mark the comment field. For example, instead of using a colon after the label field, the user may type spaces or commas between the label field and the operation code field. The Assembler will also ignore the extra delimiters or the appearance of delimiters in comments.

7.1.4 Data Forms

Data in the address field may be presented in various forms. It may be a label, decimal value, hexadecimal number, binary digits, or ASCII characters. This Assembler accepts all of the above representations, and also allows simple arithmetic operations.

For 2's complement numbers, the equivalent decimal range for one byte of data extends from -128 to 255. Similarly, two bytes of binary data range from -2048 to 65535 in decimal representation. The Assembler converts any negative decimal values, in the address field, into the corresponding 2's complement form.

This Assembler will also handle arithmetic expressions involving the operators "+" and "-". The arithmetic expressions are evaluated from left to right, and no parentheses are accepted. The operands of the expression may be in the form of a label, decimal number, hexadecimal value, or binary representation. Care must be taken to eliminate any spaces between the operand and sign.

7.2 Main Structure

The structure of the Assembler main program can be illustrated by dividing it into five parts. These include initialization, first-field scanning, second-field scanning, error displaying, and the ending procedure. In processing through each pass of the Assembler, most of these operations are encountered. The Pass pointer variable, P, guides the logic of these procedures to the appropriate execution path.

Since this Assembler adopts a free format, the first group of characters collected by the scan logic may be a label, an instruction, or a directive. Unless the syntax logic confirms that this field is an instruction or a directive, the execution logic defines this field as a label, and second-field scanning is initiated. If a proper operation code is not found in scanning the first two fields, a syntax error code is generated. Subsequent field scanning (operand/address) is implemented by each operation code routine specified by the syntax logic.

The following variables are assigned to represent the important pointers and flags throughout the Assembler program.

P - Pass Pointer (1 -pass 1, 2 -pass 2)

X - Scanning Pointer

Y - Symbol Table Pointer

A - Source File Buffer Memory Pointer

S - Object Code Buffer Memory Pointer

U - Program Counter

E - Error Counter

R - Error Code

O - Display Flag (1 -printer, 2 -screen)

F - ORG Flag (1 -no ORG yet)

F2 - Filetype Flag (1 -first file, 2 -extended file)

7.2.1 The Initialization Procedure

The initialization process is the start of the Assembler program. It handles the housekeeping work for the Assembler, and provides necessary information to the Assembler for reference.

The execution logic of the Assembler begins in building the reference tables. The sequence of building these tables is depicted in Figure 7.2. The execution logic proceeds to prompt the user, and read a keyboard entry which defines the Display flag guiding the error code output to either screen or printer.

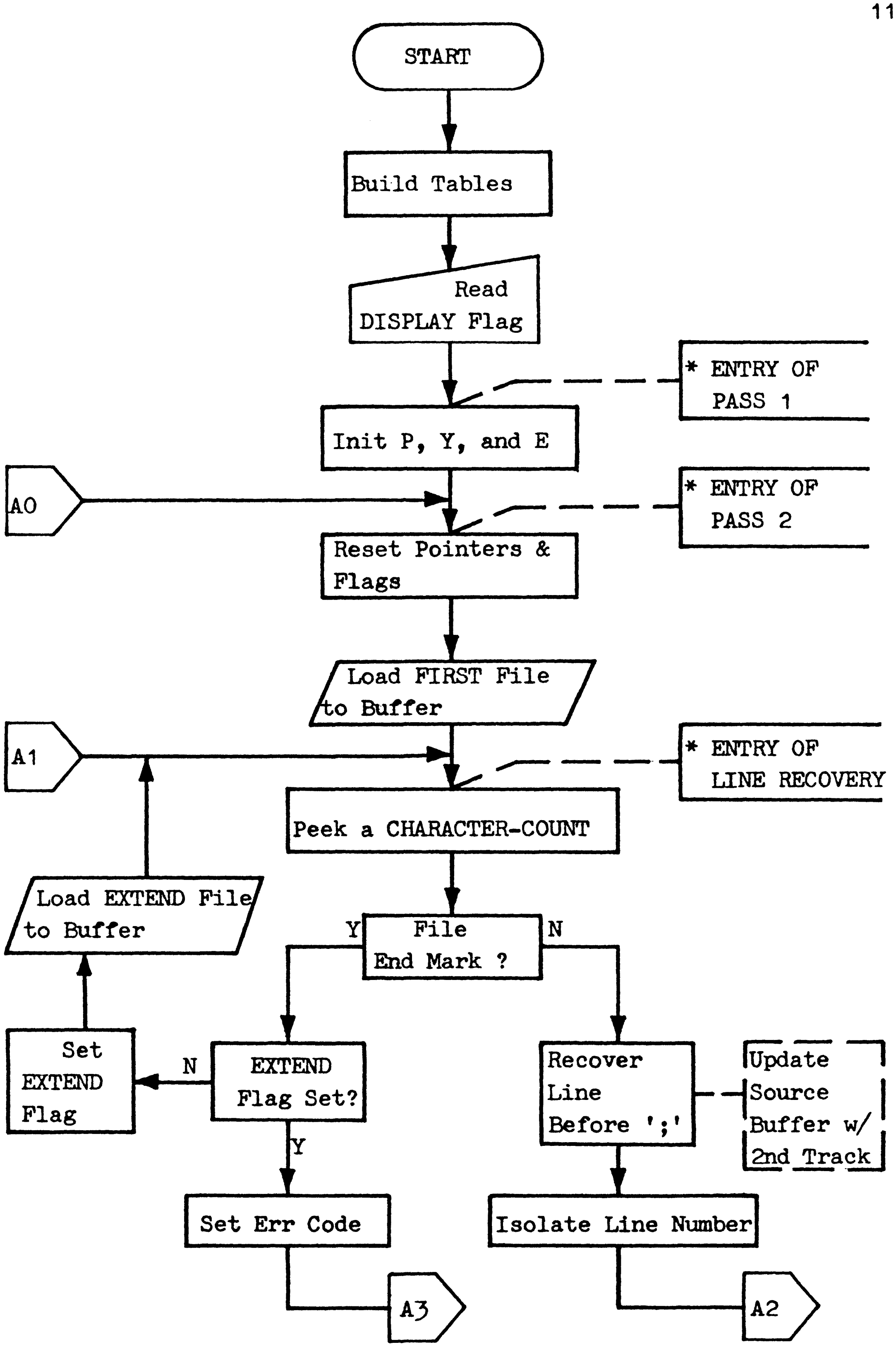

FIGURE 7.4 Flowchart for The Initialization Procedure

To begin pass 1, P is initialized to 1; Y and E are both zeroed. It should be noted that the entry point of pass 1 is only one logical operation before the entry point of pass 2. Common pointers and flags are then initialized. Subsequently the first 2K of source file is loaded to the source file buffer area.

The routine of scanning the source line begins with clearing the error code to zero. Before scanning the first field, a source line is converted into a string variable (I$) from the source file buffer. This conversion acts upon a line which was formatted by the Editor when saved to disk.

The first byte obtained from the buffer must be the character-count of that line. If the character-count is an ASCII null (00), this marks the end of the file. If a file-end mark is detected, the program checks the Filetype flag. As mentioned in Chapter 6, if the Filetype flag indicates that the current file in the buffer is not an extended file, then the logic loads the extended file from disk, and sets Filetype flag to 2. The file recovery process is repeated from the first line of the extended file. If the current file is the extended file and the end of file character-count is found, then the END directive has been omitted. The execution logic is led to the error display procedure.

A non-zero character-count sets up a FOR ... NEXT loop to recover the succeeding characters of that line. If multiple consecutive spaces are presented, they are represented as a single space followed by a repeat-count. In the Assembler only one space is loaded into BASIC workspace. The repeat-count is disregarded.

Character recovery is finished when the current line is ended or a semicolon is encountered. The source buffer memory pointer (A) points to the character-count of the next line.

A single file, even though not an extended file, may occupy more than one track of information. This means that during character recovery, the Assembler may need to access the disk in order to retrieve the remainder of the file. The subroutine CHKBUFF is called to check the buffer memory pointer (A). If the pointer indicates that the end of buffer has been reached, then new buffer contents are loaded from disk from the second track of the corresponding file.

After obtaining the line number of the current line, the execution flow is routed to the field scanning procedures. The execution algorithm for this part of the program is presented in Figure 7.4.

7.2.2 The First Field Scan Procedure

The field scanning procedure starts by calling the subroutine ISOLATE. ISOLATE is the only subroutine for line scanning in this Assembler. It starts collecting characters after finding a valid symbol (an alphanumeric digit, single quotation mark, or minus sign), and stops when the line ends or any delimiter (space, comma, or colon) is encountered.

If the current line is a comment line or has no valid starting character, then the execution logic recovers the next line. Otherwise, syntax logic starts classifying the first group of characters.

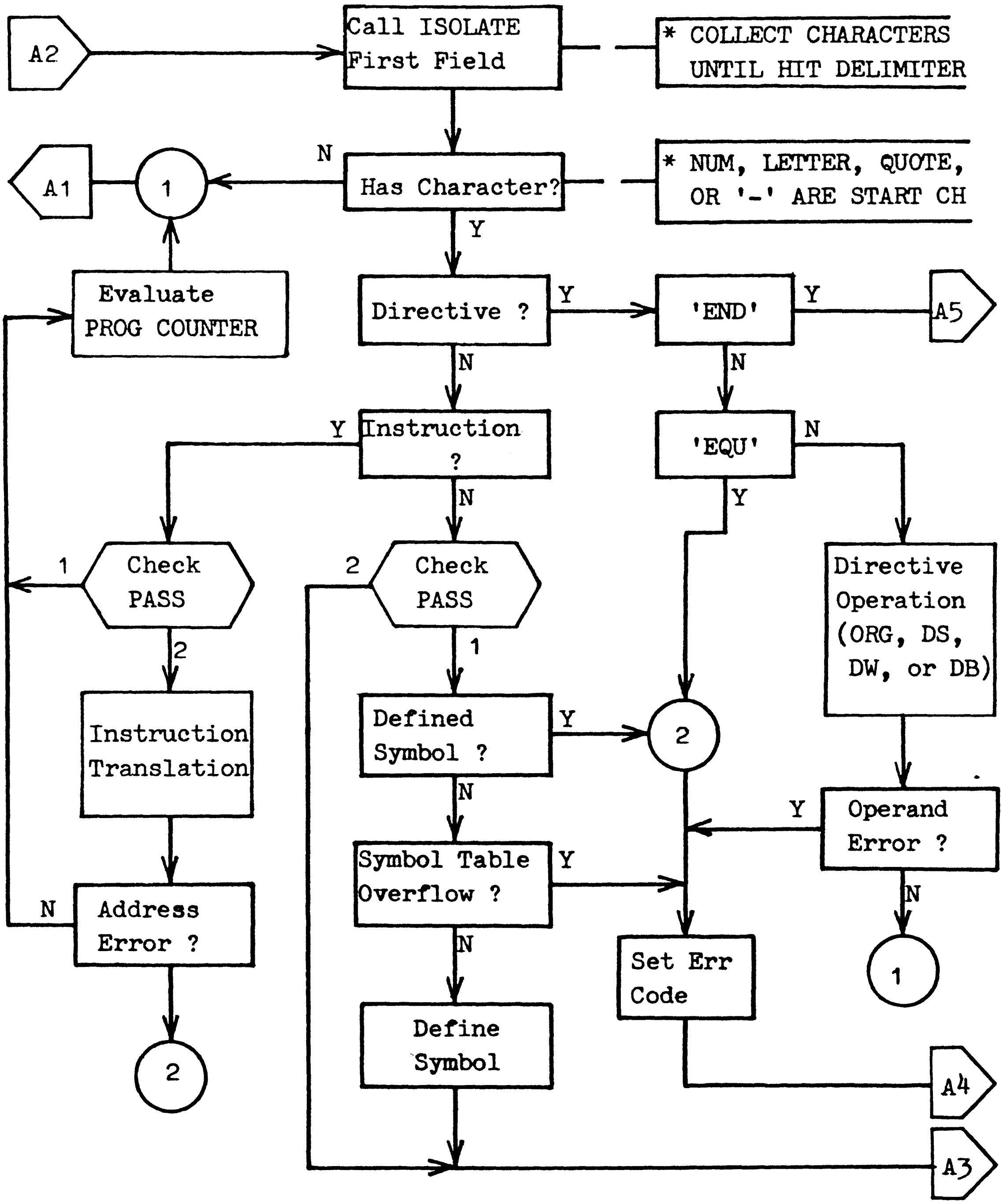

FIGURE 7.5 Flowchart for The First Field Scan Procedure

Directive EQU is not permitted in the first field, since EQU must be preceded by an alphanumeric label. If the content of this field is not a directive, then the subroutine SEARCH MNE is called to determine whether it is an instruction. The returned variable Z contains the result of the search, with a zero meaning no instruction found, or a 1-3 indicating the number of bytes required by the verified instruction. As shown in Figure 7.5, if the syntax logic confirms that it found an instruction, the proper execution path is determined by the Pass pointer, P. Pass 1 only adds Z to the current Program counter (U), and to the object code buffer pointer (S). Pass 2 performs the actual opcode and address field translations.

The Pass pointer is also checked, if this field is neither a directive nor an instruction. Pass 1 checks the symbol table to see if it is a multiple defined label, and to see if the table is full (maximum 100 entries). The first six characters of this group are taken as a label and placed into the symbol table, if the above two checking procedures are satisfied. Since all labels are defined in pass 1, pass 2 neglects label definition and proceeds to second field scanning directly.

7.2.3 The Second Field Scan Procedure

Since the characters collected in the first field are not an operation code, the syntax logic collects and scans the second group of characters. If the second group is still not a directive or instruction, the syntax error code is generated.

Like first field scanning, the procedure starts by calling the

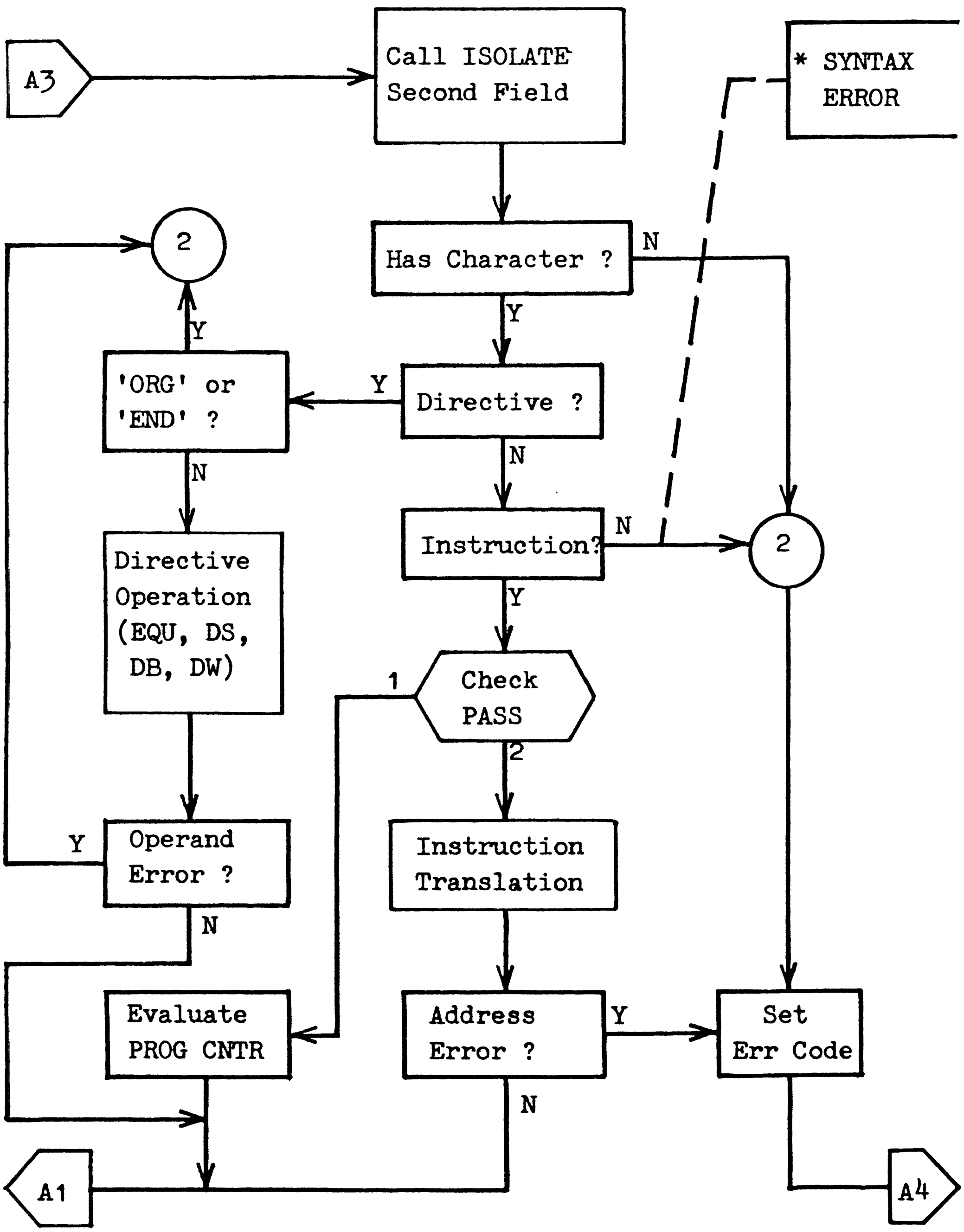

FIGURE 7.6 Flowchart for The Second Field Scan Procedure

subroutine ISOLATE to collect the second group of characters. No valid starting symbol or a non-operation code causes the syntax logic to set a syntax error code and route to error displaying. ORG and END are the two directives which cannot be preceded by label. It is a illegal statement if one of these two is found in the second field. Other directives lead the execution flow to the corresponding operation routine.

As shown in Figure 7.6, the same algorithm which is used in the first field scanning is also applied here. The Pass pointer leads the execution logic to the appropriate path. Pass 1 evaluates Program counter; pass 2 executes the found instruction translation routine.

7.2.4 The Error Display Procedure

Because the program is shared by both pass 1 and pass 2, certain errors are repeatedly generated. It is therefore necessary to determine when an error code should be displayed.

Error codes 1 to 4 are permitted to be displayed in pass 1; error codes 5 to 9 are displayed in pass 2. Other entries are rejected by this procedure, and return the execution control to the step of recovering the next source line.

Before displaying the accepted error code on the screen/printer, Error-count (E) is incremented to record this error. Then the user-defined Display flag leads the displaying statement to either screen or printer. As depicted in Figure 7.7, the last operation of this procedure is examining the error code again. If it indicates an

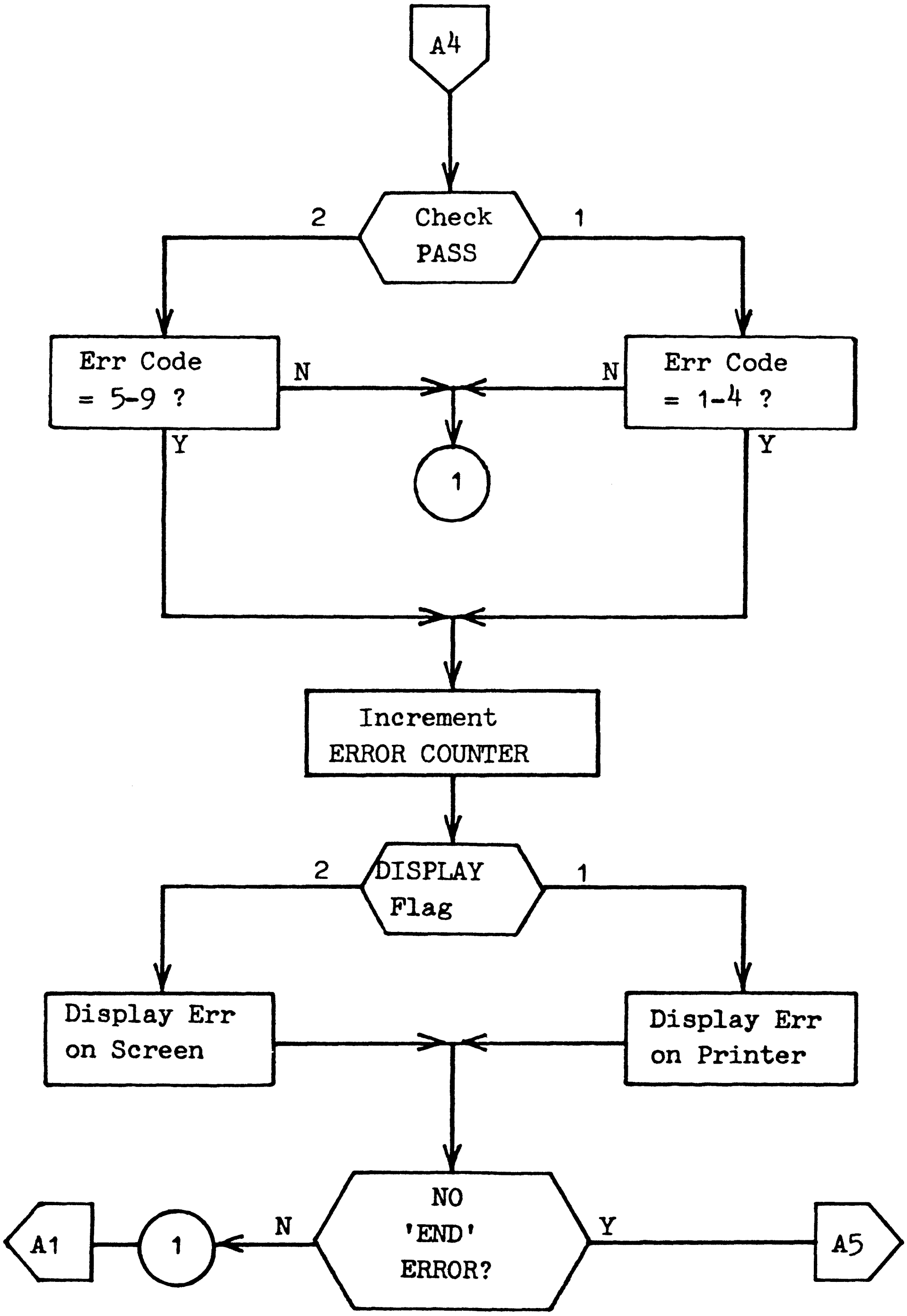

FIGURE 7.7 Flowchart for The Error Display Procedure

NO END error (code=9), then execution logic routes to the ending procedure, rather than recovering the next line.

7.2.5 The Ending Procedure

This procedure is entered when END directive is found or a file ending mark is hit.

First, the object code buffer pointer is checked to see if the size of the generated object codes is over the 1K limit. If it does exceed the boundary, no pass 2 will be processed, and the execution flow is led to the error ending procedure. Second, the Pass pointer is checked. Pass 1 increments the pointer to 2, and re-enters the initialization procedure for pass 2 operation. If Pass pointer indicates that the pass 2 operation is completed, then Error-count is checked to determine the next step. Non-zero Error-count leads the execution flow to the error ending procedure, in which the Editor may be selected for error corrections or the System Executive program take over the control. If Error-count indicates no error was detected throughout the assembling work, the hexadecimal values of start and end of Program counter are loaded to the first four bytes of the object code buffer. As illustrated in Figure 7.8, after the entire contents of the object code buffer are copied to disk track 36, the execution logic prompts the user to determine the destination. The user may select the listing function by simply entering "Y". Otherwise the System Executive program is loaded from the disk and executed.

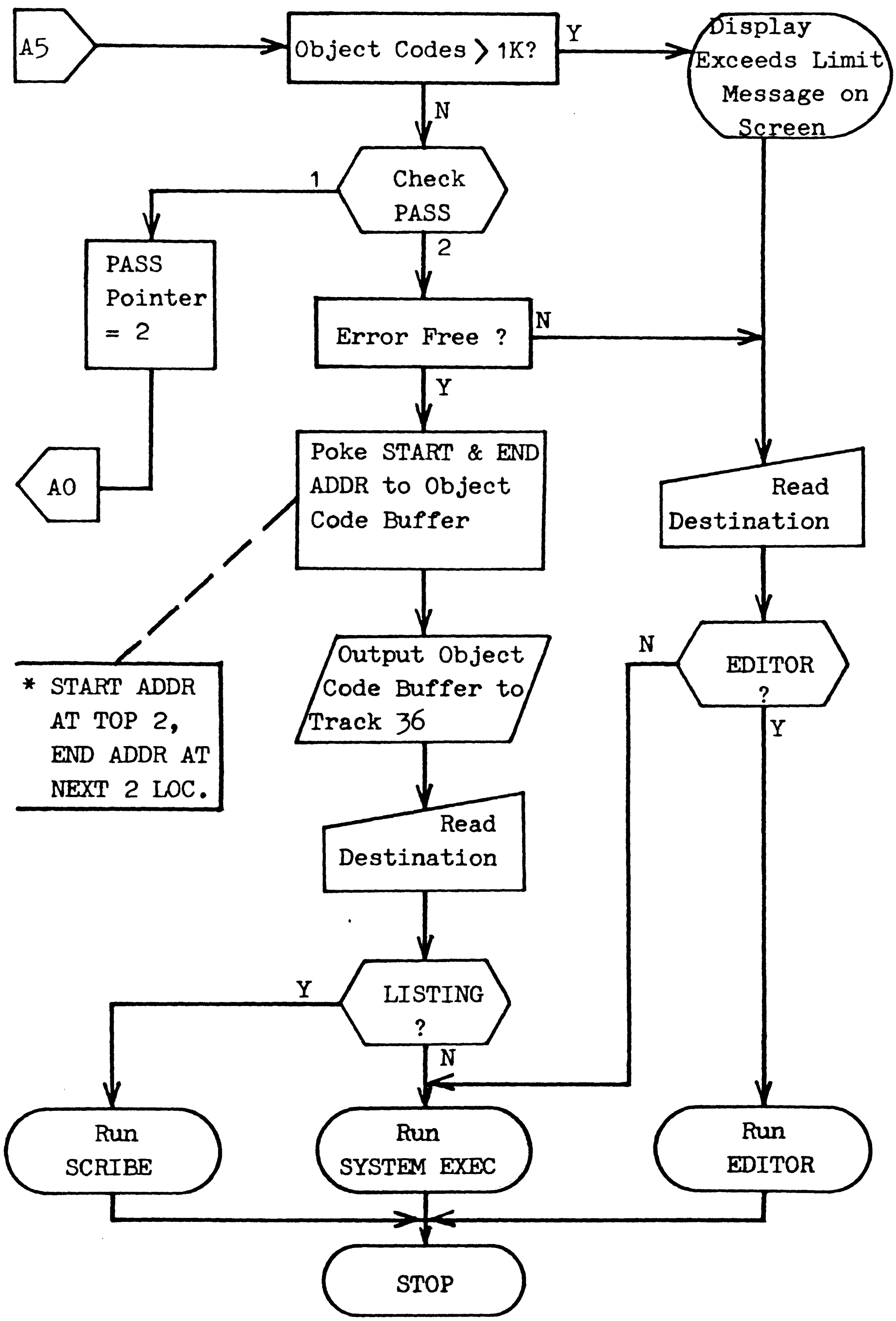

FIGURE 7.8 Flowchart for The Ending Procedure

7.3 Instruction Translation

An instruction may be interpreted into an one-byte, a two-byte, or a three-byte instruction. When syntax logic leads the execution to the corresponding instruction routine in pass 2, the found instruction mnemonic and its base-opcode have been pointed to by the variable T.

7.3.1 8080/8085 Opcode Organization & Manipulation

By examining the opcode table and the instruction format of the Intel 8080/8085 microprocessor, an important algorithm can be found. That is, all the opcodes of register-related instructions are based upon the associated sequence of register/register-pair. Therefore, these opcodes can be obtained by manipulating the base opcode with proper offset value.

According to this algorithm, the register sequence of the register table and the register-pair table are built as shown in the following figure.

REGISTER TABLE								
ARRAY SUBSCRIPTS	0	1	2	3	4	5	6	7
REGISTER SYNTAX	B	C	D	E	H	L	M	A

REGISTER-PAIR TABLE				
ARRAY SUBSCRIPTS	0	1	2	3
REGISTER-PAIR SYNTAX	B	D	H	SP

FIGURE 7.9 The Register Array and Register-pair Array

Those opcodes, which are related to Register B, are chosen as the base opcode for the corresponding instruction mnemonic family. The actual opcode then can be acquired by developing an arithmetic expression involving the array subscripts manipulation of the corresponding register. For instance, the opcode for INR B is hexadecimal value 04, then the opcodes for the entire INR family can be found by performing the following arithmetic operation:

4+(S*8) ; S is the subscript of the corresponding register

Each instruction family has its arithmetic expression to manipulate its base opcode. Figure 7.10 lists the register-related instructions and the corresponding arithmetic expressions. All of the base opcodes in the expressions are presented in decimal form. As noted in the figure, POP and PUSH families use a different register-pair table. Since this is the only exception, no extra table is built for this purpose. The element SP in the register-pair table simply is tempararily replaced with PSW when POP or PUSH is met. It may also be noted that RST family uses no table. The number digit following RST is used in the expression.

Those instructions which are not listed in Figure 7.10 use absolute opcode from the base-opcode table directly. The total entries of the instruction table and the base-opcode table are seventy nine.

7.3.2 The One-byte Instruction Routine

Most of the register-related instructions are one-byte instructions. Entering this routine with variable T containing the

INSTRUCTION	ARITHMETIC EXPRESSION	REGISTERS USE
MOV r1, r2	OPCODE = 64+(S1*8)+S2	B,C,D,E,H,L,M,A
INR r	OPCODE = 4+(S*8)	B,C,D,E,H,L,M,A
DCR r	OPCODE = 5+(S*8)	B,C,D,E,H,L,M,A
ADD r	OPCODE = 128+S	B,C,D,E,H,L,M,A
ADC r	OPCODE = 136+S	B,C,D,E,H,L,M,A
SUB r	OPCODE = 144+S	B,C,D,E,H,L,M,A
SBB r	OPCODE = 152+S	B,C,D,E,H,L,M,A
ANA r	OPCODE = 160+S	B,C,D,E,H,L,M,A
XRA r	OPCODE = 168+S	B,C,D,E,H,L,M,A
ORA r	OPCODE = 176+S	B,C,D,E,H,L,M,A
CMP r	OPCODE = 184+S	B,C,D,E,H,L,M,A
RST 0-7	OPCODE = 199+(0-7)*8	NON
POP rp	OPCODE = 193+(S*16)	B, D, H, PSW
PUSH rp	OPCODE = 197+(S*16)	B, D, H, PSW
STAX rp	OPCODE = 2+(S*16)	B, D
LDAX rp	OPCODE = 10+(S*16)	B, D
INX rp	OPCODE = 3+(S*16)	B, D, H, SP
DCX rp	OPCODE = 11+(S*16)	B, D, H, SP
DAD rp	OPCODE = 9+(S*16)	B, D, H, SP
MVI r, D8	OPCODE = 6+(S*8)	B,C,D,E,H,L,M,A
LXI rp, D16	OPCODE = 1+(S*16)	B, D, H, SP

NOTE: S is the subscript of the register sequence in table

FIGURE 7.10 Base Opcodes & Arithmetic Expressions Table for Register-related Instructions position index of the found instruction, a base opcode is obtained from the corresponding location of the base-opcode array. T is then checked to determine whether the found instruction is a register-related instruction. The program logic assigns the execution to the proper instruction family procedure to get actual opcode. If the entered instruction is not a register-related instruction, the base opcode is used as actual opcode. The execution sequence is explained in Figure 7.11, where B represents the base opcode and S stands for the subscript of the register/register-pair in the table.

After the proper opcode is obtained, the subroutine POKEBYTE is called to place this byte into the object code buffer. Then, the execution logic checks to see if there are any unnecessary fields. The error-free exit is to recover the next source line. Any detected error causes the execution to go to the error display procedure.

7.3.3 The Two-byte Instruction Routine

In the entire two-byte instruction family, only MVI is a register-related instruction. If variable T indicates that MVI is met, the procedure of obtaining the actual opcode for the MVI family is performed. Otherwise the execution logic by-passes the MVI process and calls POKEBYTE to dump the opcode. After opcode is placed at the proper location, the subroutine GETDATA is employed to scan the operand field and return the decimal operand value in variable D. As shown in Figure 7.12, if the returned error code indicates that GETDATA could not find an operand (code=1), then the

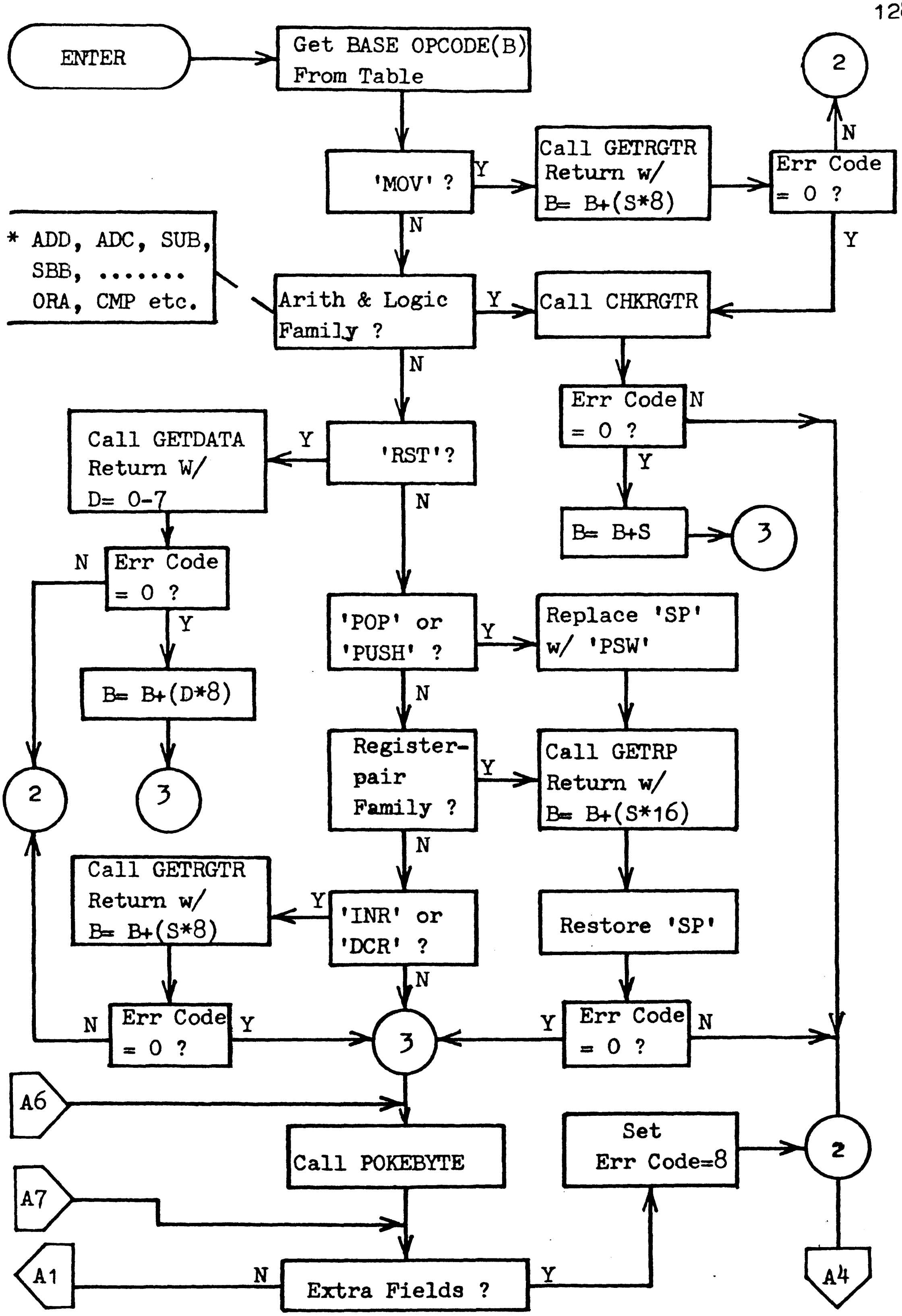

FIGURE 7.11 Flowchart for One-byte Instructions Translation

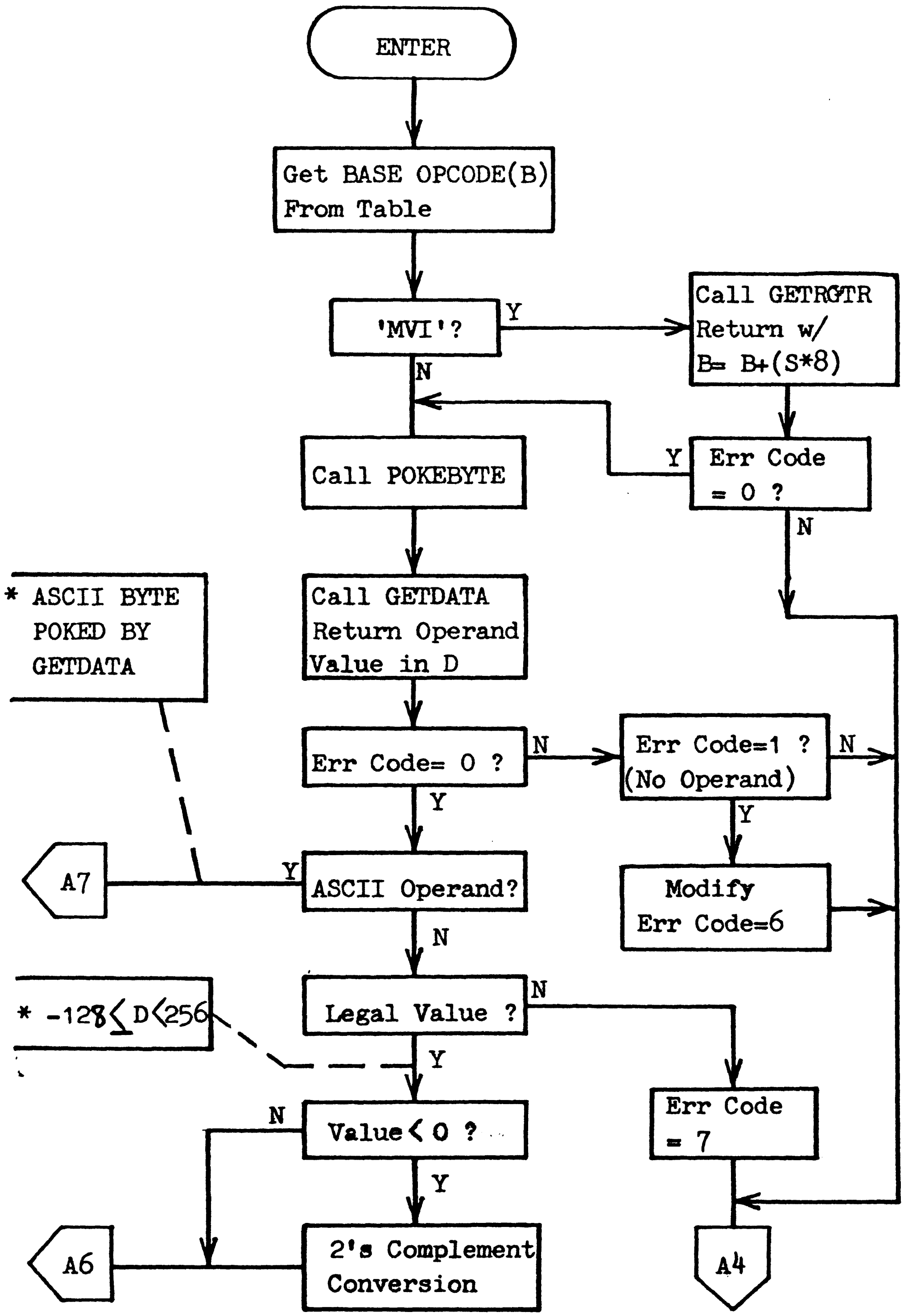

FIGURE 7.12 Flowchart for Two-byte Instructions Translation

error code is modified to 6. Because the error code 1 is not displayed in pass 2. The logic also checks to see if the operand is an ASCII character. Since ASCII data has been dumped to buffer in GETDATA, the POKEBYTE statement is by-passed.

The permissible data value in decimal is ranged from -128 to 255. Exceeding this limit causes an error to be sent. If the data value is acceptable, the program logic examines the sign of this data. A Negative value is converted to the 2's complement representation. The operand byte allocation and the extra field checking procedures are shared with the one-byte instruction routine.

7.3.4 The Three-byte Instruction Routine

Like the two-byte family, only one instruction, LXI, is register-related in this family. The algorithm shown in Figure 7.13 is similar to the two-byte instruction routine. Since this routine sees the operand as a word (two bytes), the valid range for the returned data is from -2048 to 65535 in decimal representation. The subroutine POKWORD automatically performs the 2's complement conversion if the given data is negative.

7.4 Directive Operation

The directives of the standard Intel 8080/8085 assembler are not all allowed to be used in this Assembler. Several of the pseudo-operations provided by the Intel assembler are not commonly used, and the limited workspace does not have the capacity to accomodate all of the directive operations. Therefore, only those

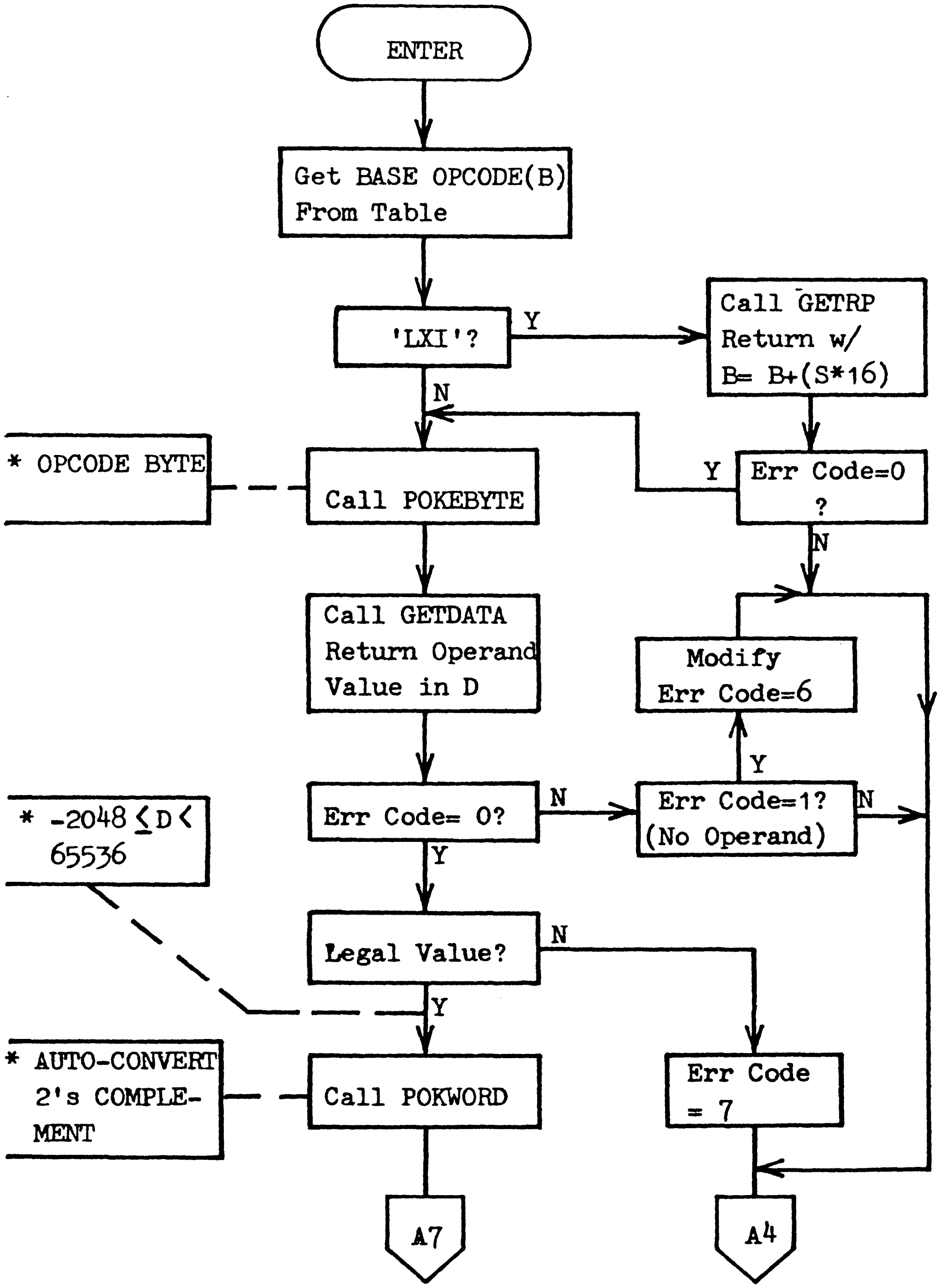

FIGURE 7.13 Flowchart for Three-byte Instructions Translation

frequently used directives are included in this Assembler. They are ORG, EQU, DS, DW, DB, and END.

Since the END directive operation is included in the ending procedure, no description is written for it in the following subsections.

7.4.1 ORG Operation

The ORG directive sets the Program counter to the value specified by the operand field, in which the operand may be in the form of a label, decimal number, hexadecimal value, or binary digits. Because the pointer of the 1K object code buffer is initialized to the start of buffer locations, multiple ORG's must specify address in ascending sequence. Otherwise the former loaded object codes might be overwritten by the latter dumped codes.

As shown in Figure 7.14, the ORG flag is developed to distinguish the first met ORG from others. This flag is reset at the pass entries in the initialization procedure. When an ORG is met, the execution logic checks ORG flag to determine the execution path. If the flag indicates that this is the first ORG operation, then the Program counter (U) is equated to the address value returned by GETDATA, and ORG flag is set to 2. If ORG flag variable contains 2, then the object code buffer pointer (S) follows the increment of the Program counter to a new location.

7.4.2 EQU Operation

This directive assigns the value of the address field to the name

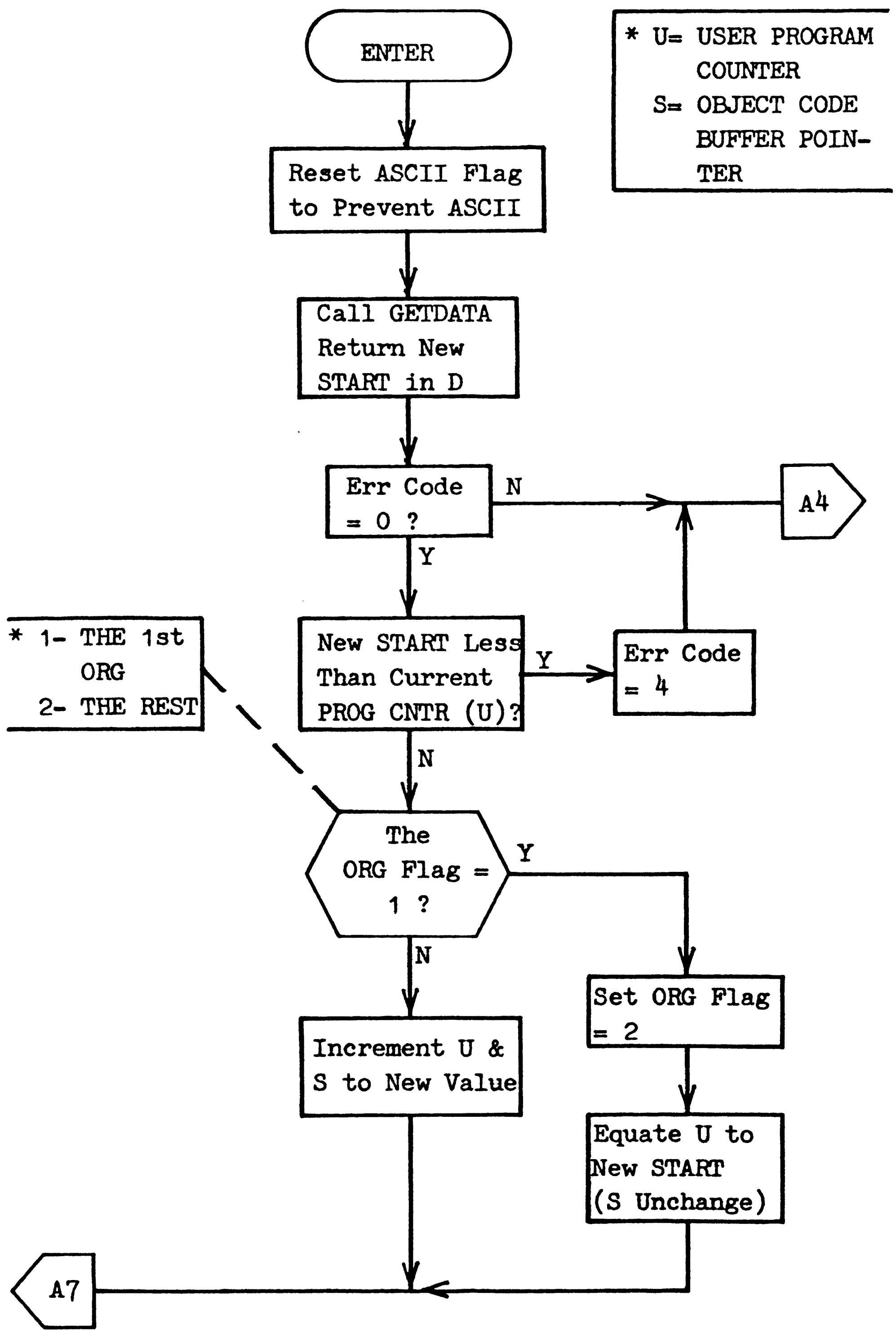

FIGURE 7.14 Flowchart for ORG Operation

specified in the label field. Figure 7.15 depicts the flowchart for this operation.

In order to avoid defining the label twice, an EQU operation is executed only in pass 1. Consequently, all the detected errors can only be displayed in pass 1. Therefore, the found error codes are modified to syntax error code before exiting the routine. The address field may take all forms described in section 7.1.4, but only one ASCII character is allowed. If the subroutine GETDATA returns ASCII data, the Program counter and the object code buffer pointer are decrement by one to eliminate the increment in GETDATA. Since the name in the label field was defined to the current value of the Program counter in the first field scanning, the EQU operation redefines this name to the value returned by GETDATA.

7.4.3 DS Operation

The DS directive orders the Assembler to reserve a number bytes specified by the value in the operand field. The operation simply increments the Program counter and the object code buffer pointer by the value obtained at the subroutine GETDATA.

ASCII and negative data are not permissible. The execution sequence of this operation is depicted in Figure 7.16.

7.4.4 DW Operation

The DW directive stores a list of words into the object code buffer. The 16-bit values (one word=two bytes) are located starting at the current setting of the object code buffer pointer. Each word

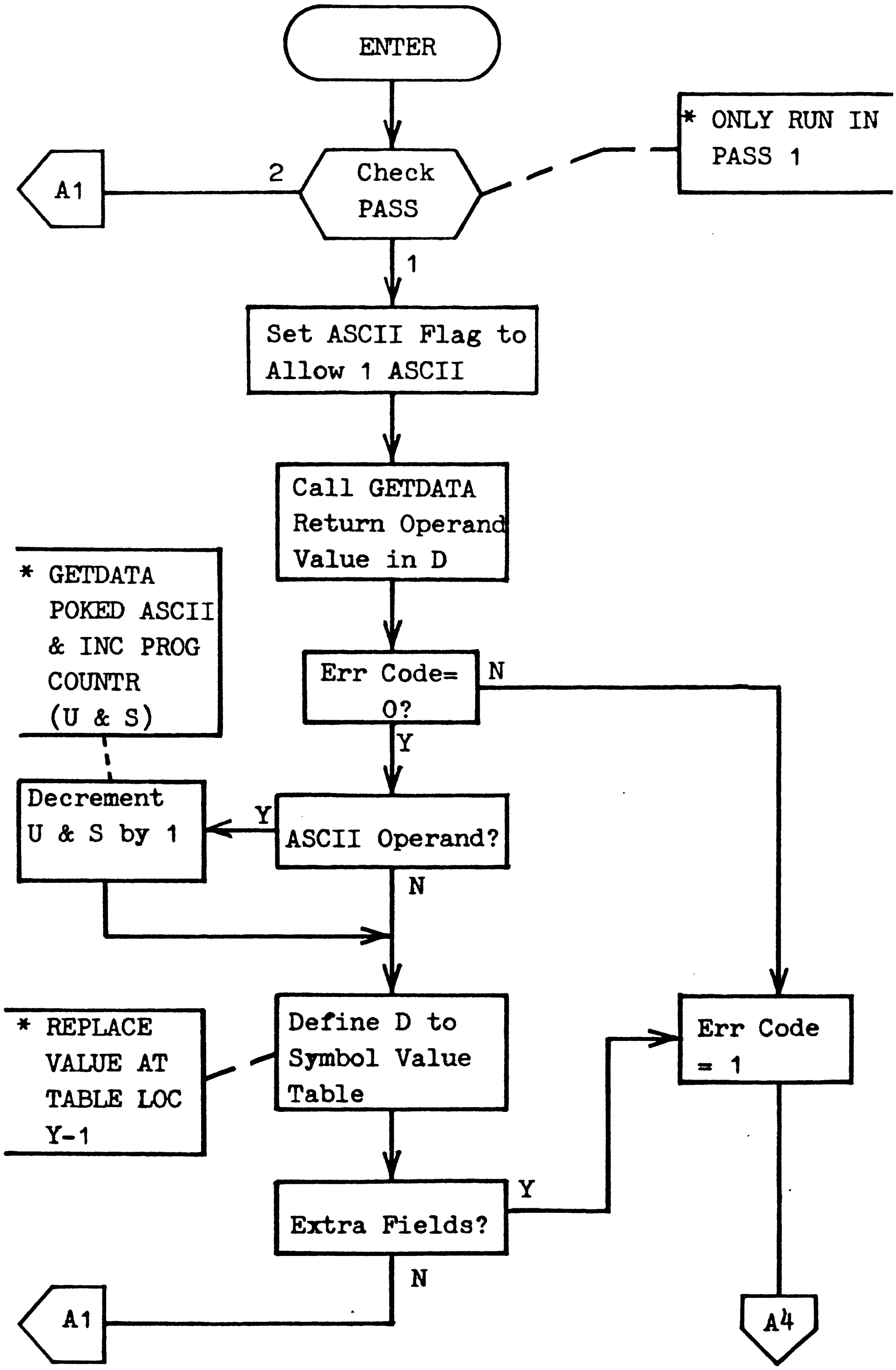

FIGURE 7.15 Flowchart for EQU Operation

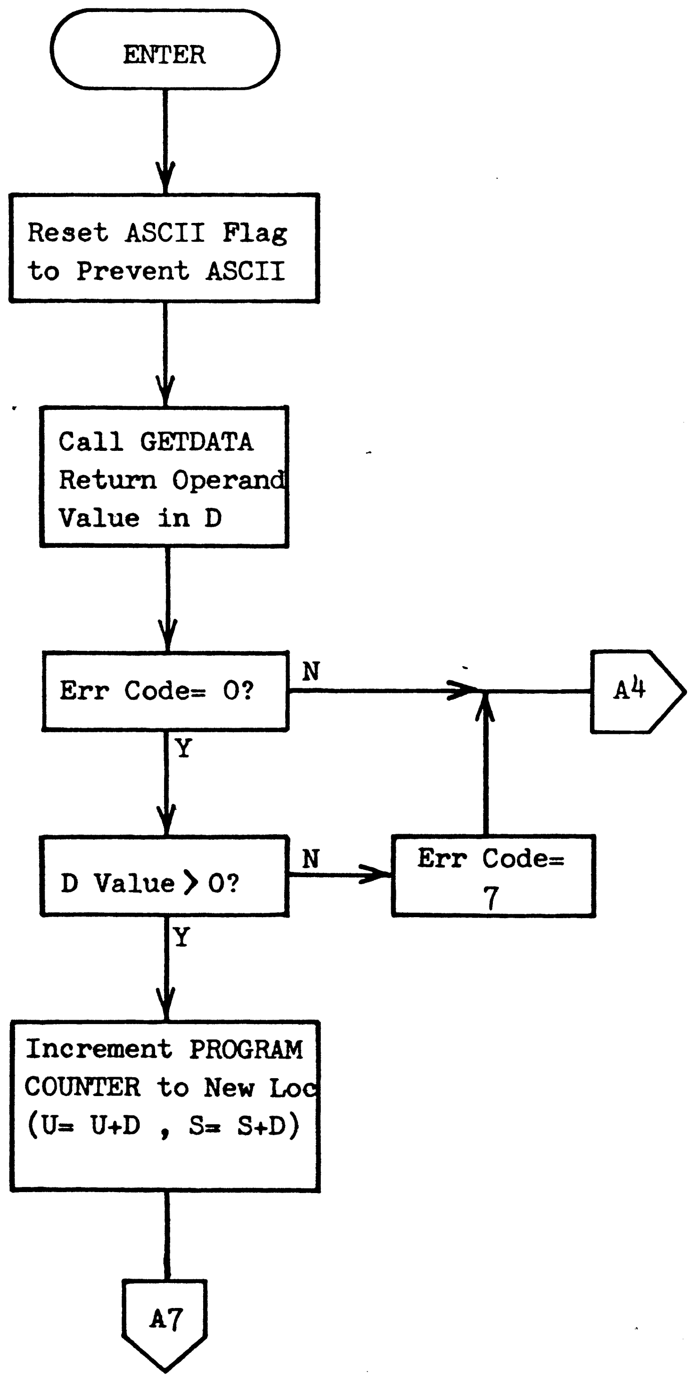

FIGURE 7.16 Flowchart for DS Operation

in the operand field is separated by either a comma, space, or colon. The words may be presented by all forms but ASCII. If the value returned by GETDATA subroutine exceeds the range (-2048 to 65535), the illegal value error is generated.

As illustrated in Figure 7.17, the execution logic is looped until all words are stored or an error is detected. There is no length limit set by this operation, but the Editor can accept a source line up to 256 characters only.

7.4.5 DB Operation

The DB directive stores a list of bytes into the object code buffer. The bytes are located starting at the current setting of the object code buffer pointer. Each operand value is returned by the subroutine GETDATA. The legal range for a 8-bit value is from -128 to 255. Unlike DW, DB also handles a string of ASCII characters enclosed in quotation marks. As aforementioned, the ASCII string is converted and stored by GETDATA.

Figure 7.18 depicts the flowchart for DB operation. As for DW operation, there is no limit on the length of the list. Each item on the list is separated by either a comma, space, or colon. An ASCII string is treated as one item.

7.5 Subroutines

As may be noted in previous sections, several processes are implemented by calling the proper subroutine. Here only certain important subroutines are discussed. Others can be reviewed in

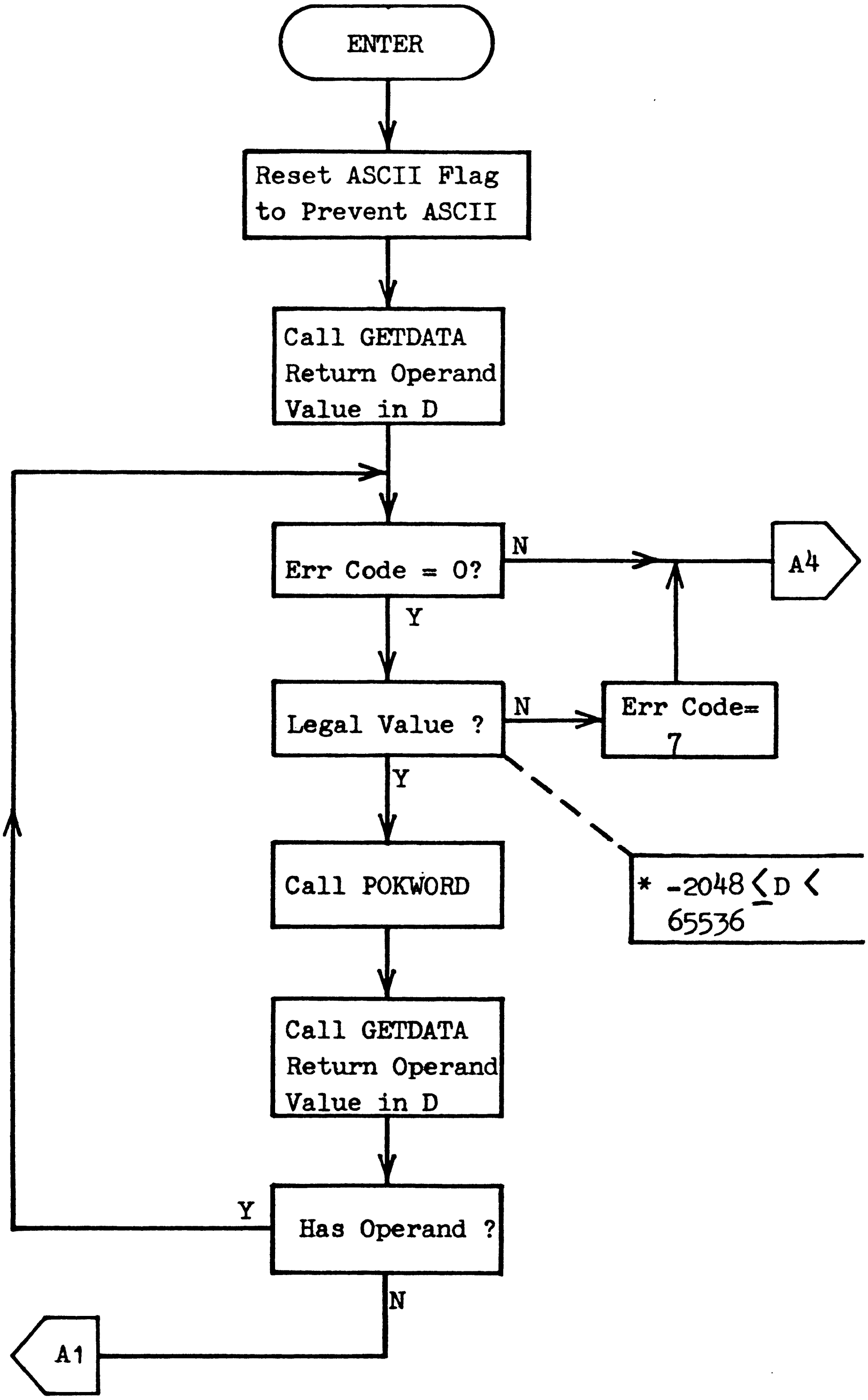

FIGURE 7.17 Flowchart for DW Operation

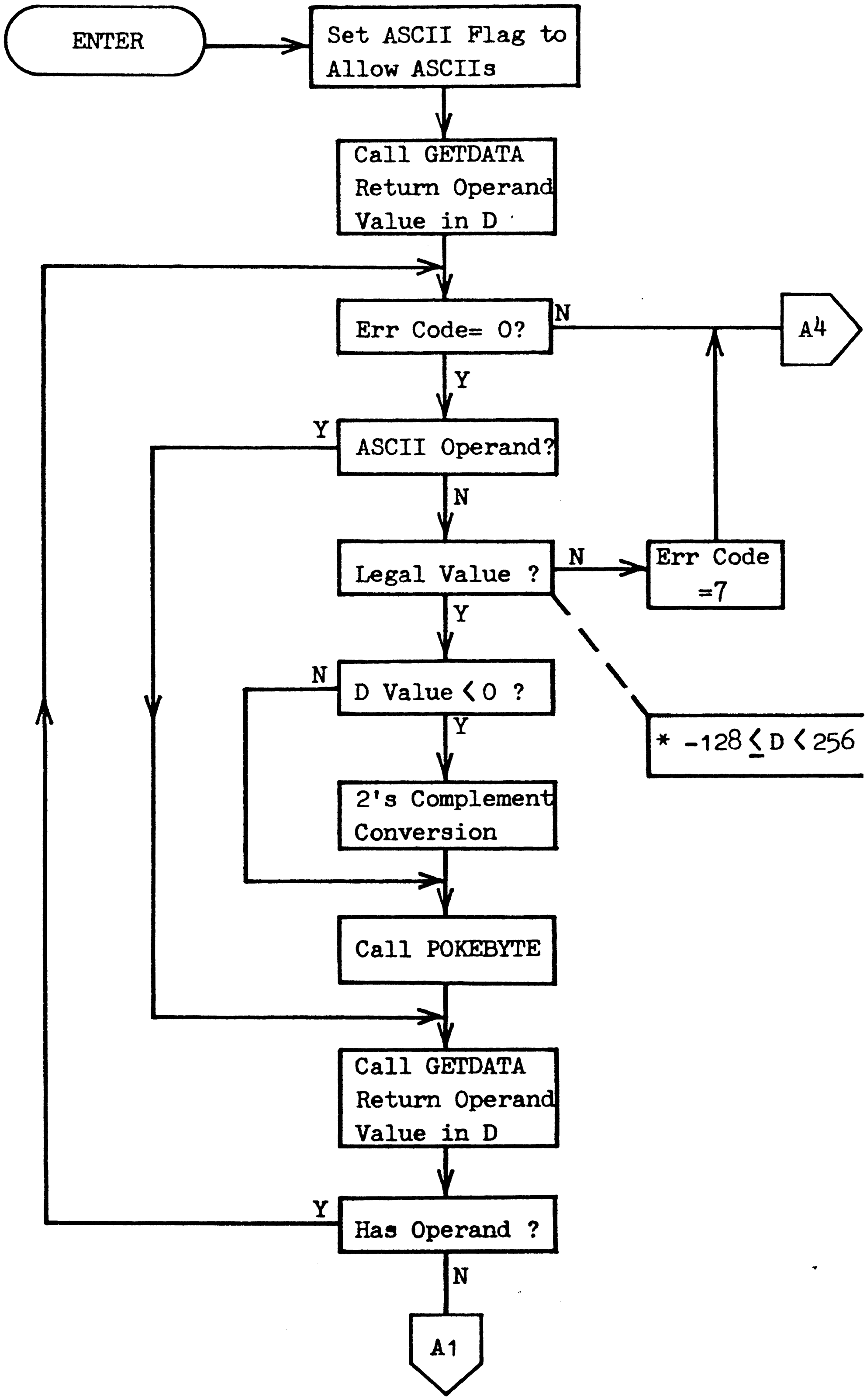

FIGURE 7.18 Flowchart for DB Operation

details by referring to the Assembler program in the Appendix.

7.5.1 ISOLATE Subroutine

This subroutine is the field scanner of the Assembler program. Whenever a field is to be isolated from the source line, ISOLATE is called. Figure 7.19 shows the flowchart for this subroutine.

The scanning pointer, X, is initialized to point to the start of a source line when that line is recovered into the workspace. X then is managed by ISOLATE to indicate the next start scanning position. As illustrated in the flowchart, ISOLATE starts with checking if the line ends. Then it starts searching a valid field starting character. An alphanumeric character, a quotation mark (indicates ASCII), and a minus sign are the valid field starting characters. Once ISOLATE hits one of these characters, the position of that character is marked in variable K, and execution logic begins searching for any delimiters. Either a comma, a space, a colon, or line ends stops the searching. X now points to the stop position. Then ISOLATE collects the substring starting from position K through X-1 in variable G$ for returning. If ISOLATE cannot find a valid character to start, the error code 1 is returned.

7.5.2 GETDATA Subroutine

Another frequently called subroutine is GETDATA, which scans the address/operand field and returns the interpreted decimal value in D. As mentioned, data in the address field may be presented in the following forms: a symbol, ASCII string, hexadecimal representation,

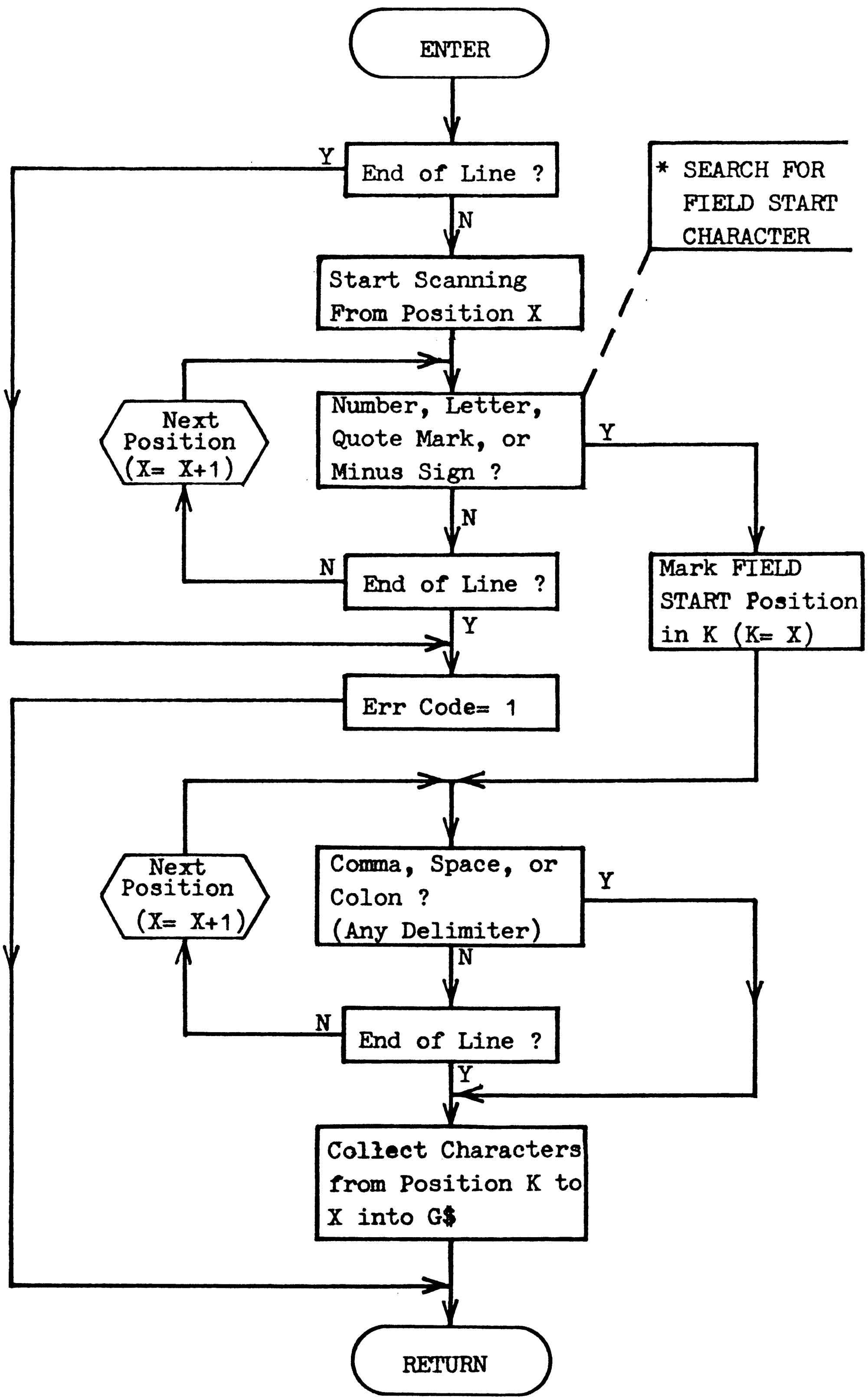

FIGURE 7.19 Flowchart for Subroutine ISOLATE

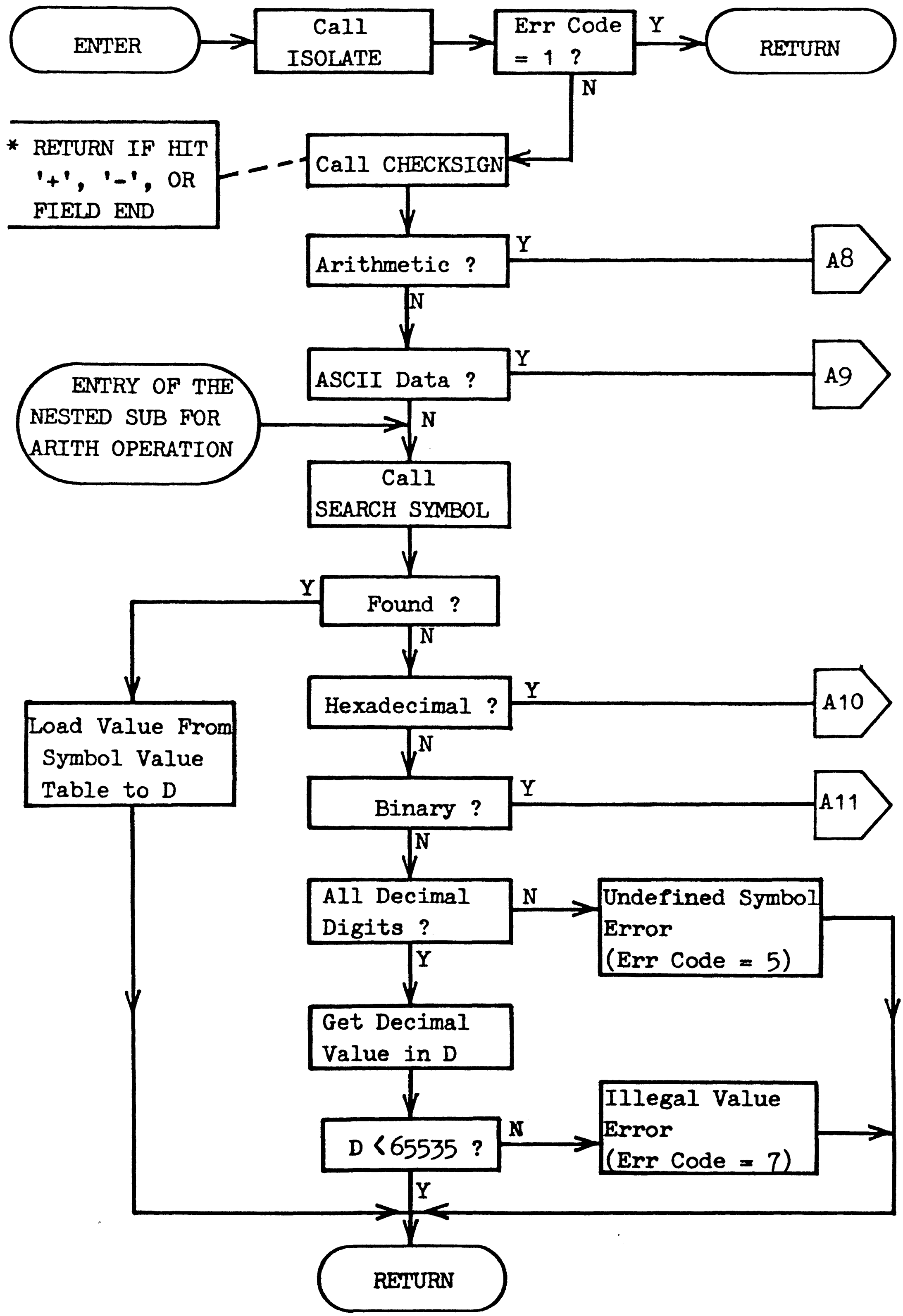

FIGURE 7.20 Flowchart for Subroutine GETDATA

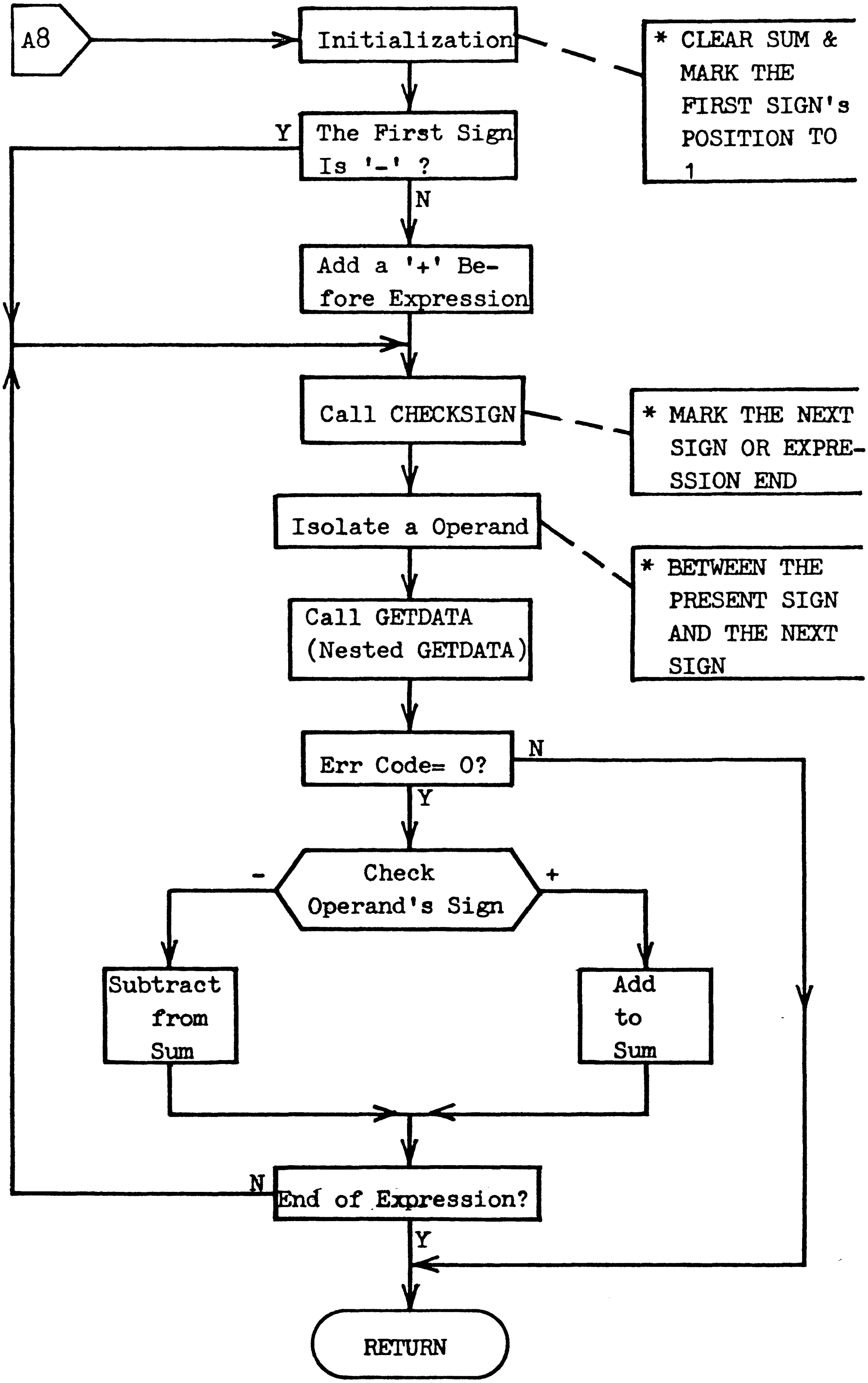

FIGURE 7.21 Flowchart for Arithmetic Operation

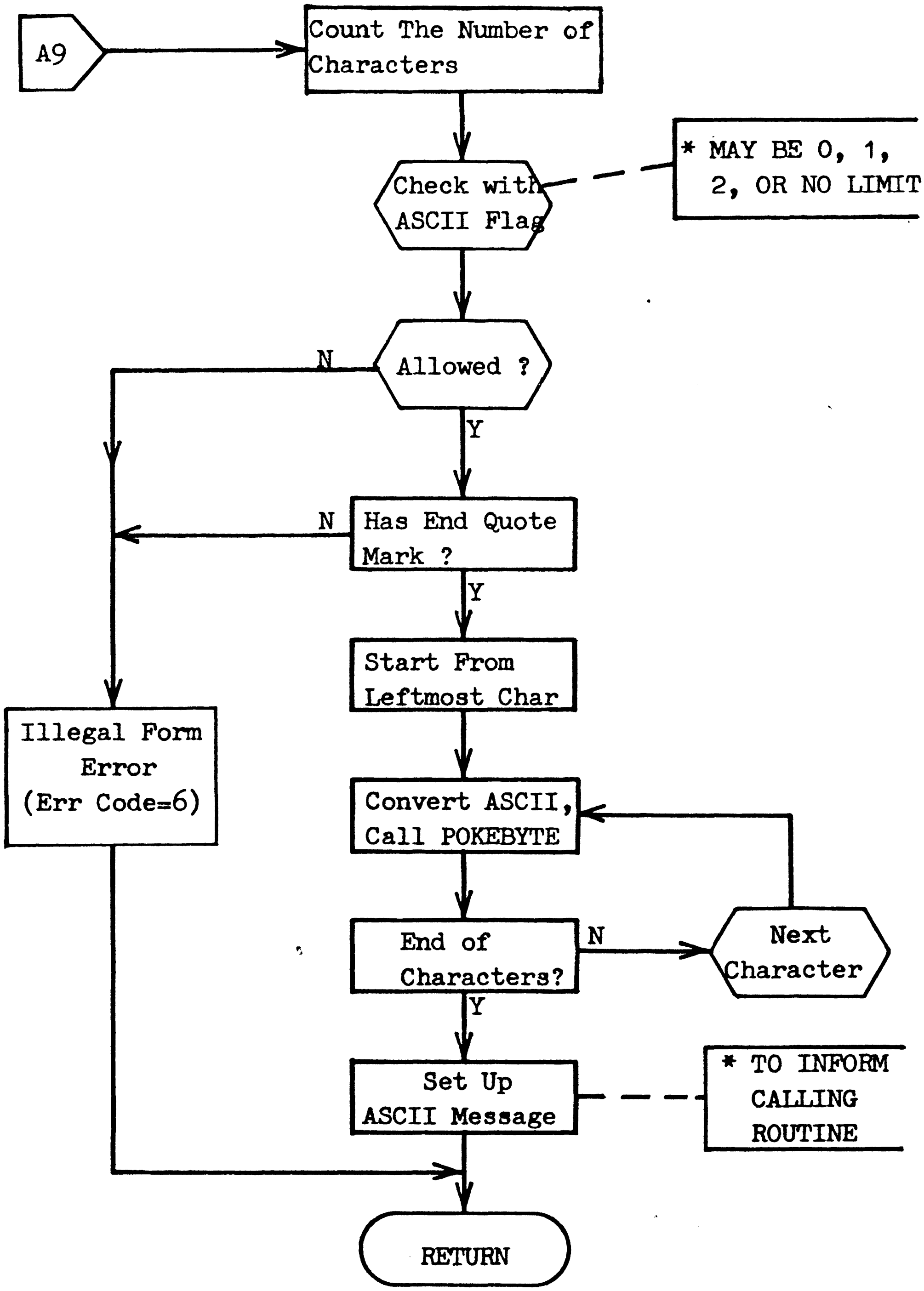

FIGURE 7.22 Flowchart for ASCII Operation

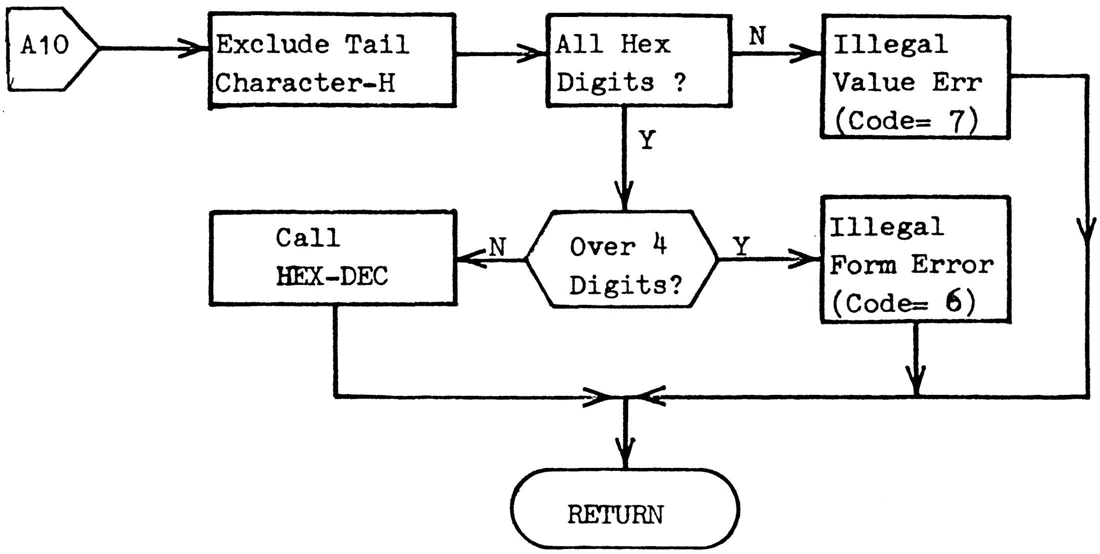

FIGURE 7.23 Flowchart for Hex Operation

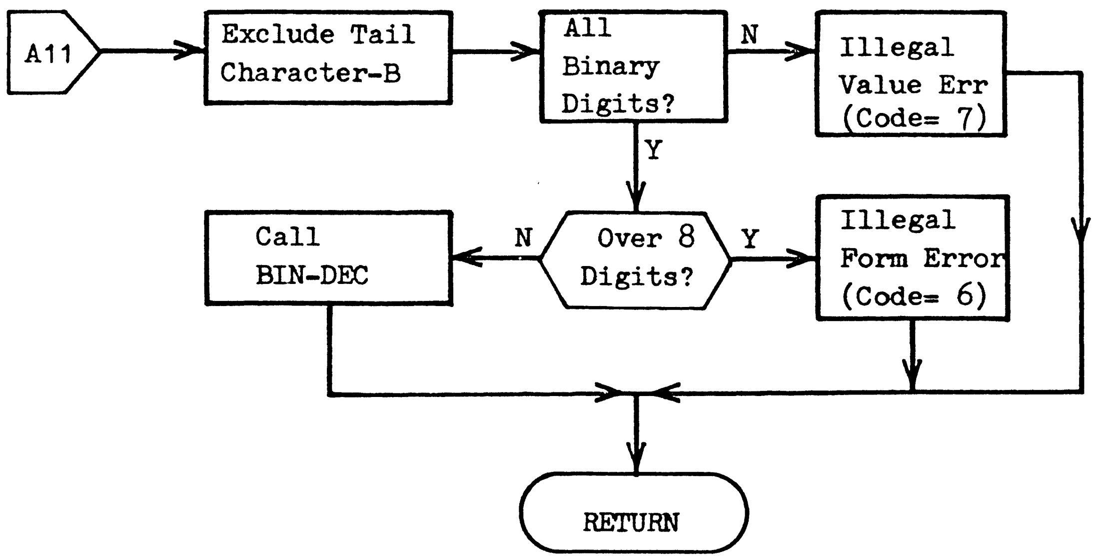

FIGURE 7.24 Flowchart for Binary Operation

decimal digits, binary representation, or arithmetic expression. The main logic of GETDATA leads the execution flow to the proper branch procedure..

As shown in Figure 7.20, GETDATA starts by calling the subroutine ISOLATE to collect an operand field. If no valid character is found, GETDATA returns the execution control to the calling routine. Otherwise the data classification is proceeded. The data classification process is executed in the following sequence: check if arithmetic, check if ASCII, check if symbol, check if hexadecimal, check if binary, check if decimal. Figure 7.21, 22, 23, 24 present the corresponding data operations. If the execution logic cannot classify data in any of the above categories, the error code is defined to UNDEFINED SYMBOL ERROR.

7.5.3 POKWORD and POKEBYTE Subroutines

The subroutine POKEBYTE dumps the entered byte value D to the object code buffer location specified by the pointer, S. Then POKEBYTE increments both Program counter (U) and object code buffer pointer (S) to the next address. The program sequence of POKEBYTE is listed in Figure 7.25.

```
4700 REM  Subroutine POKEBYTE
4710 POKE S,D : REM  Dump byte
4720 S=S+1 : U=U+1
4730 RETURN
```

FIGURE 7.25 Execution Sequence of Subroutine POKEBYTE

The subroutine POKWORD converts the entered value D to two

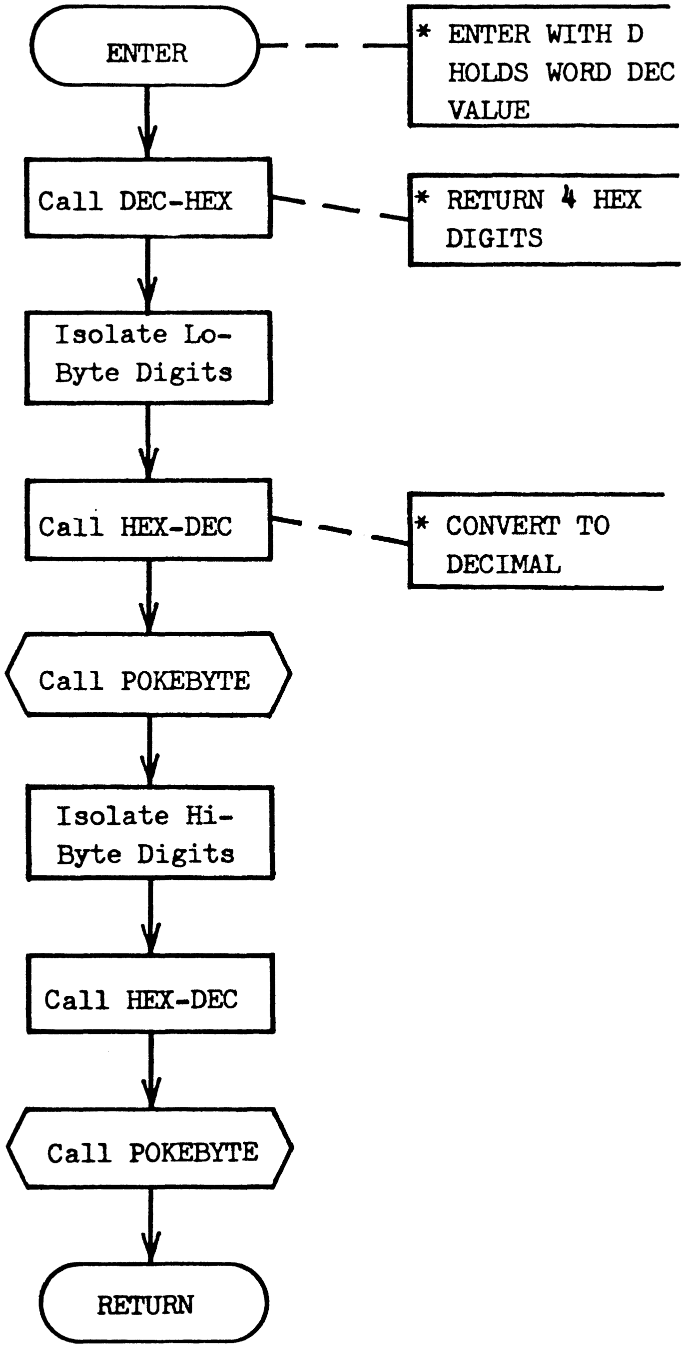

FIGURE 7.26 Flowchart for Subroutine POKWORD

decimal equivalent bytes and stores these two bytes to the object code buffer. This subroutine starts with calling the subroutine DEC-HEX to convert the decimal value D to an equivalent 4 digits hexadecimal representation. DEC-HEX subroutine will convert the negative decimal entry to the equivalent 2's complement form. Then POKWORD takes the low-byte of the returned hexadecimal representation and calls HEX-DEC subroutine. HEX-DEC returns the decimal equivalent value in D. Next, POKEBYTE subroutine is called to load this low-byte value to object code buffer. Similar procedures, as shown in Figure 7.26, are implemented by POKWORD to store the high-byte value to the next buffer location.

7.6 The Listing Program

As mentioned before, the Assembler main program does not have the capacity to install the listing operation. Therefore, this program is developed to perform the listing function for the Assembler. It is stored on disk under the file name SCRIBE. This program is loaded to workspace and executed only if no error was detected by the Assembler.

Since the only reference that can be passed from the Assembler is the object code file, SCRIBE re-establishes the symbol table for its own use. Each source line is recovered and scanned before displaying. The format of line displaying is divided into the following fields: the address field, the opcode field, the data field, the source statement field. After printing the file, the symbols and the corresponding hexadecimal values are listed in the

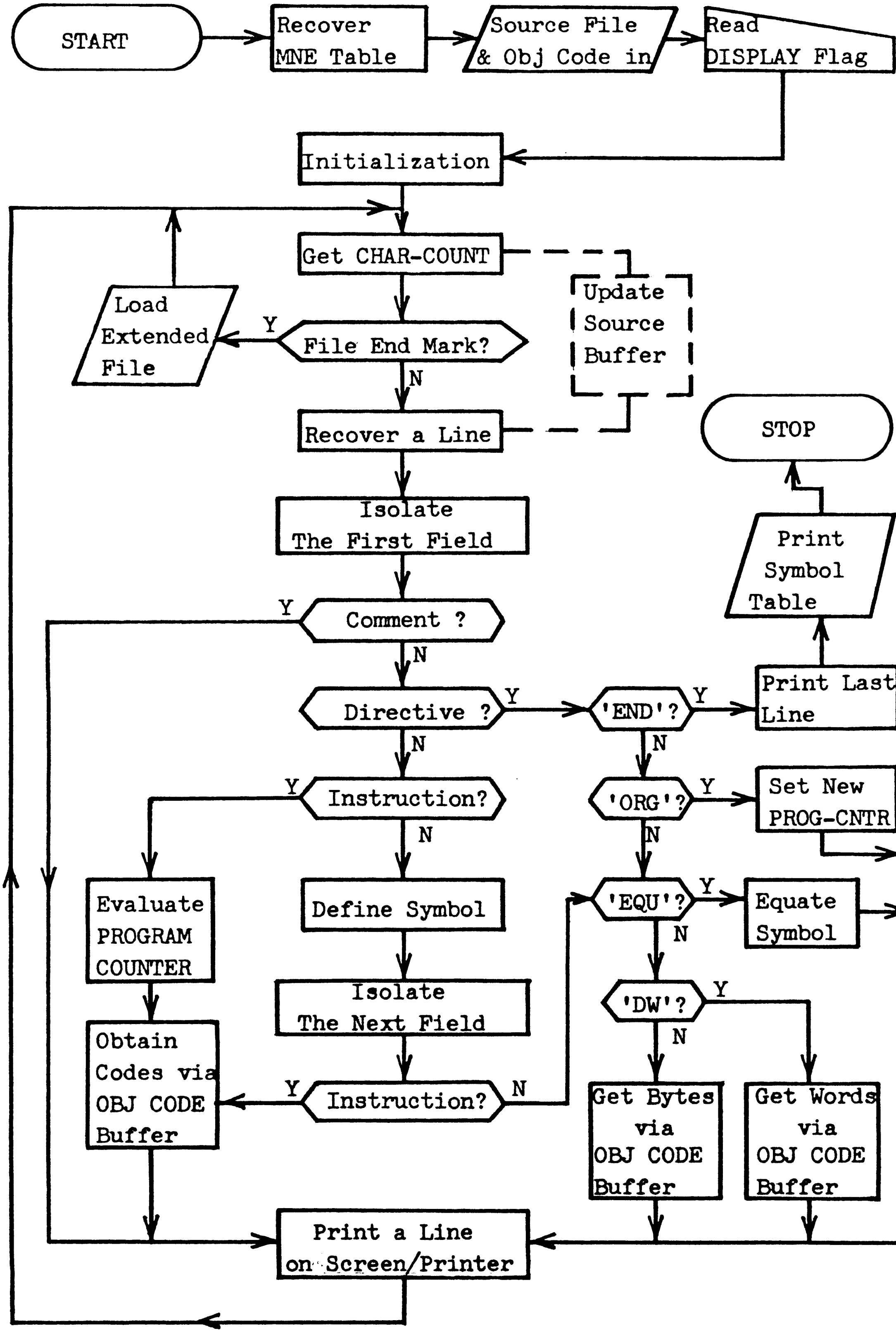

FIGURE 7.27 Generalized Flowchart for File Listing Program SCRIBE

form of five sets per row.

Instructions, DB and DW directives generate object codes. SCRIBE processes these operation codes by obtaining byte/word from the proper location of the assembled object code buffer. Therefore, only the instruction and directive tables are restored from disk. Like the Assembler main program, the program counter and object code buffer pointer are evaluated follow each operation in order to record the data code location and rebuild the symbol references.

Before SCRIBE performs the listing work, it prompts the user and reads a user defined display flag. This flag guides the listing logic to send the file to either the screen or the printer. After the listing work is completed, the execution logic interrogates the user to determine transferring control to the Extended Monitor or the System Executive program.

Figure 7.27 presents a generalized flowchart to depict the execution sequence for SCRIBE program.

CHAPTER 8 SYSTEM OPERATIONS

In this chapter, the operation procedures of this software system are explained by demonstrating the typical processing sequence of a simple example program. This example source program will be entered by using the Editor, and will be converted to an 8085 machine language program by the Assembler. Then the Extended Monitor will be employed to file this object code program, and send this program to the SDK-85 for execution. Those procedures of how to obtain information from the SDK-85 and how to modify the program also will be illustrated.

The source program in Figure 8.1 is the example program to be demonstrated. It calculates the sum of a series of data bytes. The length of the series is in location labeled LENGTH and the series itself starts in location next to LENGTH. The sum is stored in the hexadecimal address 2000. This addition program ignores carries.

8.1 Initialization

To start this operation, both SDK-85 and OSI-C4PMF systems first must undergo hardware initialization. After power up the SDK-85, the user should press the EXEC key followed by entering the hexadecimal address 8227 to enable the data communication program. When the SDK-85 is controlled by this program, an 'E' is shown in the leftmost digit of the LED display. Then, a diskette contained the system programs must be inserted into disk drive A of the OSI-C4PMF computer. Upon pressing the BREAK key, the OSI prompts the message

```
         ORG   9000H
         LXI   H,LENGTH      ; Points to LENGTH
         MOV   B,M           ; B = data counter
         SUB   A             ; Clears A
  NEXT:  INX   H             ; Points to data byte
         ADD   M             ; Addition
         DCR   B             ; Data end ?
         JNZ   NEXT          ; No, adds the next byte
         STA   2000H         ; Stores the sum
LENGTH:  DB    2             ; 2 data bytes follows
         DB    01H, 02H      ; Data bytes
         END
```

FIGURE 8.1 An Example Program

```
1       ; Addition of a string of data bytes
5       ;
10               ORG   9000H
15               LXI   H,LENGTH      ; Points to LENGTH
20               MOV   B,M           ; B = data counter
25               SUB   A             ; Clears A
30        NEXT:  INX   H             ; Points to data byte
35               ADD   M             ; Addition
40               DCR   B             ; Data end ?
45               JNZ   NEXT          ; No, adds the next byte
50               STA   2000H         ; Yes, stores the sum
55      LENGTH:  DB    2             ; 2 data bytes follows
60               DB    01H, 02H      ; Data bytes
65               END
```

FIGURE 8.2 Source File of the Example Program

'H/D/M' on the screen. D selects the disk operation and boots the DOS from the disk. The DOS then loads the System Executive program to workspace, and executes this program to provide the following menu display.

```
FUNCTIONS AVAILABLE:

    (1) EXTENDED MONITOR - INTERCHANGE, MODIFY, & FILE DATA
    (2) EDITOR - EDIT THE 8080/8085 SOURCE LANGUAGE FILES
    (3) ASM85 - ASSEMBLE THE 8080/8085 SOURCE LANGUAGE FILE
    (4) FREE - FREE SYSTEM FOR USER PROGRAMMING

SELECT FUNCTION (1-4)?
```

The user may select the desired operation by entering the corresponding numerical digit. Any entry that fails to fall into the range from 1 to 4 will cause this menu to be displayed again. If the user intends to exit the developed system, the FREE function may be selected. When the following message is displayed, the workspace is cleared and the DOS is ready to accept the BASIC language programming or a DOS command.

```
SYSTEM FREE
11645 BYTES AVAILABLE
OK
```

8.2 Edit Source File

To enter and to edit the source file of the example program, the numerical key "2" specifying the edit operation is pressed. The Editor program then is loaded and executed. A message 'Command?' prompts the user for a command entry. As mentioned in Chapter 6, all of the Editor commands can be abbreviated to one letter. For entering the input mode, an "I" keyboard entry is issued. When the input mode prompt '?' is displayed on the screen, the Editor is ready to accept a source line input.

The program is entered line by line with a non-zero decimal number at the start of each line. These numbers represent the sequence of the program statements. Before pressing the RETURN key to end a line, the SHIFT-O can be used to delete the preceding one character. After the input mode prompt ('?'), another source line can be typed or a command can be entered to exit the input mode. Suppose the form of this program is entered as shown in Figure 8.2. In order to reserve the insertion capability, the line-increment value must be at least greater than one. In this demonstration, the line-increment is five. After having all lines entered, the user may want to examine this entered source program on screen, or obtain a hard copy from printer. To do so, the user simply types an "L" for screen listing, or a "P" for printer output. The user may specify the range of displaying by entering the line specification following the command syntax. These commands make the Editor exit the input mode and to perform the specified command.

Before exiting the Editor, the source file just entered must be filed to disk. This can be done by typing "F". The filing speed is 0.52 second per line. Since this example program did not exceed the buffer capacity, no extended file is needed. If in the input process a 'Buffer ends at line XXX' message is displayed on the screen, this means the program is too large and line XXX is the last line. The user may then use the extended file mode to accommodate the rest of the lines. To enter the extended file mode, the user should file the current file to disk, then type "E". Command NEW ("N") will clear the extended mode.

Now, the example program source file is in the disk. The Editor then can be exited by typing "Q" (QUIT). This makes the menu selections to reappear on the screen.

8.3 Assemble Source File

To assemble the source program, a "3" is entered to select the Assembler operation. Before the assembling process begins, the Assembler program sends the following message to interrogate the user.

```
List errors on printer instead of screen (Y/N)?
```

After reply, the Assembler starts translating the source file at the rate of 30 lines per minute, and the following messages will be shown to indicate the processing status.

```
This is a slow assembler!
```

```
Begin assembling .....
 0 errors in PASS 1
Continue PASS 2 .....
End assembling.  Total 0 errors.
```

These messages indicate the case of error free. If any errors are detected, a proper error message will be sent. For example, if there is a syntax error in line XXX, the message will be:

```
Error #1 in line XXX
```

In the case of errors, the last message sent by the Assembler is:

```
Go back to Editor for corrections (Y/N)?
```

In the case of error free, the last message is:

```
Do you want a completed listing (Y/N)?
```

In both cases, a "N" entry causes the menu selections to reappear on the screen. If the user selects the listing function, the succeeding question is:

```
List on printer instead of screen (Y/N)?
```

Either listed on printer or screen, the assembled example program will be listed as shown in Figure 8.3. After the listing work is completed, the following message is:

Do you want to go to the Loader (Y/N)?

If the reply is "Y", then the Extended Monitor will be enabled. Otherwise, the menu selections will be raised.

8.4 Operations of Extended Monitor

In order to communicate with the SDK-85, Extended Monitor function is selected. When this program is loaded and executed, the user should see the welcome messages as followed:

```
    *** SDK-85 EXTENDED MONITOR ***

Current data in buffer are released by the Assembler
Simulated SDK-85 Memory Starting Address - 9000
                          Ending Address - 9010

Command?
```

The object code file of the example program now resides in the Extended Monitor simulated SDK-85 memory buffer area. These boundary addresses can be changed to simulate another portion of the SDK-85 memory by using the SE command. The SE command may alter the range but has no affect on the contents in the range.

8.4.1 Insertion of an RET

As mentioned, in order to regain control of SDK-85, an RET

```
8080/8085  CROSS ASSEMBLER,    RELEASED 1982.    E.E. OHIO U.

ADDR OP DATA  SEQ      SOURCE STATEMENT

               1      ; Addition of a string of data bytes
               5      ;
9000          10              ORG  9000H
9000 21 0E90  15              LXI  H,LENGTH      ; Points to LENGTH
9003 46       20              MOV  B,M           ; B = data counter
9004 97       25              SUB  A             ; Clears A
9005 23       30       NEXT:  INX  H             ; Points to data byte
9006 86       35              ADD  M             ; Addition
9007 05       40              DCR  B             ; Data end ?
9008 C2 0590  45              JNZ  NEXT          ; No, adds the next byte
900B 32 0020  50              STA  2000H         ; Yes, stores the sum
900E    02    55     LENGTH:  DB   2             ; 2 data bytes follows
900F    01    60              DB   01H, 02H      ; Data bytes
9010    02
              65              END

SYMBOL TABLE:

NEXT   9005  LENGTH 900E
```

FIGURE 8.3 Listing File of the Example Program

(Return-from-subroutine) instruction should be installed at the end. In editing the example program source file, this instruction was not included. Therefore, an insertion is needed. By examining the listing printout in Figure 8.3, the RET instruction should be placed at address 900E. This means those data bytes starting from the labeled address LENGTH through the end must be moved down one location. To do this, the IN command (INSERT) first can be used. By typing "IN 900E/1", the data block is relocated and the address 900E is available to enter the opcode of RET. To enter this opcode into address 900E, the command statement "SU 900E/C9" is employed, and the followed message is:

Substitute 900F?

Since only a byte is to be entered, the reply should be simply a "N". A 2-digit hexadecimal input will replace the contents of address 900F, and a similar message for the succeeding substitution will be displayed.

Because the address of LENGTH is changed to 900F, the corresponding contents of address 9001 must also be modified to 0F by using the same procedure just demonstrated.

One would normally put RET into the original source program.

The modified object codes can be examined by screen display or printer output. The command EX (EXAM) selects the screen; the command PR (PRINT) selects the printer. If the user issues the PR command without address specification followed, the whole object code program will be sent to the DECWRITER printer. Before printing this file, the following message is asked.

```
Do you need a title (Y/N)?
```

If the reply is YES, then the next question is:

```
Title?
```

Suppose the title is given as "OBJECT CODE LISTING OF THE ADDITION PROGRAM". Then the printout from the printer will be shown as following:

```
OBJECT CODE LISTING OF THE ADDITION PROGRAM:
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
9000 21 0F 90 46 97 23 86 05 C2 05 90 32 00 20 C9 02
9010 01 02
```

As noted, the last address is extended to 9011. In order to confirm that the simulated memory range covers this expansion, the SE command can be used. The SE command raises the following messages:

```
Simulated SDK-85 Memory Starting Address - 9000
                          Ending Address - 9011

Change Starting Address?
```

The message verifies that the previous insertion extended the boundary to include address 9011 already. Therefore, no change needs to be made. A "N" entry leads the execution to escape the present function.

8.4.2 Save Object Code File

Before sending this modified object code program to the SDK-85 for execution, the user may wish to save this program to disk. The user may use any created filename in the directory, or may create a new filename. However, the CR command must be involved. This command will display the current directory and will allow creation of new filenames. For instance the directory messages are:

```
-- DIRECTORY --

LOC.  FILE NAME
1     CHECKIN
2     APPTEST
3     KEY
4     ???
5     ???

Are you sure (Y/N)?
```

If the user simply want to check the directory, the above question helps the user to escape creation of filename. If the user intends to create a filename for the example program, then the succeeding question is:

```
Enter new file name?
```

Suppose, the example program is named ADDITION. After entering this filename, the followed question is:

At which storage location (1-5)?

As noted, locations 1 through 3 already have names, and locations 4 & 5 are undefined. The user may select any location. For those defined locations, this will be a rename process. For the two no-named locations, this will be a creation process. Suppose the location 4 is selected. The updated directory will be displayed as following:

```
-- DIRECTORY --

LOC.  FILE NAME
1     CHECKIN
2     APPTEST
3     KEY
4     ADDITIO
5     ???

Create another file (Y/N)?
```

As noted, the created filename ADDITION is placed into location 4, but only the leftmost seven characters were defined. The user may create or rename another filename by typing "Y".

The example program ADDITION now is ready to be stored to disk file location 4 under the filename ADDITIO. The user is able to save this program by typing "SA ADDITIO".

8.4.3 Load Program to SDK-85 for Execution

The next step is to load this example program to the SDK-85 resident memory for execution. Since the range of the simulated

SDK-85 memory has not been altered, the loading operation can be done by simply entering "DU" (DUMP command) without address specifications. The contents of the current simulated memory then will be loaded to the corresponding SDK-85 resident RAM locations. When the prompt "Done" is displayed, the program is loaded.

To order the SDK-85 to execute this program, the RU command (RUN) must be employed. Either "RU" or "RU 9000" will command the SDK-85 to execute that program. Since this example program is not a looping structure and is equipped with an RET, the data communication channel is still maintained after the program is executed.

8.4.4 Get Result from SDK-85

As noted, this example program ADDITION stores the sum to SDK-85 location 2000. The current simulated SDK-85 memory does not cover this address. It is therefore necessary to set a new pseudo memory range. After using the SE command to define a new boundary to include the address 2000, the GE command (GET) then can be issued. Suppose the new simulated memory range is set to 2000-2010. Upon the information is received, the result may be examined by typing "EX 2000-2000" to display only that byte on the screen.

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
2000 03
```

8.5 Modify Program

The example program just executed performs the addition of two numbers. As noted from the structure of this program, it can be modified to calculate more numbers by changing LENGTH and adding data bytes. This may be accomplished in two ways.

The first way is to use the Editor to modify the source program. To do this, first, the user should type "QU" to exit the Extended Monitor, then, select the Editor when the menu selection appears. After entering the Editor, the source file can be retrieved by issuing the C command (CALL). The example source program will be loaded to the buffer at the average speed of 0.9 second per line. When the Editor prompts 'Done', the user can use the I command to enter the input mode. The newly entered statement will replace the same numbered statement in the file. After the proper lines are entered, the user should file the modified source program to disk, then exit the Editor and select the Assembler to assemble this file. Those procedures of re-entering the Extended Monitor and Loading program to SDK-85 are the same as mentioned before.

The other way is to modify the object code file directly. Since the object code file of the example program had been filed to disk, the command statement "LO ADDITIO" entry will retrieve that file. After the file ADDITIO is loaded, the screen will show the following messages:

```
Simulated SDK-85 Memory Starting Address - 9000
                          Ending Address - 9011
```

The SU command now can be used to substitute and enter contents at proper locations. Following those loading and executing procedures described in the previous subsections, this modified program then can be executed in the SDK-85.

Those operations which are not demonstrated above can be reviewed in the chapters of the Editor and the Extended Monitor description.

CHAPTER 9 SUMMARY AND FUTURE DEVELOPMENTS

9.1 Summary

The goals established at the start of this project have been accomplished. In the SDK-85, the resident RAM has been expanded to accommodate a larger user program. A data communication circuit has been constructed on the SDK-85 board for serial interfacing with the OSI-C4PMF system. The communication control program has been developed in the expanded EPROM memory to co-operate with the host system to implement the user specified operation. In the host system, OSI-C4PMF, a cross-assembling and file managing system for the SDK-85 has been written and installed. This software system includes the Text Editor, the 8085 Cross Assembler, and the SDK-85 Extended Monitor. The Editor provides the functions for editing the source assembly language file. The Assembler translates the source codes to the 8080/8085 machine code program. The Extended Monitor performs the data interchanging with the SDK-85 and supplies the data modifications, and the binary file maintenance capabilities. Through the assistance offered by this enhancement system, the user now is able to manage the operation of the SDK-85 microcomputer more efficiently and conveniently.

This development provides a model of using a DOS-based personal computer to enhance a kit computer's operating capabilities without extensive resident hardware and software expansion. Except for the assembly language programs and the DOS command statements, the BASIC language programs (Editor/Assembler/Extended Monitor) are machine-independent, and can be executed on other personal computers.

9.2 Future Developments

Although the present version of the developed system uses almost all of the memory and disk space, it is still possible to advance the operation capabilities. The following sections provide both hardware and software enhancements that can be developed in the future expansions.

9.2.1 Double-Disk System Expansion

The present software developed is a single-disk operation system. The operating programs and the user files are both on one diskette. It is possible to make minor software modifications to expand the system to a double-disk operation system.

To support this, the DOS commands, DISK!"SELECT A" and DISK!"SELECT B", can be used in the BASIC program to guide the disk access to drive A or B respectively. One may construct the system so that the system programs can be read from disk drive A, and the user file information can be retrieved from disk drive B. Since track 0 through 9 are reserved by DOS, a total of thirty tracks can be accessed by the DOS commands CALL and SAVE. Excluding the tracks used by the user assembly language source file, object code file, and directory, twenty four user binary files can be installed on the user file diskette. To initiate this operation, a command INIT, which will format a file disk, may be added to the Extended Monitor program. On the system program disk, those tracks which were used to store the user files, then are available to develop other utility programs to enhance the capability of the system operation.

9.2.2 Hardwired Interrupt

Another major improvement can be scheduled in the future is to install the hardware RESET function for the Extended Monitor. As described in Chapter 3 and Chapter 5, the RUN command causes an user specified program to be executed in the SDK-85. If the specified program is a looping structure or has no RET instruction at the end, the user loses control of SDK-85. To improve this, the hardwired interrupt of the SDK-85 can be employed.

The available SDK-85 user interrupt is RST 6.5 which can be accessed at connector J1 of the SDK-85 circuit board. At present, RST 6.5 is disabled and will be available to use after the jumper wire is removed from jumper 3-4. The 8085 RST 6.5 is a high-level sensitive interrupt input. The interrupt signal must be held on for at least 5,770 ns. Therefore, the hardware design could be developed by using a one-shot chip and an inverter to generate a proper timing signal to the RST 6.5 input. The falling-edge trigger signal for the one-shot chip can be fed from the OSI-C4PMF ACIA's $\overline{\text{RTS}}$ output pin or a PIA's control line. To co-operate with the hardwired signal, the SDK-85 communication program must also be modified. Since the vector for RST 6.5 is set to branch to RAM location 20C8, the communication program should place a JMP instruction for re-entry in locations 20C8-20CA during initialization.

In doing so, the Extended Monitor command RESET is able to generate an interrupt to the SDK-85 system for restoring the data communication channel.

REFERENCES

1. Intel Corporation, SDK-85 System Design Kit User's Manual, 1978
2. Intel Corporation, MCS-80/85 Family User's Manual, 1979
3. Ohio Scientific Inc., OSI-C4PMF Challenger User's Manual, 1978
4. MOS Technology Inc., MCS6500 Microcomputer Family Programming Manual, 1975
5. Lance A. Leventhal, 8080A/8085 Assembly Language Programming, Osborne & Associates Inc.
6. Lance A. Leventhal, Introduction to Microprocessors: Software, Hardware, Programming, Prentice-Hall Inc., 1978

APPENDIX A - CROSS ASSEMBLER ERROR CODE INTERPRETATION

CODE		INTERPRETATION
1	-	OPERATION CODE SYNTAX ERROR
2	-	MULTIPLE SYMBOL DEFINITION
3	-	SYMBOL TABLE OVERFLOW (MAXIMUM 100 ENTRIES)
4	-	NON-ASCENDING ORG SEQUENCE
5	-	UNDEFINED SYMBOL
6	-	ILLEGAL OPERAND FORM
7	-	ILLEGAL OPERAND VALUE
8	-	UNNECESSARY/ILLEGAL OPERAND
9	-	NO END DIRECTIVE

APPENDIX B - SDK-85 DATA COMMUNICATION PROGRAM

```
8080/8085  CROSS ASSEMBLER,   RELEASED 1982.  E.E. OHIO U.

ADDR OP DATA  SEQ      SOURCE STATEMENT

                  ;*****************************************************************
                  ;
                  ;                    SDK-85 DATA COMMUNICATION PROGRAM
                  ;                    ---------------------------------
                  ;
                  ;This program resides permanently in the SDK-85 EPROM memory loca-
                  ;tions starting from address 8227H.  It accepts commands from the
                  ;OSI-C4P system, and executes the corresponding command routines.
                  ;
                  ;*****************************************************************
                  ;
                  ;
                  ;Definitions:
                  ;
       8227        BEGIN    EQU   8227H          ; Program starting address
       0057        RESET    EQU   01010111B      ; Pattern for ACIA master reset
       0015       PROGRM    EQU   00010101B      ; Pattern for programming ACIA
       0008       MSKCTS    EQU   00001000B      ; Mask pattern for CTS test
       0001       MSKRRF    EQU   00000001B      ; Mask pattern for RDRF test
       0002       MSKTRE    EQU   00000010B      ; Mask pattern for TDRE test
       008E       STATUS    EQU   8EH            ; ACIA Status Reg.(Read Only)
       008E       CONTRL    EQU   STATUS         ; ACIA Control Reg.(Write Only)
       008F       OSIC4P    EQU   8FH            ; ACIA Transmit/Receive Reg.
                  ;
                  ;
8227                        ORG   BEGIN
                  ;
                  ;
                  ;************************ Main Routine ************************
                  ;
                  ;Initialization
                  ;
8227 31 C220                LXI   SP,20C2H       ; Initialize Stack Pointer
822A 3E 57                  MVI   A,RESET
822C D3 8E                  OUT   CONTRL         ; Master reset ACIA
822E 3E 15                  MVI   A,PROGRM
8230 D3 8E                  OUT   CONTRL         ; Programming ACIA, RTS low
                  ;
8232 DB 8E        NOTYET:   IN    STATUS         ; Status Reg. to A
8234 E6 08                  ANI   MSKCTS         ; Check if OSI ready
8236 C2 3282                JNZ   NOTYET         ; No, check again
                  ;
                  ;Re-entry location for Command Routines
                  ;Wait command input from OSI-C4P
                  ;
8239 CD B682      WACOMD:   CALL  DATAIN         ; Yes, get command in
823C 21 C882                LXI   H,TABLE        ; Set Command Table Pointer
823F 06 04                  MVI   B,4            ; Set Counter
8241 BE             NEXT:   CMP   M              ; Match?
8242 23                     INX   H              ; Point to Command Routine addr HI
8243 CA 4F82                JZ    FOUND          ; Yes, found it
8246 05                     DCR   B              ; Check if end of table
8247 CA 3982                JZ    WACOMD         ; Yes, invalid command
824A 23                     INX   H              ; No, skip address bytes
824B 23                     INX   H
824C C3 4182                JMP   NEXT           ; Try next
                  ;
                  ;Command verification
                  ;
```

```
824F D3 8F       FOUND:  OUT   OSIC4P          ; Send same command to OSI-C4P
                 ;
                 ;Transfer control to the Command Routine
                 ;
8251 7E                  MOV   A,M             ; Load Routine address lo-byte
8252 5F                  MOV   E,A
8253 23                  INX   H
8254 7E                  MOV   A,M             ; Load Routine address hi-byte
8255 57                  MOV   D,A             ; (D,E)= Routine address
8256 EB                  XCHG                  ; Prepare for passing address to PC
8257 E9                  PCHL                  ; Go to execute Command Routine
                 ;
                 ;
                 ;************************ Routine TRANSM ************************
                 ;
                 ;Routine TRANSM transmits a string of data bytes specified by the
                 ;OSI-C4P to ACIA
                 ;
8258 CD 9782     TRANSM: CALL  SETUP           ; Set memory pointer & byte counter
                 ;Start transmission procedure
825B CD C082     NEXOUT: CALL  EMPTY           ; Wait until ACIA ready to transmit
825E 7E                  MOV   A,M             ; Get a data byte
825F D3 8F               OUT   OSIC4P          ; Transmit data to OSI-C4P
8261 CD AC82             CALL  CHKSUM          ; End of data?
8264 C2 5B82             JNZ   NEXOUT          ; No, go for next
8267 CD C082             CALL  EMPTY           ; Yes, wait until ACIA ready
                 ;
                 ;Routine CHECK is shared by TRANSM and RECEIV for checking the
                 ;accumulated checksum
                 ;
826A 78           CHECK: MOV   A,B             ; Get checksum high-byte
826B D3 8F               OUT   OSIC4P          ; Send to OSIC4P
826D CD B682             CALL  DATAIN          ; Get response from OSIC4P
8270 B8                  CMP   B               ; OSI-C4P agree with?
8271 C2 3982             JNZ   WACOMD          ; No, go to waiting for command in
8274 79                  MOV   A,C             ; Yes, get checksum low-byte
8275 D3 8F               OUT   OSIC4P          ; Send to OSIC4P
8277 C3 3982             JMP   WACOMD
                 ;
                 ;************************ Routine RECEIV ************************
                 ;
                 ;Routine RECEIV receives a string of data bytes from the OSI-C4P,
                 ;and locates the received data to the address specified by the OSI
                 ;
827A CD 9782     RECEIV: CALL  SETUP           ; Set memory pointer & byte counter
                 ;Start receiving procedure
827D CD B682      NEXIN: CALL  DATAIN          ; Receive a data byte from ACIA
8280 77                  MOV   M,A             ; Store the byte to specified addr.
8281 CD AC82             CALL  CHKSUM          ; End of data?
8284 C2 7D82             JNZ   NEXIN           ; No, go for next
8287 C3 6A82             JMP   CHECK           ; Yes, go to check error
                 ;
                 ;************************* Routine RUN **************************
                 ;
                 ;Routine RUN transfers execution control to the program specified
                 ;by the OSI-C4P.  The specified program is executed as subroutine.
                 ;
828A CD B682        RUN: CALL  DATAIN          ; Get starting address hi-byte
828D 67                  MOV   H,A
828E CD B682             CALL  DATAIN          ; Get starting address low-byte
8291 6F                  MOV   L,A             ; (H,L)=Starting address
                 ;
                 ;Set up re-entry address for returning from the specified program
8292 11 3982             LXI   D,WACOMD        ; Re-entry is WACOMD
8295 D5                  PUSH  D               ; WACOMD to Stack
```

```
8296 E9                       PCHL                          ; Go to execute the specified prog.
                   ;
                   ;
                   ;*********************** Subroutine SETUP ***********************
                   ;
                   ;Subroutine SETUP gets Starting address & Byte-count from the OSI,
                   ;and clears Checksum.
                   ;
8297 CD B682        SETUP:    CALL   DATAIN               ; Get starting address hi-byte
829A 67                       MOV    H,A                  ; Set memory pointer hi-byte
829B CD B682                  CALL   DATAIN               ; Get starting address low-byte
829E 6F                       MOV    L,A                  ; Set memory pointer low-byte
829F CD B682                  CALL   DATAIN               ; Get byte-count hi-byte
82A2 57                       MOV    D,A                  ; Set byte counter hi-byte
82A3 CD B682                  CALL   DATAIN               ; Get byte-count low-byte
82A6 5F                       MOV    E,A                  ; Set byte counter low-byte
82A7 3E 00                    MVI    A,0
82A9 47                       MOV    B,A                  ; Clear checksum hi-byte
82AA 4F                       MOV    C,A                  ; Clear checksum low-byte
82AB C9                       RET
                   ;
                   ;*********************** Subroutine CHKSUM **********************
                   ;
                   ;Subroutine CHKSUM accumulates Checksum, increments Memory Pointer
                   ;and decrements Byte Counter.
                   ;
82AC 81            CHKSUM:    ADD    C                    ; Add data to checksum low-byte
82AD 4F                       MOV    C,A
82AE 78                       MOV    A,B
82AF CE 00                    ACI    0                    ; Propagate CY to checksum hi-byte
82B1 23                       INX    H                    ; Point to next location
82B2 1B                       DCX    D                    ; Decrement byte counter by 1
82B3 7A                       MOV    A,D
82B4 B3                       ORA    E                    ; Set or reset Z flag
82B5 C9                       RET
                   ;
                   ;*********************** Subroutine DATAIN **********************
                   ;
                   ;Subroutine DATAIN checks the status of RDRF bit, and loads the
                   ;received byte to Accumulator.
                   ;
82B6 DB 8E         DATAIN:    IN     STATUS               ; Load ACIA Status Register
82B8 E6 01                    ANI    MSKRRF               ; Is a data received?
82BA CA B682                  JZ     DATAIN               ; No, keep trying
82BD DB 8F                    IN     OSIC4P               ; Yes, get it
82BF C9                       RET
                   ;
                   ;*********************** Subroutine EMPTY ***********************
                   ;
                   ;Subroutine EMPTY checks the status of TDRE bit until TDRE is set,
                   ;which means ACIA is ready to transmit another byte.
                   ;
82C0 DB 8E          EMPTY:    IN     STATUS               ; Load ACIA Status Register
82C2 E6 02                    ANI    MSKTRE               ; Is Transmit Data Register busy?
82C4 CA C082                  JZ     EMPTY                ; Yes, keep checking
82C7 C9                       RET                         ; No
                   ;
                   ;
                   ;------------------------- Command Table -------------------------
                   ;
82C8     4F         TABLE:    DB     'O'                  ; TRANSMIT command byte
82C9     5882                 DW     TRANSM               ; TRANSM entry address
82CB     49                   DB     'I'                  ; RECEIVE command byte
82CC     7A82                 DW     RECEIV               ; RECEIV entry address
82CE     52                   DB     'R'                  ; RUN command byte
```

```
82CF    8A82  191             DW     RUN          ; RUN entry address
82D1    45    192             DB     'E'          ; RESET command byte
82D2    0800  193             DW     08H          ; Monitor RST 1 routine entry addr.
              194             END
```

SYMBOL TABLE:

```
BEGIN  8227  RESET  0057  PROGRM 0015  MSKCTS 0008  MSKRRF 0001
MSKTRE 0002  STATUS 008E  CONTRL 008E  OSIC4P 008F  NOTYET 8232
WACOMD 8239  NEXT   8241  FOUND  824F  TRANSM 8258  NEXOUT 8258
CHECK  826A  RECEIV 827A  NEXIN  827D  RUN    828A  SETUP  8297
CHKSUM 82AC  DATAIN 82B6  EMPTY  82C0  TABLE  82C8
```

OBJECT CODES

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
8220                       31 C2 20 3E 57 D3 8E 3E 15
8230 D3 8E DB 8E E6 08 C2 32 82 CD B6 82 21 C8 82 06
8240 04 BE 23 CA 4F 82 05 CA 39 82 23 23 C3 41 82 D3
8250 8F 7E 5F 23 7E 57 EB E9 CD 97 82 CD C0 82 7E D3
8260 8F CD AC 82 C2 5B 82 CD C0 82 78 D3 8F CD B6 82
8270 B8 C2 39 82 79 D3 8F C3 39 82 CD 97 82 CD B6 82
8280 77 CD AC 82 C2 7D 82 C3 6A 82 CD B6 82 67 CD B6
8290 82 6F 11 39 82 D5 E9 CD B6 82 67 CD B6 82 6F CD
82A0 B6 82 57 CD B6 82 5F 3E 00 47 4F C9 81 4F 78 CE
82B0 00 23 1B 7A B3 C9 DB 8E E6 01 CA B6 82 DB 8F C9
82C0 DB 8E E6 02 CA C0 82 C9 4F 58 82 49 7A 82 52 8A
82D0 82 45 08 00
```

APPENDIX C - OSI-C4PMF DATA COMMUNICATION PROGRAM

```
                    ;**************************************************
                    ;
                    ;        OSI-C4P DATA COMMUNICATION PROGRAM
                    ;        ----------------------------------
                    ;
                    ;This 6502 assembly language program is loaded from
                    ;disk whenever the Extended Monitor written in BASIC
                    ;is executed.  It occupies memory from 5E00 through
                    ;5EE9 and uses page zero locations 9B & 9C.  It is
                    ;composed of four major subroutines called by the
                    ;BASIC routine LINK of the Extended Monitor program
                    ;to implement the corresponding data communication
                    ;command with the SDK-85 system.
                    ;
                    ;**************************************************
                    ;
                    ;
                    ;Definitions:
                    ;
5E00=                START  =  $5E00          Program starting location
009C=                 MPHI  =  $9C            Local memory pointr hi-byt
009B=                 MPLO  =  $9B            Local memory pointr lo-byt
FC00=               STATUS  =  $FC00          Status Register of ACIA
FC00=               CONTRL  =  $FC00          Control Register of ACIA
0008=               MSKCTS  =  %00001000      Pattern for testing CTS
0001=               MSKRRF  =  %00000001      Pattern for testing RDRF
0002=               MSKTRE  =  %00000010      Pattern for testing TDRE
FC01=                SDK85  =  $FC01          ACIA Trans/Receiv Register
                    ;
                    ;
5E00                        *  =  START
                    ;
                    ;
                    ;*************** Subroutine TRANSM ***************
                    ;
                    ;TRANSM is called by BASIC to implement SEND comm-
                    ;and of Extened Monitor.  It orders the SDK-85 to
                    ;enter the receiving mode, and transmits the data
                    ;block specified by BASIC to the SDK-85.
                    ;
5E00 A001                   LDY  #$01         Y points RECEIVE command
5E02 20625E                 JSR  BEGIN        Return w/ SDK-85 entered
                    ;                         receiving mode
5E05 208A5E                 JSR  SETUP        Return w/ SDK-85 ready to
                    ;                         accept data bytes, & Y=0
5E08 B19B           NEXOUT  LDA  (MPLO),Y     Get a byte
5E0A 20D05E                 JSR  DATAO        Transmit the byte
5E0D 20A15E                 JSR  CHKSUM       Add CHECKSUM, inc pointer,
                    ;                         dec byte-count
5E10 D0F6                   BNE  NEXOUT       Data end?/ No, go for next
                    ;                         / Yes, check CHECKSUM
                    ;
                    ;The following procedures are shared by TRANSM and
                    ;RECEIV. It compares SDK-85 checksum to OSI check-
                    ;sum, and generates status code to notify BASIC.
                    ;
5E12 20C55E          CHECK  JSR  DATAIN       Get SDK-85 checksum hi-by
5E15 CDE35E                 CMP  CHKHI        Agree w/ OSI's?
5E18 D00C                   BNE  ERRHI        No
5E1A 8D01FC                 STA  SDK85        Yes, request checksum lo
5E1D 20C55E                 JSR  DATAIN       Get SDK-85 checksum lo-by
5E20 CDE25E                 CMP  CHKLO        Agree w/ OSI's?
5E23 D007                   BNE  ERRLO        No
5E25 60                     RTS               Yes, return to BASIC
```

```
                   ;
5E26 ADE35E         ERRHI  LDA  CHKHI        Wrong checksum hi-byte
5E29 8D01FC                STA  SDK85        No need to send lo-byte
5E2C A903           ERRLO  LDA  #$03         Transmission error message
5E2E 8DE45E                STA  MSG          For informing BASIC
5E31 60                    RTS
                   ;
                   ;
                   ;*************** Subroutine RECEIV ****************
                   ;
                   ;RECEIV is called by BASIC to implement GET command
                   ;of Extended Monitor.  It orders the SDK-85 to en-
                   ;ter the transmission mode, and receives the data
                   ;block specified by BASIC from SDK-85 to the corre-
                   ;sponding simulated memory locations in OSI-C4P.
                   ;
5E32 A000                  LDY  #$00         Y points TRANSMIT command
5E34 20625E                JSR  BEGIN        Return w/ SDK-85 entered
                   ;                         TRANSMISSION mode
5E37 208A5E                JSR  SETUP        Return w/ SDK-85 ready to
                   ;                         send data bytes, & Y=0
5E3A 20C55E         NEXIN  JSR  DATAIN       Get a byte from SDK-85
5E3D 919B                  STA  (MPLO),Y     Allocate the byte
5E3F 20A15E                JSR  CHKSUM       Add checksum, Inc pointer,
                   ;                         Dec byte-count
5E42 D0F6                  BNE  NEXIN        Data end?/ No, go for next
5E44 4C125E                JMP  CHECK        Yes, go to check CHECKSUM
                   ;
                   ;
                   ;***************** Subroutine RUN *****************
                   ;
                   ;RUN is called by BASIC to implement RUN command of
                   ;Extended Monitor.  It orders the SDK-85 to execute
                   ;a user specified 8085 program.
                   ;
5E47 A002                  LDY  #$02         Y points RUN command
5E49 20625E                JSR  BEGIN        Return w/ SDK-85 read
                   ;                         to accept address
                   ;       LDY  #4           Y points to STAHI
5E4C A004                  LDY  #4           Y points to STAHI
5E4E B9DD5E                LDA  BYCLO-1,Y    Get address hi-byte
5E51 8D01FC                STA  SDK85        Send to SDK-85
5E54 88                    DEY               Y points to STALO
5E55 B9DD5E                LDA  BYCLO-1,Y    Get address lo-byte
5E58 20D05E                JSR  DATAO        Send to SDK-85
5E5B 60                    RTS               Return to BASIC
                   ;
                   ;
                   ;*************** Subroutine RESET ****************
                   ;
                   ;RESET is called by BASIC to implement RESET com-
                   ;mand of the Extended Monitor.  It orders the SDK-
                   ;85 to enter the System Monitor.
                   ;
5E5C A003                  LDY  #$03         Y points RESET command
5E5E 20625E                JSR  BEGIN        Return w/ SDK-85 reset
5E61 60                    RTS               Return to BASIC
                   ;
                   ;
                   ;*************** Subroutine BEGIN ****************
                   ;
                   ;BEGIN is called to send the command byte pointed
                   ;by the calling subroutine to SDK-85.  If SDK-85
                   ;returns a wrong echo, execution is return to the
                   ;BASIC program.
                   ;
```

```
                   ;Check if SDK-85 ready to accept command
                   ;
5E62 AD00FC         BEGIN  LDA   STATUS        Get ACIA STATUS Register
5E65 2508                  AND   MSKCTS        Check CTS
5E67 F00A                  BEQ   READY         SDK-85 ready?/ Yes
5E69 A901                  LDA   #1            No, prepare Err Message
5E6B 8DE45E                STA   MSG           For informing BASIC
5E6E BA            RETURN  TSX
5E6F E8                    INX
5E70 E8                    INX
5E71 9A                    TXS                 Point return to BASIC
5E72 60                    RTS                 Return to BASIC
5E73 B9E55E        · READY LDA   CMDTB,Y       Get command byte
5E76 8D01FC                STA   SDK85         Send to SDK-85
5E79 20C55E                JSR   DATAIN        Get echo from SDK-85
5E7C D9E55E                CMP   CMDTB,Y       Right command?
5E7F F008                  BEQ   RIGHT         Yes, go to RIGHT
5E81 A902                  LDA   #2            No, prepare Err Message
5E83 8DE45E                STA   MSG           For informing BASIC
5E86 4C6E5E                JMP   RETURN        Prepare return BASIC
5E89 60             RIGHT  RTS                 Error-free return
                   ;
                   ;
                   ;**************** Subroutine SETUP ***************
                   ;
                   ;SETUP sends the Starting address & Byte-count to
                   ;SDK-85.  It then loads the Memory pointer with it
                   ;Image.  It returns to the calling major subroutin
                   ;with Checksum byte & DECIMAL bit cleared.
                   ;
5E8A A004           SETUP  LDY   #4            Set Y as counter/pointer
5E8C B9DD5E          NEXT  LDA   BYCLO-1,Y
5E8F 20D05E                JSR   DATAO         Send to SDK-85
5E92 88                    DEY
5E93 D0F7                  BNE   NEXT          More to send?/ Yes, next
5E95 ADDD5E                LDA   IMLO
5E98 859B                  STA   MPLO
5E9A ADDD5E                LDA   IMHI
5E9D 859C                  STA   MPHI          Memory pointer in Page 0
5E9F D8                    CLD                 Clear DECIMAL bit
5EA0 60                    RTS
                   ;
                   ;
                   ;*************** Subroutine CHKSUM ***************
                   ;
                   ;CHKSUM accumulates checksum, increments Memory -
                   ;inter, decrements Byte-count
                   ;
5EA1 18            CHKSUM  CLC                 Clear Carry
5EA2 6DE25E                ADC   CHKLO         Accumulate data byte
5EA5 8DE25E                STA   CHKLO
5EA8 9003                  BCC   MPBYT         Test if Carry clear
5EAA EEE35E                INC   CHKHI         Yes, propagate Carry
5EAD E69B           MPBYT  INC   MPLO          No, inc Mem ptr lo-byte
5EAF D002                  BNE   THEN          Test if need inc hi-byte
5EB1 E69C                  INC   MPHI          Yes
5EB3 CCDE5E          THEN  CPY   BYCLO         Test if need decremen
                   ;                           both BYCLO & BYCHI bytes
5EB6 D003                  BNE   NODEC         No, only lo-byte
5EB8 CEDF5E                DEC   BYCHI         Yes
5EBB CEDE5E         NODEC  DEC   BYCLO
5EBE ADDF5E                LDA   BYCHI         Prepare for testing end
5EC1 0DDE5E                ORA   BYCLO         Set/reset zero bit
5EC4 60                    RTS
                   ;
```

```
                    ;*************** Subroutine DATAIN ***************
                    ;
                    ;Gets a data byte from ACIA and returns data in A.
                    ;
5EC5 AD00FC         DATAIN  LDA  STATUS        ACIA Status register in
5EC8 2901                   AND  #MSKRRF       Mask RDRF bit
5ECA F0F9                   BEQ  DATAIN        Data in ?/ No, try again
5ECC AD01FC                 LDA  SDK85         Yes, get data to A
5ECF 60                     RTS
                    ;
                    ;
                    ;*************** Subroutine DATAO ****************
                    ;
                    ;Sends the data byte in A to ACIA for transmission
                    ;
5ED0 AA              DATAO  TAX                Save data byte to X
5ED1 AD00FC           TDRE  LDA  STATUS        Status register to A
5ED4 2902                   AND  #MSKTRE       Mask TDRE bit
5ED6 F0F9                   BEQ  TDRE          Busy?/ Yes, wait
5ED8 8E01FC                 STX  SDK85         No, ready to senddata
5EDB 8A                     TXA                data back to A
5EDC 60                     RTS
                    ;
                    ;
                    ;************ Reserved Memory Bytes **************
                    ;
                    ;This area is initialized by BASIC program
                    ;
5EDD                  IMLO * = *               Image of MPLO
5EDE                  IMHI * = *+1             Image of MPHI
5EDF                 BYCLO * = *+1             Byte-count lo-byte
5EE0                 BYCHI * = *+1             Byte-count hi-byte
5EE1                 STALO * = *+1             SDK-85 start addr lo-byte
5EE2                 STAHI * = *+1             SDK-85 start addr hi-byte
5EE3                 CHKLO * = *+1             Checksum lo-byte
5EE4                 CHKHI * = *+1             Checksum hi-byte
5EE5                   MSG * = *+1             Message byte
                    ;
                    ;
                    ;**************** Command Table *****************
                    ;
5EE6 4F              CMDTB  .BYTE  'O'         TRANSMIT command byte
5EE7 49                     .BYTE  'I'         RECEIVE command byte
5EE8 52                     .BYTE  'R'         RUN command byte
5EE9 45                     .BYTE  'E'         RESET command byte
                    ;
                    ;************************************************
                             END
```

APPENDIX D - ENHANCEMENT SYSTEM EXECUTIVE PROGRAM

```
24 REM SETUP INFLAG & OUFLAG FROM DEFAUL
25 X=PEEK(10950): POKE 8993,X: POKE 8994,X
27 REM CHECK FOR E000 MEMORY
28 FOR SC=1TO30:PRINT:NEXT
29 IFPEEK(57088)=223 THEN POKE9794,37
30 PRINT"SDK-85 EXTENDED MONITOR & CROSS ASSEMBLER SYSTEM EXECUTIVE"
40 PRINT:PRINT" JULY 25, 1982  RELEASE": PRINT
48 POKE 64512,2: REM  SET UP 300 BAUD FOR DECWRITER PRINTER
50 GOTO 100
60 PRINT:PRINT: INPUT "SELECT FUNCTION (1-4)";A
70 ON A GOTO 500,800,300,10000
100 PRINT:PRINT:PRINT "FUNCTIONS AVAILABLE:":PRINT:PRINT
110 PRINT"   (1) EXTENDED MONITOR - INTERCHANGE, MODIFY, & FILE DATA"
115 PRINT
120 PRINT"   (2) EDITOR - EDIT THE 8080/8085 SOURCE LANGUAGE FILES"
125 PRINT
130 PRINT"   (3) ASM85 - ASSEMBLE THE 8080/8085 SOURCE LANGUAGE FILE"
135 PRINT
140 PRINT"   (4) FREE - FREE SYSTEM FOR USER PROGRAMMING"
150 GOTO 60
160 REM
300 REM   ASM85 - ASSEMBLER
310 REM
330 REM CHANGES LOWER WORKING LIMIT TO $53FF
340 POKE 133,83
360 GOSUB 2000
370 RUN"ASM85"
380 REM
500 REM   EXTENDED MONITOR
510 REM
530 REM CHANGES LOWER WORKING LIMIT TO $55FF
550 POKE 133,85
570 DISK!"CALL 5600=36,1": REM BRING ASSEMBLED DATA TO BUFFER
575 DISK!"CALL 5E00=39,1": REM 6502 PROG.IN
580 GOSUB 2000
590 RUN"OSI-85"
600 REM
800 REM   EDIT
810 REM
860 REM CHANGES LOWER WORKING LIMIT TO $57FF
870 POKE 133,87
880 GOSUB 2000
890 RUN"EDIT"
1990 REM
2000 REM   ENABLE "REDO FROM START"
2010 POKE 2893,28:POKE 2894,11
2020 REM   DISABLE "," & ":"
2030 POKE 2972,13: POKE 2976,13
2050 RETURN
3000 REM
10000 REM   FREE THE SYSTEM FOR USER PROGRAMMING
10018 REM
10020 REM ENABLE "," & ":"
10025 POKE 2972,58: POKE 2976,44
10026 REM FULL WORKING SPACE
10028 POKE 133,95
10030 REM REPLACE "NEW" AND "LIST"
10040 POKE 741,76 : POKE 750,78
10060 REM DISABLE "REDO FROM START"
10070 POKE 2893,55:POKE 2894,8
10090 REM ENABLE CONTROL-C
10100 POKE 2073,173
10110 PRINT:PRINT "SYSTEM FREE": PRINT:PRINT"11645 BYTES AVAILABLE"
10120 NEW: END
```

APPENDIX E - SDK-85 EXTENDED MONITOR PROGRAM

```
1 PRINT:PRINT:PRINT "  *** SDK-85 EXTENDED MONITOR ***"
2 PRINT:PRINT"Current data in buffer are released by the Assembler"
7 REM  Display & define pseudo memory range and command array
8 BS=22016:GOSUB 30500:GOSUB 40000
9 REM  Recover User Directory
10 DISK!"CALL 5F00=39,2"
11 A=24320: FOR X=1 TO 5: T$=""
12 N=PEEK(A):A=A+1:IF N>7 GOTO 100
14 FOR Y=1 TO N:T$=T$+CHR$(PEEK(A)):A=A+1:NEXT Y
16 F$(X)=T$:P(X)=PEEK(A):A=A+1:NEXTX
18 GOTO 502: REM  To start command recognition
20 GOTO 600: REM  DUMP Routine entry
22 GOTO 650: REM  GET Routine entry
24 GOTO 700: REM  RUN Routine entry
26 GOTO 750: REM  RESET Routine entry
28 GOTO 800: REM  EXAM Routine entry
30 GOTO 1600: REM  SUBSTITUTE Routine entry
32 GOTO 2400: REM  INSERT Routine entry
34 GOTO 3200: REM  ERASE Routine entry
36 GOTO 4000: REM  SAVE Routine entry
37 GOTO 4500: REM  LOAD Routine entry
38 GOTO 4800: REM  PRINT Routine entry
40 GOTO 5600: REM  MOVE Routine entry
42 GOTO 6400: REM  SEE/SET Routine entry
44 GOTO 1000: REM  CREATE Routine entry
45 GOTO 3500: REM  CHAIN Routine entry
46 GOTO 2000: REM  QUIT Routine entry
90 REM
95 REM  For uninitialized Directory
100 FOR Y=X TO 5:F$(Y)="???":NEXT
110 REM
502 REM  ***** Command Recognition
504 PRINT
505 INPUT"Command";A$
510 N=LEN(A$):T=ASC(LEFT$(A$,1)):IF T<65 OR T>90 GOTO 30000
515 REM  Scan and isolate the leftmost 2 command characters
520 FOR K=1 TO N
525 T=ASC(MID$(A$,K,1))
530 IF T>64 AND T<91 THEN NEXT
540 CM$=LEFT$(LEFT$(A$,K-1),2)
550 J=N-(K-1):CHK=0
555 REM  Check with Command Array entries
560 FOR X=1 TO 16
570 IF CM$<>CT$(X) THEN NEXT
580 ON X GOTO 20,22,24,26,28,30,32,34,36,37,38,40,42,44,45,46
590 GOTO 30000: REM  Syntax error
595 REM
600 REM  ***** DUMP Command Routine
610 GOSUB 10000: REM  Call PARSE
620 ON CHK GOTO 30000,30050,30100,30300
630 LO=0: GOTO 11500: REM  To LINK
635 REM
650 REM  ***** GET Command Routine
660 GOSUB 10000: REM  Call PARSE
670 ON CHK GOTO 30000,30050,30100,30300
672 REM  Extend the end of simulating range if necessary
675 IF EN>DN THEN DN=EN:D=DN:F=2:GOSUB 20600
680 LO=50: GOTO 11500: REM  To LINK
690 REM
700 REM  ***** RUN Command Routine
702 REM  Use default value if no specification
705 IF J=0 THEN NS=ST:GOTO 720
710 GOSUB 20100: REM  Use specification
715 ON CHK GOTO 30000,30050,30100
```

```
718 IF J-(K+3)<>0 GOTO 30000
720 LO=71: GOTO 11500: REM  To LINK
730 REM
750 REM  ***** RESET Command Routine
760 LO=92: GOTO 11640: REM  To LINK with only command
770 REM
900 REM  ***** EXAM Command Routine
810 DP=1: REM  Set flag for screen display
815 GOSUB 10000: REM  Call PARSE
820 ON CHK GOTO 30000,30050,30100,30300
825 DS=NS: REM  NS will be redefined
828 GOSUB 7000: REM  Call DISPLAY
830 PRINT:INPUT"Continue next 256 bytes (Y/N)";A$
840 IF LEFT$(A$,1)<>"Y" GOTO 504
850 BC=256:GOTO 828
860 REM
1000 REM  ***** CREATE Command Routine
1010 REM  Display the current Directory
1030 GOSUB 1200:INPUT"Are you sure (Y/N)";B$
1040 IF LEFT$(B$,1)="N" GOTO 504
1050 PRINT:INPUT"Enter new file name";B$:B$=LEFT$(B$,7)
1060 PRINT:INPUT"At which storage location (1-5)";A$
1070 T=VAL(A$):IF T=0 OR T>5 GOTO 1060
1075 REM  Define filename to Directory & display updated Directory
1080 F$(T)=B$:GOSUB 1200:INPUT"Create another file (Y/N)";B$:GOTO 1040
1090 REM
1198 REM  ***** DISPDIR Subroutine
1199 REM  Display the current Directory on screen
1200 PRINT:PRINT" -- DIRECTORY --":PRINT:PRINT" LOC.  FILE NAME":PRINT
1210 FOR X=1 TO 5:PRINT X;"     ";F$(X):NEXT
1220 PRINT:RETURN
1230 REM
1600 REM  ***** SUBSTITUTE Command Routine
1605 IF J=0 GOTO 30050: REM  No specifications
1610 P=2: GOSUB 20200: ON CHK GOTO 30000,30050,30100,30300
1620 IF NS-ST>2043 GOTO 30300
1625 REM  Substitution
1630 POKE SA,D: SA=SA+1: IF NS<DN GOTO 1650
1635 DN=NS: D=DN: F=2: GOSUB 20600: REM  Extend the simulating range
1640 IF NS-ST=2043 THEN PRINT:PRINT"The end of buffer": GOTO 504
1650 NS=NS+1: D=NS: GOSUB 11200: REM  For next prompt message
1660 PRINT:PRINT"Substitute ";HEX$;" with";
1670 INPUT A$: IF LEFT$(A$,1)="N" GOTO 504
1690 J=LEN(A$): GOSUB 20700: ON CHK GOTO 30000,30050,30100
1700 GOTO 1630
1710 REM
2000 REM  ***** QUIT Command Routine
2005 REM  Save the current Directory to disk before exiting
2010 A=24320:FOR X=1 TO 5
2020 N=LEN(F$(X)):POKE A,N:A=A+1
2025 FOR Y=1 TO N:POKE A,ASC(MID$(F$(X),Y,1)):A=A+1:NEXT Y
2030 POKE A,P(X):A=A+1:NEXTX
2040 DISK!"SAVE 39,2=5F00/1":RUN"BEXEC*"
2050 REM
2400 REM  ***** INSERT Command Routine
2405 IF J=0 GOTO 30050
2410 P=1: GOSUB 20200: ON CHK GOTO 30000,30050,30100,30300
2420 IF DN+D-ST>2043 OR NS>DN GOTO 30300
2425 REM  Move block down
2430 BC=(DN-NS)+1: SA=SA+(DN-NS): F=1: GOSUB 20500
2440 REM  Extend the end of simulating range
2450 DN=DN+D: D=DN: F=2: GOSUB 20600: GOTO 11690
2460 REM
```

```
3200 REM  ***** ERASE Command Routine
3205 IF J=0 GOTO 30050
3210 P=1: GOSUB 20200: ON CHK GOTO 30000,30050,30100,30300
3215 IF NS>DN GOTO 30300
3220 BC=(DN-NS-D)+1: IF BC<0 GOTO 30000
3225 REM  Move data block up for deletion
3230 SA=SA+D: F=-1: D=-D: GOSUB 20500: GOTO 2450
3240 REM
3500 REM  ***** CHAIN Command Routine
3510 GOSUB 8000:IF X>5 GOTO 30400
3515 REM  Calculate the pages of the file in buffer
3520 GOSUB 15000: IF P+P(X)>8 GOTO 30300
3525 REM  Load the specified disk file
3530 T$=RIGHT$(STR$(X+30),2): SA=BS+4+(DN-ST)+1: D=SA: GOSUB 11200
3540 DISK!"CA "+HEX$+"="+T$+",1": BS=SA: F=0: GOSUB 20000: V=D
3545 REM  Extend the simulating range to include the disk file
3550 F=2: GOSUB 20000: BS=22016: BC=D-V+1: D=DN+BC: F=2: GOSUB 20600
3555 REM  Delete the ranging bytes of the loaded disk file
3560 D=-4: SA=SA+4: F=-1: GOSUB 20500: GOSUB 30500: GOTO 11690
3570 REM
4000 REM  ***** SAVE Command Routine
4020 GOSUB 8000: IF X>5 GOTO 30400
4030 GOSUB15000:T$=RIGHT$(STR$(X+30),2):P(X)=P:P$=RIGHT$(STR$(P),1)
4040 DISK!"SA "+T$+",1=5600/"+P$:GOTO 11690
4050 REM
4500 REM  ***** LOAD Command Routine
4520 GOSUB 8000:IF X>5 GOTO 30400
4530 T$=RIGHT$(STR$(X+30),2):DISK!"CA 5600="+T$+",1"
4540 GOSUB 30500:GOTO 11690
4550 REM
4800 REM  ***** PRINT Command Routine
4810 DP=2:PRINT:INPUT"Do you need any title (Y/N)";B$
4820 IF LEFT$(B$,1)="N" GOTO 815
4830 PRINT:INPUT"Title";B$:PRINT#1,B$:PRINT#1:GOTO815
4840 REM
5600 REM  ***** MOVE Command Routine
5610 IF J=0 GOTO 30050
5620 GOSUB 20100:ON CHK GOTO 30000,30050,30100
5630 IF J-(K+3)=0 GOTO 30050
5640 MS=NS:J=J-(K+3):P=4:GOSUB 20200
5650 ON CHK GOTO 30000,30050,30100,30300
5660 EN=D:BC=(EN-NS)+1:D=MS-NS
5670 IF BC+(MS-ST)>2044 OR EN>DN OR NS<ST GOTO 30300
5675 REM  Check move upward or downward
5680 IF MS<NS THEN F=-1:GOSUB 20500:GOTO 5700
5690 F=1:SA=SA+BC-1:GOSUB 20500:GOTO 5720
5695 REM  For upward movement only
5700 IF EN<>DN GOTO 11690: REM  No need to reduce the end
5705 REM  Change the end of simulating range
5710 DN=MS+BC-1:F=2:D=DN:GOSUB 20600:GOTO 11690
5715 REM  For downward movement only
5720 IF MS+BC-1<DN GOTO 11690
5730 GOTO 5710
5740 REM
6400 REM  ***** SEE/SET Command Routine
6410 GOSUB 30500:PRINT:NS=ST:EN=DN
6420 INPUT"Change starting address";A$
6430 PRINT:IF LEFT$(A$,1)="N" GOTO 6480
6440 GOSUB 6600:ON CHK GOTO 30000,30050,30100
6450 NS=D
6480 INPUT"Change ending address";A$
6490 IF LEFT$(A$,1)="N" GOTO 6520
6500 GOSUB 6600:ON CHK GOTO 30000,30050,30100
6510 EN=D
6520 IF EN-NS>2043 OR NS>EN GOTO 30300
6530 ST=NS:D=ST:F=0:GOSUB 20600:DN=EN:D=DN:F=2:GOSUB 20600:GOTO 11690
6550 REM
6600 J=LEN(A$):P=4:GOSUB 20700:RETURN
```

```
6610 REM
7000 REM  ***** DISPLAY Subroutine
7080 NS=INT(DS/16)*16:BK=DS-NS:T=16-BK
7090 PRINT
7120 IF DP=2 GOTO 7150
7130 PRINT"       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F"
7140 GOTO 7160
7150 PRINT#1,"       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F"
7155 REM  For the first row only
7160 BK$="   ":D=NS:GOSUB 11200:DSP$=HEX$
7170 REM  Fill blanks
7180 FOR X=1 TO T
7190 IF BK=0 GOTO 7220
7200 DSP$=DSP$+BK$:BK=BK-1:GOTO 7190
7220 D=PEEK(SA):GOSUB 11200:DSP$=DSP$+" "+RIGHT$(HEX$,2)
7240 DS=DS+1:SA=SA+1:BC=BC-1:IF BC=0 GOTO 7270
7260 NEXT X
7270 GOSUB 7500:NS=NS+16:IF BC=0 THEN RETURN
7280 REM  For the rest of rows
7310 D=NS:GOSUB 11200:DSP$=HEX$
7330 FOR X=1 TO 16
7340 D=PEEK(SA):GOSUB 11200:DSP$=DSP$+" "+RIGHT$(HEX$,2)
7360 DS=DS+1:SA=SA+1:BC=BC-1:IF BC=0 GOTO 7390
7380 NEXT X
7390 GOSUB 7500:IF BC=0 THEN RETURN
7410 NS=NS+16
7420 GOTO 7310
7430 REM
7500 IF DP=1 THEN PRINT DSP$:RETURN
7530 PRINT#1,DSP$:RETURN
7540 REM
8000 REM  ***** GETFILE Subroutine
8005 IF J=0 GOTO 8018: REM  No filename specified
8008 REM  Get filename specification
8010 B$=RIGHT$(A$,J):FOR X=1 TO J:IF MID$(B$,X,1)<>" " GOTO 8020
8015 NEXT
8018 X=6:RETURN
8020 B$=RIGHT$(B$,J-(X-1))
8025 REM  Check with Directory
8030 FOR X=1 TO 5:IF B$<>F$(X) THEN NEXT
8040 RETURN
10000 REM  ***** PARSE Subroutine
10005 REM  Parse the address specification field to return either the
10006 REM  specified value(s) or the default value(s)
10010 IF J<>0 GOTO 10040
10020 SA=BS+4:NS=ST:BC=(DN-ST)+1:EN=DN:GOTO 10150
10040 GOSUB 20100:IF CHK<>0 THEN RETURN
10060 IF J-(K+3)<>0 GOTO 10080
10070 BC=(DN-NS)+1:EN=DN:GOTO 10130
10080 J=J-(K+3):I=4:GOSUB 10500:IF CHK<>0 THEN RETURN
10100 IF J-(K+3)<>0 THEN CHK=1:RETURN
10110 HEX$=DG$:GOSUB 11000:EN=D:BC=(D-NS)+1
10130 SA=BS+4+NS-ST
10140 IF EN-ST>2043 OR NS-ST<0 THEN CHK=4:RETURN
10150 IF BC<0 THEN CHK=1:RETURN
10160 RETURN
10180 REM
10500 REM  ***** GETDG Subroutine
10502 REM  Get I digits from specification field
10510 B$=RIGHT$(A$,J)
10520 FOR K=1 TO J
10530 T=ASC(MID$(B$,K,1))
10540 IF T>47 AND T<58 GOTO 10580
10550 IF T>64 AND T<71 GOTO 10580
10560 NEXT K
10570 CHK=1:RETURN
```

```
10580 DG$=MID$(B$,K,I):T=LEN(DG$):IF T<I THEN CHK=2:RETURN
10590 FOR N=1 TO I
10600 T=ASC(MID$(DG$,N,1))
10610 IF T<48 OR T>70 THEN CHK=3:RETURN
10620 IF T>57 AND T<65 THEN CHK=3:RETURN
10630 NEXT N
10640 RETURN
10650 REM
11000 REM  ***** HTOD Subroutine
11005 REM  Convert input HEX$ to the equivalent decimal output in D
11010 L=LEN(HEX$):D=0
11020 FOR I=1 TO L
11030 N=L+1-I:T=ASC(MID$(HEX$,N,1))
11050 S1=D+16^(I-1)*(T-55):S2=D+16^(I-1)*(T-48)
11070 IF T>64 THEN D=S1
11080 IF T<64 THEN D=S2
11090 NEXT I
11100 RETURN
11110 REM
11200 REM  ***** DTOH Subroutine
11205 REM  Convert the input D to the equivalent 4-digit hex in HEX$
11210 TD(0)=D
11220 FOR I=1 TO 4
11230 TD(I)=INT(TD(I-1)/16):TP(I)=TD(I-1)-TD(I)*16
11250 N=I:IF INT(TD(I))=0 GOTO 11280
11270 NEXT
11280 FOR I=1 TO N
11290 TE$(N+1-I)=CHR$(48+TP(I))
11300 IF TP(I)>9 THEN TE$(N+1-I)=CHR$(55+TP(I))
11310 NEXT
11320 HEX$=""
11330 FOR I=1 TO N
11340 HEX$=HEX$+TE$(I):NEXT
11370 IF N=4 THEN RETURN
11380 HEX$="0"+HEX$:N=N+1:GOTO 11370
11390 REM
11500 REM  ***** LINK Routine - A linkage between BASIC & machine subs
11508 REM  Place the SDK-85 starting address
11510 D=NS:GOSUB 11800:SH=DH:SL=DL:IF LO=71 GOTO 11630
11540 REM  Place the byte-count
11550 D=BC:GOSUB 11800:POKE 24288,DH:POKE 24287,DL
11560 REM  Place the OSI local starting address
11570 D=SA:GOSUB 11800:POKE 24286,DH:POKE 24285,DL
11620 POKE 24291,0:POKE 24292,0: REM  Clear CHECKSUM bytes
11630 POKE 24289,SL:POKE 24290,SH
11635 REM  Zero MSG byte and Set up machine subroutine entry address
11640 POKE 24293,0:POKE 8955,LO:POKE 8956,94
11650 REM  Call the corresponding machine subroutine
11660 POKE 64512,21:X=USR(X):POKE 64512,2
11670 CHK=PEEK(24293): REM  Check communication error status in MSG
11680 ON CHK GOTO 30120,30140,30160
11690 PRINT:PRINT"Done":GOTO 504
11700 REM
11800 REM  ***** SPLIT Subroutine
11805 REM  Split input D to two decimal-byte, DH and DL
11810 GOSUB 11200:T$=HEX$
11820 HEX$=LEFT$(T$,2):GOSUB 11000:DH=D
11830 HEX$=RIGHT$(T$,2):GOSUB 11000:DL=D:RETURN
11840 REM
15000 REM  ***** CALCPAGE Subroutine
15010 P=INT(((DN-ST)+3)/256):IF P*256=(DN-ST)+3 THEN RETURN
15020 P=P+1:RETURN
15050 REM
```

```
20000 REM  ***** STEND Subroutine
20005 REM  Get simulating start or end address value in D
20010 D=PEEK(BS+1+F):GOSUB 11200:SE$=RIGHT$(HEX$,2)
20020 D=PEEK(BS+F):GOSUB 11200:HEX$=SE$+RIGHT$(HEX$,2)
20030 GOSUB 11000:RETURN
20040 REM
20100 REM  ***** GETNS Subroutine
20105 REM  Get the specified starting address value in NS
20110 I=4:GOSUB 10500:IF CHK<>0 THEN RETURN
20120 HEX$=DG$:GOSUB 11000:NS=D:RETURN
20130 REM
20200 REM  ***** SCAN Subroutine
20205 REM  Translate specification field with no default options
20210 GOSUB 20100:IF CHK<>0 THEN RETURN
20230 IF NS<ST THEN CHK=4:RETURN
20240 IF J-(K+3)=0 THEN CHK=2:RETURN
20250 J=J-(K+3):GOSUB 20700:IF CHK<>0 THEN RETURN
20260 SA=BS+4+(NS-ST):RETURN
20270 REM
20500 REM  ***** UPDN Subroutine
20505 REM  Move a block of data upward or downward
20510 FOR X=1 TO BC
20520 T=PEEK(SA):POKE SA+D,T
20530 SA=SA-F:NEXT
20540 RETURN
20550 REM
20600 REM  ***** CHANGE Subroutine
20605 REM  Change simulating start or end boundary to D
20610 GOSUB 11800
20620 POKE BS+F,DL:POKE BS+F+1,DH:RETURN
20630 REM
20700 REM  ***** GETDATA Subroutine
20705 REM  Get P-digit of data from specification field & return value
20710 I=P:GOSUB 10500:IF CHK<>0 THEN RETURN
20720 IF J-(K+(I-1))<>0 THEN CHK=1:RETURN
20730 IF P=1 THEN D=VAL(DG$):RETURN
20740 HEX$=DG$:GOSUB 11000:RETURN
20750 REM
29999 REM  ***** Error Display Procedures
30000 PRINT:PRINT"?Syntax error":GOTO504
30050 PRINT:PRINT"?Lack of data":GOTO504
30100 PRINT:PRINT"?Non-hex error":GOTO504
30120 PRINT:PRINT"Go to initialize SDK-85":GOTO 30200
30140 PRINT:PRINT"Reset & initialize SDK-85":GOTO 30200
30160 PRINT:PRINT"Transmission error"
30200 PRINT:INPUT"Execute again (Y/N)";B$
30210 IF LEFT$(B$,1)="Y" GOTO 11500
30230 GOTO 504
30300 PRINT:PRINT"?Exeeds limits":GOTO504
30400 PRINT:PRINT"?Undefined file":GOTO504
30450 REM
30500 REM  ***** SHOW Subroutine
30505 REM  Define & display the simulating range by the 1st 4 bytes of
30506 REM  the buffer contents
30510 F=0:GOSUB 20000:ST=D
30530 PRINT:PRINT"Simulated SDK-85 Memory Starting Address - ";HEX$
30540 F=2:GOSUB 20000:DN=D
30550 PRINT"                               Ending Address - ";HEX$:RETURN
30560 REM
40000 REM  ***** Define Command Array Subroutine
40010 DIM CT$(16)
40020 FOR X=1 TO 16
40030 READ CT$(X)
40040 NEXT X
40045 RETURN
40050 DATA "DU","GE","RU","RE","EX","SU","IN","ER"
40060 DATA "SA","LO","PR","MO","SE","CR","CH","QU"
```

APPENDIX F - TEXT EDITOR PROGRAM

```
5 REM  Text File Editor Program
10 REM
20 PRINT:PRINT:PRINT"-- TEXT FILE EDITOR --"
30 V=0:GOTO40: REM  Clear Extended mode (Re-run entry for NEW command)
35 V=1: REM  Set Extended mode (Re-run entry for EXTEND command)
36 REM
38 REM  Definitions
40 DIMI$(281),I(280):FORX=1TO9:READT$:C$(X)=T$:NEXT
80 DATA "I","N","F","C","L","P","D","E","Q"
90 REM
100 REM  ***** Command Recognition
105 I=0:C=0: REM  Initialize line-count & data count
110 PRINT:INPUT"Command";A$:N=LEN(A$)
112 REM
115 REM  Test if the leftmost character is a letter
120 T=ASC(LEFT$(A$,1)):IFT<65ORT>90GOTO20000
122 REM
125 REM  Isolate the leftmost character of the syntax field
130 FORK=1TON:T=ASC(MID$(A$,K,1)):IFT>64ANDT<91THENNEXT
150 M$=LEFT$(LEFT$(A$,K-1),1):J=N-(K-1):R=0
155 REM
160 REM  Check with the command array
170 FORX=1TO9:IFM$<>C$(X)THENNEXT
190 ONXGOTO200,500,600,800,1000,2000,3000,4000,4500
195 GOTO20000: REM  Syntax error
196 REM
200 REM  ***** INPUT Command Routine
205 PRINT
210 IFI=280GOTO20300: REM  Test if reach maximum line limit
220 INPUTA$:N=LEN(A$):IFVAL(LEFT$(A$,1))=0GOTO120: REM  May be command
222 REM
225 REM  Shrink the entered line
230 X=I+1:I$(X)=A$:GOSUB9000:IFR=1GOTO20400: REM  Violate space limit
232 REM
235 REM  Fill the line number array
240 GOSUB5000:IFI=0ORI(X)>I(I)GOTO300: REM  No nes     ting
242 REM
245 REM  Sorting Procedures - either replacement or insertion
250 FORY=1TOI:IFI(X)<>I(Y)GOTO270
255 REM  Replace line Y with the new line
260 C=C-LEN(I$(Y))-1:I(Y)=I(X):I$(Y)=I$(X):GOTO310
270 IFI(X)>I(Y)GOTO290
275 REM  Insert the new line at Y and reposition the rest of lines
280 T=I(X):T$=I$(X):A=(I-Y)+1:S=X:E=S:F=-1:GOSUB3500:I(Y)=T:I$(Y)=T$
290 NEXTY
300 I=I+1
305 REM  Test if data-count overflowed
310 C=C+N+1:IFC<4096GOTO210
320 REM  Adjust the file by deleting the highest-numbered line
330 C=C-LEN(I$(I))-1:I$(I)="":I=I-1:R=2:IFC>4095GOTO330
350 GOTO20300: REM  To inform the user that file ends
360 REM
500 REM  ***** NEW Command Routine
510 PRINT:PRINT"OK":RUN30: REM  Clear all variables
520 REM
600 REM  ***** FILE Command Routine
605 PRINT:PRINT"Dumping...":A=22528:P=1:FORX=1TOI
608 REM  Load the character-count of that line
610 N=LEN(I$(X)):POKEA,N:A=A+1:GOSUB700: REM  Test if need 2nd track
615 REM  Load characters of that line
620 FORY=1TON:T=ASC(MID$(I$(X),Y,1)):POKEA,T:A=A+1:GOSUB700:NEXTY
630 NEXTX: REM  Continue the next line
632 REM
635 REM  Install the file-end mark
```

```
640 POKEA,0:IFP=1THENGOSUB750:GOTO20500
645 REM  Store the current buffer to proper file track
650 IFV=0THENDISK!"SAVE 38,1=5800/8":GOTO20500
660 DISK!"SAVE 30,1=5800/8":GOTO20500
670 REM
700 REM  ***** CHKFULL Subroutine
701 REM  Store the buffer to the 1st track of the corresponding file
702 REM  /and initialize buffer pointer if the current buffer full
705 IFA<>24576THENRETURN: REM  Not full yet
710 GOSUB750:P=2:A=22528:RETURN
720 REM
750 REM  ***** SAVEFIRST Subroutine
751 REM  Filemode flag guides the buffer to be saved to track 37 or 29
755 IFV=0THENDISK!"SAVE 37,1=5800/8":RETURN
760 DISK!"SAVE 29,1=5800/8":RETURN
770 REM
800 REM  ***** CALL Subroutine
801 REM  Load either track 37 or 29 to buffer
802 IFV=0THENDISK!"CALL 5800=37,1":GOTO810
804 DISK!"CALL 5800=29,1"
810 PRINT:PRINT"Recovering...":I=0:C=0:X=1:A=22528
812 REM
815 REM  Test if the character-count byte is the end mark (0)
820 I$(X)="":N=PEEK(A):IFN=0GOTO20500
830 C=C+N+1:A=A+1:GOSUB900:FORY=1TON
840 T$=CHR$(PEEK(A)):I$(X)=I$(X)+T$:A=A+1:GOSUB900:NEXTY
850 GOSUB5000:I=I+1:X=X+1:GOTO820
860 REM
900 REM  ***** CHKEMPTY Subroutine
901 REM  Load the 2nd track of corresponding file if necessary
905 IFA<>24576THENRETURN
910 IFV=0THENDISK!"CALL 5800=38,1":GOTO930
920 DISK!"CALL 5800=30,1"
930 A=22528:RETURN
940 REM
1000 REM  ***** LIST Command Routine
1005 F=1: REM  Set flag for screen display
1010 IFI=0GOTO110: REM  Nothing to display
1020 IFJ=0THENS=1:E=I:GOTO1050: REM  Default to all lines
1025 REM  Call STEND to return the proper display range
1030 O=0:GOSUB8000:ONRGOTO20000,20200
1050 GOSUB6000:GOTO20500
1060 REM
2000 REM  ***** PRINT Command Routine
2010 F=2:GOTO1010: REM  Set flag for printer & join LIST
2020 REM
3000 REM  ***** DELETE Command Routine
3010 IFI=0ORJ=0GOTO110: REM  Do nothing when no specifications
3015 REM  Call STEND to return the exact deleting range
3020 O=2:GOSUB8000:ONRGOTO20000,20200
3030 REM  Prepare for deletion
3040 A=I-E:Y=I:I=I-(E+1-S):FORX=STOE:N=LEN(I$(X)):C=C-N-1:NEXT
3090 REM  Call MOVE for deletion and clear useless lines
3100 F=1:GOSUB3500:FORX=I+1TOY:I$(X)="":NEXT
3110 GOTO20500
3120 REM
3500 REM  ***** MOVE Subroutine
3502 REM  Move a block of lines upward or downward
3505 IFA=0THENRETURN: REM  A is the count for how many lines to be move
3510 I(S)=I(E+F):I$(S)=I$(E+F):S=S+F:E=E+F:A=A-1:GOTO3505
3520 REM
4000 REM  ***** EXTEND Command Routine
4010 PRINT:PRINT"OK":RUN35: REM  Go to clear all variables & enter Extend mode
4020 REM
```

```
4500 REM  ***** QUIT Command Routine
4510 RUN"BEXEC*": REM  Exit Editor
4520 REM
5000 REM  ***** PUTID Subroutine
5001 REM  Isolate the line number & place it to line number array
5010 FORK=1TON:T=ASC(MID$(I$(X),K,1)):IFT>47ANDT<58THENNEXT
5040 I(X)=VAL(LEFT$(I$(X),K-1)):RETURN
5050 REM
5999 REM  ***** DISPLAY Subroutine
6000 PRINT:FORX=STOE:GOSUB9600:IFF=1THENPRINTT$:GOTO6020
6010 PRINT#1,T$
6020 NEXT
6050 RETURN
6060 REM
7000 REM  ***** GETPOSITION Subroutine
7005 REM  Return the specified line position in the line number array
7010 REM
7020 REM  Isolate a line specification
7030 FORK=1TOJ:T=ASC(MID$(B$,K,1)):IFT>47ANDT<58GOTO7060
7040 NEXT
7050 R=1:RETURN
7060 A=K:FORK=ATOJ:T=ASC(MID$(B$,K,1)):IFT>47ANDT<58THENNEXT
7070 REM  Get specification value and start searching
7100 J=J-(K-1):L=VAL(MID$(B$,A,(K-A))):T=I:FORX=1TOI
7130 ONOGOTO7160,7180,7180
7140 IFL<=I(X)THENRETURN
7150 GOTO7190
7160 IFL>=I(T)THENRETURN
7170 T=T-1:NEXT
7175 GOTO7200
7180 IFL=I(X)THENRETURN: REM  For DELETE only
7190 NEXT
7200 R=2:RETURN: REM  Not in the file
7210 REM
8000 REM  ***** STEND Subroutine
8005 REM  Interpret the specification field with default value(s)
8010 B$=RIGHT$(A$,J):FORK=1TOJ:T=ASC(MID$(B$,K,1))
8040 IFT=45THENS=1:GOTO8160
8050 IFT>47ANDT<58GOTO8080
8060 NEXT
8070 R=1:RETURN
8080 GOSUB7000:IFR<>0THENRETURN
8090 S=X:IFJ=0THENE=S:RETURN
8100 B$=RIGHT$(A$,J):FORK=1TOJ:T=ASC(MID$(B$,K,1)):IFT<>45THENNEXT
8150 IFJ-K=0THENJ=0:E=I:GOTO8190
8160 O=O+1:GOSUB7000:IFR<>0THENRETURN
8170 IFO=3THENE=X:GOTO8180
8175 S=T
8180 IFJ<>0ORE-S<0THENR=1:RETURN
8190 RETURN
8200 REM
9000 REM  ***** SHRINK Subroutine
9010 T$="":A=1
9015 REM  Search space and collect those preceding non-space characters
9020 GOSUB9200:T$=T$+MID$(I$(X),A,(K-A))
9040 REM  Test if line ends
9050 IFK-1=NTHENI$(X)=T$:N=LEN(T$):RETURN
9060 REM  Search non-space character
9070 A=K:GOSUB9400:IFK-A>26THENR=1:RETURN
9080 IFK-1=NGOTO9050: REM  Ignore the spaces at the end
9085 REM  Collect one space and a repeat-count
9090 T$=T$+" "+CHR$((K-A)+64):A=K:GOTO9020
9100 REM
9199 REM  ***** SEARCHSPACE Subroutine
9200 FORK=ATON:IFMID$(I$(X),K,1)<>" "THENNEXT
9230 RETURN
```

```
9250 REM
9399 REM  ***** SEARCHARAC Subroutine
9400 FORK=ATON:IFMID$(I$(X),K,1)=" "THENNEXT
9430 RETURN
9440 REM
9598 REM  ***** RECOVER Subroutine
9599 REM  Recover a line to its original shape
9600 T$="":A=1:N=LEN(I$(X))
9620 GOSUB9200:T$=T$+MID$(I$(X),A,(K-A)):IFK-1=NTHENRETURN
9650 REM  Recover the space from the repeat-count
9670 A=ASC(MID$(I$(X),K+1,1))-64:FORB=1TOA:T$=T$+" ":NEXT
9720 A=K+2:GOTO9620
9730 REM
19999 REM  Error Procedures
20000 PRINT:PRINT"?Syntax error":GOTO110
20200 PRINT:PRINT"?Not in listing":GOTO110
20300 PRINT:PRINT"Buffer ends at line";I(I):GOTO110
20400 PRINT:PRINT"?Over 26 spaces":PRINT:R=0:GOTO210
20500 PRINT:PRINT"Done":GOTO110
```

APPENDIX G - 8085 CROSS ASSEMBLER PROGRAM

```
9 REM  This Assembler assembles the source program from the
10 REM Editor, and stores the object codes to track 36.  If
11 REM any error is detected, a corresponding error code is
12 REM displayed.  The Assembler would not prepare the file
13 REM listing unless the source file is error free.
15 REM ..................................................
16 REM
17 REM   BRING IN ALL ASCII DATA FOR THE TABLES
18 REM   AND BUILD TABLES
19 REM
20 DISK!"CALL 5400=39,4"
21 DIM B$(79),C(79),T$(100),T(100) : REM  MAX.100 SYMBOLS
22 A=21504 : REM  INIZ MEMORY PTR
23 REM
24 REM   BUILD INSTRUCTION AND BASE-OPCODE TABLES
26 FOR X=0 TO 79: GOSUB 6000: B$(X)=T$: C(X)=T: A=A+1: NEXT
27 REM
28 REM   BUILD REGISTER TABLE (B,C,D,E,H,L,M,A)
30 FOR X=0 TO 7: GOSUB 6000: R$(X)=T$: NEXT
31 REM
32 REM   BUILD REGISTER PAIR TABLE (B,D,H,SP)
34 FOR X=0 TO 3: GOSUB 6000: RP$(X)=T$: NEXT
35 REM
36 REM   BUILD DIRECTIVE  TABLE
38 FOR X=1 TO 6: GOSUB 6000: D$(X)=T$: NEXT
39 REM
40 REM
45 REM   USER SELECTS PRINTER OR SCREEN, SET DISPLAY FLAG (O)
48 REM
50 PRINT : PRINT
55 INPUT "List errors on printer instead of screen (Y/N)";A$
60 IF LEFT$(A$,1)="Y" THEN O=1 : GOTO 80 : REM  PRINTER
65 O=2 : REM  SELECT SCREEN
80 PRINT: PRINT "This is a slow assembler!": PRINT
81 PRINT "Begin assembling .....": PRINT
82 REM
84 REM   ...........................................
85 REM
86 REM   PASS 1 : SET UP MEMORY LAYOUT AND DEFINE SYMBOLS
87 REM   PASS 2 : FILL MEMORY WITH OPCODES AND DATA
90 REM
91 REM   P= PASS FLAG     E= ERROR COUNTER     Y= SYMBOL PTR
92 REM   A= SOURCE MEMORY PTR        S= BUFFER MEMORY PTR
93 REM   U= PROGRAM COUNTER       F= ORG FLAG
94 REM   F2= EXTENDED FILE FLAG
95 REM
96 REM  **** PASS 1 ENTRY
97 REM
98 REM   INITIATES FLAGS AND POINTERS
99 REM
105 P=1 : Y=0 : E=0
106 REM
107 REM  **** PASS 2 ENTRY
108 REM
110 DISK!"CALL 5800=37,1" : REM  BRING THE 1ST SOURCE TRACK IN
111 A=22528: S=21508: F=1: F2=1: U=0:  REM  RESET FLAG AND PTR.
112 REM
```

```
113 REM  **** ENTRY OF SCANNING EACH SOURCE LINE
114 REM
115 R=0 : REM  RESET LINE ERROR CODE (E IS ERROR COUNTER)
116 N=PEEK(A) : A=A+1
117 IF N=0 GOTO 700:  REM  HITS END MARK OF FILE
118 GOSUB 950 : REM  CHECK IF NEEDS 2ND TRACK
119 I$=""      : REM  INIZ
120 REM
122 REM    RECOVER STATEMENT BEFORE ';' AND RECOVER ONE SPACE
123 REM    ONLY EVEN IF THERE ARE SEVERAL SPACES
124 REM
125 FOR X=1 TO N: I=PEEK(A): A=A+1: GOSUB 950
130 IF I=59 GOTO 150 : REM  STOP IF HITS SEMICOLON
135 I$=I$+CHR$(I)
136 REM
137 REM   CHECK IF SPACE THEN SKIP REPEAT-COUNT
138 REM
140 IF I=32 THEN X=X+1 : A=A+1 : GOSUB 950
142 NEXT X
144 REM
145 REM    ADJUST SOURCE MEMORY PTR FOR NEXT LINE
146 REM
150 IF X-1=N GOTO 160 : REM  NO ADJUSTING NEEDED
155 A=A+(N-X) : GOSUB 950 : REM  A POINTS THE START OF NEXT LN
156 REM
158 REM    GET LINE NUMBER
159 REM
160 N=LEN(I$)
162 REM   LOOPING UNTIL HITS NON-NUMBER
164 FOR X=1 TO N
165 T=ASC(MID$(I$,X,1)) : IF T>47 AND T<58 THEN NEXT
172 L=VAL(MID$(I$,1,(X-1))) : REM  L IS LINE NUMBER
176 REM
180 REM  **** ENTER THE FIRST FIELD SCANNING PROCEDURE
182 REM
185 GOSUB 900 : REM  GET THE 1ST FIELD OF CHAR.
190 IF R=1 GOTO 115 : REM  NO CHAR. BACK FOR NEXT LINE
192 REM
194 REM   CHECK IF IT'S DIRECTIVE ('EQU' IS NOT ALLOWED IN
195 REM   THE 1ST.FIELD)
196 REM
200 GOSUB 980 : ON I GOTO 500,9000,280,3500,4000,4500
202 REM
203 REM   CHECK IF IT'S INSTRUCTION
204 REM
210 GOSUB 8500
211 REM
212 REM  ONLY PASS 2 NEEDS SCANNING DATA FIELD
213 REM
215 IF P=2 THEN ON Z GOTO 1000,2000,2500
216 S=S+Z : U=U+Z   : REM  NO EFFECT EVEN Z=0
217 IF Z>0 GOTO 115 : REM  IT'S AN INSTRUCTION, NO SCANNING IN PASS 1
218 IF P=2 GOTO 240 : REM  NO SYMBOL BE DEFINED IN PASS 2
219 REM
222 REM   DEFINE SYMBOL (THE 1ST.FIELD AND PASS 1 ONLY)
224 REM
225 GOSUB 8600 : REM  CHECK IF MULTI.DEFINED
228 IF T<Y THEN R=2 : GOTO 8700 : REM  YES, ERROR!
230 IF Y>100 THEN R=3:GOTO 8700 : REM  SYMBOL TB OVERFLOW!
232 T$(Y)=LEFT$(G$,6) : REM  TAKE FIRST 6
233 T(Y) =U           : REM  DEFINE VALUE (CURRENT ADDR.)
234 Y=Y+1             : REM  INCREMENT SYMBOL PTR
235 REM
236 REM
237 REM  **** ENTER THE SECOND FIELD SCANNING PROCEDURE
239 REM
240 GOSUB 900 : IF R=1 GOTO 8700 : REM  NO CHAR.SYNTAX ERROR
```

```
244 REM
245 REM   CHECK IF IT'S DIRECTIVE ('ORG' & 'END' ARE NOT
246 REM   ALLOWED TO BE PRESENTED)
247 REM
250 GOSUB 980 : ON I GOTO 280,280,3000,3500,4000,4500
254 REM
255 REM   CHECK IF IT'S INSTRUCTION
256 REM
260 GOSUB 8500
265 IF P=1 AND Z>0 THEN S=S+Z: U=U+Z: GOTO 115
270 ON Z GOTO 1000,2000,2500 : REM  PASS2 OR NON-MNE AT PASS1
280 R=1 : GOTO 8700 : REM  CAN NOT RECOGNIZE
285 REM
290 REM  ..................................................
295 REM
300 REM
310 REM
500 REM  ------------ DIRECTIVE: ORG ------------
501 REM
502 REM  SET MEMORY POINTER TO NEW VALUE.
503 REM  NEW START LESS THAN LAST ORG IS
504 REM  NOT ALLOWED.
505 REM  ----------------------------------------
506 REM
510 Z=1 : REM  SET FLAG TO INDICATE ASCII ARE NOT ALLOWED
520 GOSUB 5000 : REM GET DATA FIELD VALUE
525 IF R>0 GOTO 8700
528 REM
530 REM   CHECK NEW START VALUE
532 REM
535 T=D-U : IF T<0 THEN R=4 : GOTO 8700 : REM  NOT ALLOWED
538 REM
540 REM   CHECK IF IT'S THE FIRST ORG
542 REM
545 IF F=1 THEN U=D : F=2 : GOTO 560 : REM  THE 1ST
550 U=U+T : S=S+T                     : REM  THE OTHERS
555 REM
560 GOTO 1090 : REM  TO CHECK ERROR AND EXIT
565 REM
570 REM
700 REM  ------------ CHECK EXTEND ------------
701 REM
702 REM  CHECK EXTEND FLAG.  IF IT WAS SET, THEN
703 REM  IT'S NO 'END' ERROR OTHERWISE SET FLAG
704 REM  ENTER EXTEND MODE.
705 REM  ----------------------------------------
706 REM
710 IF F2=2 THEN R=9: L=0: GOTO 8700: REM  NO 'END'
715 F2=2 : REM  SET EXTEND FLAG
720 DISK!"CALL 5800=29,1"
730 A=22528: GOTO 115:  REM  RESET PTR AND CONTINUE
735 REM
740 REM
750 REM
899 REM
900 REM  ---------- SUBROUTINE: ISOLATE ----------
901 REM
902 REM  SCANNING I$ UNTIL HITS THE DELIMITER THEN
903 REM  RETURNS WITH CHARACTERS OR ERROR MESSAGE.
904 REM
905 REM   ENTRY: X= THE POSITION OF START
906 REM   RETURN:X= THE POSITION OF DELIMITER
907 REM          R= ERROR CONDITION
908 REM          G$
909 REM  -----------------------------------------
910 IF X>N GOTO 921 : REM  THE END OF I$ ALREADY
```

```
911 REM
912 REM  LOOPING UNTIL HITS NO., LETTER, QUOTATION MARK, OR MINUS SIGN
913 REM
915 FOR K=X TO N : I=ASC(MID$(I$,K,1))
916 IF I>47 AND I<58 GOTO 926 : REM  NUMBER
918 IF I>64 AND I<91 GOTO 926 : REM  LETTER
919 IF I=39 OR  I=45 GOTO 926 : REM  ASCII OR MINUS SIGN
920 NEXT K
921 R=1 : RETURN : REM  NO CHAR.INDICATED
922 REM
923 REM  LOOPING UNTIL HITS DELIMITER (EITHER COMMA, COLON, OR SPACE)
925 REM
926 X=K : REM  K MARKS THE START OF CHAR.
928 FOR X=K TO N : I=ASC(MID$(I$,X,1))
930 IF I=58 OR I=44 OR I=32 GOTO 946 : REM  HITS DELIMITER
932 NEXT X
942 REM
946 G$=MID$(I$,K,X-K) : RETURN
947 REM
948 REM
949 REM
950 REM  --------- SUBROUTINE: CHKBUFF ----------
951 REM
952 REM  CHECK IF NEEDS TO BRING THE 2ND.TRACK TO
953 REM  BUFFER. IF SO, RESET SOURCE MEMORY PTR.
954 REM  ----------------------------------------
955 REM
960 IF A<24576 THEN RETURN : REM  NO NEED
962 IF F2=1 THEN DISK!"CALL 5800=38,1": GOTO 970
965 DISK!"CALL 5800=30,1" : REM  EXTENDED MODE
970 A=22528+(A-24576) : RETURN
971 REM
972 REM
973 REM
980 REM  ---------- SUBROUTINE: CMPDIR ----------
981 REM
982 REM  COMPARE CHARACTERS (G$) WITH DIRECTIVE
983 REM  TABLE. RETURN WITH I (1-7)
984 REM  ----------------------------------------
985 REM
990 FOR I=1 TO 6: IF G$<>D$(I) THEN NEXT
992 RETURN
993 REM
994 REM
995 REM
1000 REM  ---------- ONE-BYTE INSTRUCTION ----------
1001 REM
1002 REM  FILLS MEMORY BUFFER WITH OPCODE.
1003 REM  ENTER WITH T POINTS THE FOUND MNEMONIC
1004 REM  ----------------------------------------
1006 REM
1010 B=C(T) : REM  GET BASE OPCODE
1011 REM
1012 REM    CLASSIFICATION
1013 REM
1015 IF T=0  GOTO 1100 : REM  'MOV'
1020 IF T=1 GOTO 1200 : REM  'RST'
1030 IF T<4  GOTO 1300 : REM  'POP' & 'PUSH'
1040 IF T<6  GOTO 1400 : REM  'INR' & 'DCR'
1050 IF T<14 GOTO 1130 : REM  ARITH.& LOGIC
1060 IF T<19 GOTO 1320 : REM  RP FAMILY
1062 REM
1065 REM   THE REST OF ONE-BYTES
1070 REM
1080 D=B
1085 GOSUB 4700 : REM  POKES OPCODE
1086 REM
```

```
1087 REM   ENTRY OF CHECKING UNNECESSARY (EXTRA) OPERAND
1088 REM
1090 GOSUB 900 : IF R=1 GOTO 115 : REM  NO MORE
1095 R=8 : GOTO 8700               : REM  ERROR
1096 REM  ........................................
1097 REM
1100 REM    ENTRY OF 'MOV' (OPCODE=B+R1*8+R2)
1101 REM
1102 GOSUB 1900 : REM  B=B+R1*8
1105 IF R>0 GOTO 8700
1110 REM
1120 REM    ENTRY OF ARITH.& LOGIC (OPCODE=B+R)
1125 REM
1130 GOSUB 1700
1135 IF R>0 GOTO 8700
1150 D=B+T : GOTO 1085 : REM  EXIT OF 'MOV' AND A&L
1155 REM  .........................................
1160 REM
1200 REM    ENTRY OF 'RST' (OPCODE=B+(0-7)*8)
1201 REM
1210 GOSUB 5000 : REM  GET DATA (0-7)
1215 IF R>0 OR D>7 THEN R=8 : GOTO 8700 : REM  ILLEGAL
1220 D=B+D*8 : GOTO 1085 : REM  REENTER ONE-BYTE
1230 REM  .........................................
1240 REM
1300 REM    ENTRY OF 'POP' & 'PUSH' (OPCODE=B+RP*16)
1301 REM
1305 RP$(3)="PSW" : REM  TEMP.CHANGE SP TO PSW FOR HERE ONLY
1310 REM
1315 REM    ENTRY OF RP FAMILY (OPCODE=B+RP*16)
1316 REM
1320 GOSUB 1800 : REM  B=B+RP*16
1330 RP$(3)="SP" : REM  PUT SP BACK
1340 IF R>0 GOTO 8700 : REM  DATA FIELD ERROR
1350 GOTO 1080 : REM  EXIT OF 'POP' & 'PUSH' AND RP FAMILY
1355 REM  ..........................................
1360 REM
1400 REM    ENTRY OF 'INR' & 'DCR' (OPCODE=B+R*8)
1401 REM
1410 GOSUB 1900 : REM  B=B+R*8 BACK
1420 IF R>0 GOTO 8700
1430 GOTO 1080  : REM  REENTER ONE-BYTE
1440 REM  ..........................................
1445 REM
1450 REM
1460 REM
1700 REM  ---------- SUBROUTINE: CHKRGTR ----------
1701 REM
1702 REM  GET NEXT FIELD OF CHARACTERS AND COMPARE
1703 REM  WITH REGISTERS TABLE.  RETURN WITH T POINTS
1704 REM  THE FOUND REGISTER, OR ERROR BACK.
1705 REM  ----------------------------------------
1706 REM
1710 GOSUB 900 : IF R=0 GOTO 1720
1715 R=8 : RETURN : REM  NO CHAR.OR NOT MATCH ERROR
1716 REM
1717 REM    COMPARE WITH TABLE
1718 REM
1720 FOR T=0 TO 3 : IF G$=R$(T) THEN RETURN
1740 NEXT T
1750 GOTO 1715 : REM  CAN NOT FIND
1760 REM
1765 REM
1770 REM
```

```
1800 REM  ---------- SUBROUTINE: GETRP ----------
1801 REM
1802 REM  GET REGISTER-PAIR VALUE (B=0,D=1,H=2,SP OR PSW=3)
1804 REM  RETURN WITH  B=BASE+RP*16
1805 REM  -------------------------------------------------
1806 REM
1810 GOSUB 900 : IF R=1 GOTO 1840 : REM  NO CHAR.ERROR
1815 REM
1816 REM    COMPARE WITH TABLE
1817 REM
1820 FOR T=0 TO 3
1825 IF G$=RP$(T) THEN B=B+T*16 : RETURN : REM  FOUND
1835 NEXT T
1840 R=8 : RETURN : REM  CAN NOT FIND
1845 REM
1850 REM
1855 REM
1900 REM  ---------- SUBROUTINE: GETRGTR ----------
1901 REM
1902 REM  GET REGITER VALUE BACK (B=0,C=1,D=2,...,A=7)
1903 REM  RETURN WITH  B=BASE+R*8
1904 REM  ----------------------------------------
1905 REM
1910 GOSUB 1700 : REM  GET T OR ERROR
1920 IF T>7 THEN R=8 : RETURN
1940 B=B+T*8 : RETURN
1945 REM
1950 REM
1955 REM
2000 REM  ---------- TWO-BYTE INSTRUCTIONS ----------
2001 REM
2002 REM  FILLS MEMORY BUFFER WITH OPCODE AND 1-BYTE
2003 REM  DATA.  ENTER WITH T POINTS THE POSITION OF
2004 REM  THE MNEMONIC IN THE TABLE.
2005 REM  ------------------------------------------
2006 REM
2010 B=C(T)              : REM  GET BASE OPCODE
2020 IF T>46 GOTO 2070 : REM  NOT 'MVI'
2025 REM
2030 REM    'MVI' ONLY
2035 REM
2040 GOSUB 1900          : REM  B=B+R*8
2050 IF R>0 GOTO 8700
2055 REM
2060 REM    REENTRY OF ALL 2-BYTES
2065 REM
2070 D=B : GOSUB 4700  : REM  POKE OPCODE
2080 GOSUB 5000         : REM  GET OPERAND
2090 IF R=1 THEN R=6   : REM  NO OPERAND ERROR
2100 IF R>0 GOTO 8700  : REM  OTHER ERROR
2110 IF G$="'" GOTO 1090:REM  ASCII DATA BEEN POKEN ALREADY
2120 IF D>255 OR D<-128 THEN R=7 : GOTO 8700 : REM  ILLEGAL VALUE
2130 IF D<0 THEN D=256+D:REM  GET 2'S COMP.
2140 GOTO 1085          : REM  EXIT
2150 REM
2160 REM
2170 REM
2500 REM  ---------- THREE-BYTE INSTRUCTIONS ----------
2501 REM
2502 REM  FILLS MEMORY BUFFER WITH OPCODE AND 2-BYTE
2503 REM  DATA (ADDRESS).  ENTER WITH T POINTS THE FOUND
2504 REM  MNEMONIC.
2505 REM  ------------------------------------------
2506 REM
2510 B=C(T)              : REM  GET BASE OPCODE
2520 IF T>57 GOTO 2580 : REM  NOT 'LXI'
```

```
2525 REM     'LXI' ONLY
2530 REM
2540 GOSUB 1800          : REM  B=B+RP*16
2550 IF R>0 GOTO 8700    : REM  ERROR
2560 REM
2565 REM     REENTRY OF ALL 3-BYTES
2570 REM
2580 D=B : GOSUB 4700    : REM  POKE OPCODE
2590 GOSUB 5000          : REM  GET 2-BYTE DATA
2600 IF R=1 THEN R=6 : GOTO 8700 : REM  NO DATA
2610 IF R>0 GOTO 8700    : REM  OTHER ERRORS
2615 IF D>65535 OR D<-2048 THEN R=7 : GOTO 8700 : REM  ILLEGAL VALUE
2620 GOSUB 4600          : REM  POKE 2-BYTE
2630 GOTO 1090           : REM  EXIT
2640 REM
2650 REM
2670 REM
3000 REM  ---------- DIRECTIVE: EQU ----------
3001 REM
3002 REM  GIVES VALUE TO THE SYMBOL JUST DEFINED
3003 REM  OPERAND CAN BE A DECIMAL, HEX, BINARY,
3004 REM  DEFINED SYMBOL, OR AN ASCII DATA.
3005 REM  EXECUTES AT PASS 1 ONLY, ALL ERRORS
3006 REM  WILL BE DISPLAYED IN ERROR 1.
3007 REM  ------------------------------------
3008 REM
3010 IF P=2 GOTO 115 : REM  NO ACTION AT PASS 2
3020 Z=2             : REM  SET FLAG TO PERMIT 1 ASCII
3030 GOSUB 5000      : REM  GET OPERAND IN DECIMAL
3040 IF R>0 GOTO 3070: REM  ERROR
3050 IF G$="'" THEN S=S-1: U=U-1: REM  ASCII HAD BEEN POKED
3060 T(Y-1)=D        : REM  Y WAS INCREASED BY SYMBOL DEF.
3065 GOSUB 900       : REM  CHECK NO MORE
3068 IF R=1 GOTO 115 : REM  ERROR FREE EXIT
3070 R=1 : GOTO 8700 : REM  ERROR EXIT
3080 REM
3090 REM
3500 REM  ---------- DIRECTIVE: DS ----------
3501 REM
3502 REM  RESERVES D BYTES OF MEMORY BUFFER
3505 REM  ------------------------------------
3506 REM
3510 Z=1                 : REM  SET FLAG TO PREVENT ASCII
3520 GOSUB 5000          : REM  GET D
3530 IF R>0 GOTO 8700 : REM  ERROR
3535 IF D<0 THEN R=7: GOTO 8700: REM  ILLEGAL VALUE
3540 S=S+D : U=U+D       : REM  INCREMENT MEMORY POINTERS
3550 GOTO 1090           : REM  CHECK NO MORE, EXIT
3560 REM
3570 REM
4000 REM  ---------- DIRECTIVE: DW ----------
4001 REM
4002 REM  GETS DATA WORDS FOLLOWING THE DW
4003 REM  AND FILLS THOSE WORDS TO MEMORY.
4004 REM  WORD FORM CAN BE EITHER DECIMAL,
4005 REM  HEX, BINARY, OR SYMBOL. NO ASCII
4006 REM  WILL BE ACCEPTED.
4007 REM  ------------------------------------
4008 REM
4010 Z=1                 : REM  SET FLAG TO PREVENT ASCII
4020 GOSUB 5000          : REM  GET FIRST DATA (WORD)
4025 IF R>0 GOTO 8700
4026 REM
4027 REM  REENTRY OF THE NEXT WORD (IF MORE THAN ONE IN A LINE)
4028 REM
4029 IF D>65535 OR D<-2048 THEN R=7 : GOTO 8700 : REM  ILLEGAL VALUE
```

```
4030 GOSUB 4600        : REM  POKE WORD
4040 GOSUB 5000        : REM  CHECK IF MORE
4050 IF R=1 GOTO 115   : REM  NO MORE, EXIT
4060 IF R>1 GOTO 8700 : REM  ERROR EXIT
4070 GOTO 4029
4080 REM
4090 REM
4500 REM  ---------- DIRECTIVE: DB ----------
4501 REM
4502 REM  GETS DATA BYTES FOLLOWING THE DB
4503 REM  AND FILLS THOSE DATA INTO MEMORY.
4504 REM  DATA FORM CAN BE A COMBINATION OF
4505 REM  DECIMAL, HEX, BINARY, DEFINED SYMBOL,
4506 REM  AND A STRING OF ASCII. AS LONG AS,
4507 REM  BYTE VALUE IN THE RANGE OF -127 TO 255
4508 REM  ----------------------------------
4509 REM
4510 Z=80               : REM  RELEASE FLAG TO ALLOW ASCII
4512 GOSUB 5000         : REM  GET DATA
4515 IF R>0 GOTO 8700
4516 REM
4518 REM  REENTRY OF THE NEXT DATA (MORE THAN ONE)
4519 REM
4520 IF G$="'" GOTO 4540: REM  ASCII HAD BEEN POKED
4530 IF D>255 OR D<-128 THEN R=7 : GOTO 8700 : REM  ILLEGAL VALUE
4532 IF D<0 THEN D=256+D: REM  GET 2'S COMP.
4535 GOSUB 4700         : REM  POKE DATA BYTE
4540 GOSUB 5000         : REM  CHECK MORE, AND GET IT
4550 IF R=1 GOTO 115    : REM  NO MORE, EXIT
4560 IF R>0 GOTO 8700   : REM  ERROR EXIT
4570 GOTO 4520          : REM  MORE THAN 1 DATA
4580 REM
4590 REM
4600 REM  ---------- SUBROUTINE: POKWORD ----------
4601 REM
4602 REM  SPLITS INPUT D TO 2 BYTES AND POKES LOW,
4603 REM  HIGH BYTE INTO MEMORY BUFFER IN SEQUENCE.
4604 REM
4605 REM    ENTRY : D= DATA WORD
4606 REM    RETURN: S=S+2, U=U+2
4607 REM  ----------------------------------------
4608 REM
4610 GOSUB 8100         : REM  CONVERTS D TO 4 DIGITS HEX
4620 T$=H$: H$=RIGHT$(T$,2)
4630 GOSUB 8000         : REM  GET LOW-BYTE VALUE
4640 GOSUB 4700         : REM  POKE LOW
4645 H$=LEFT$(T$,2)
4650 GOSUB 8000         : REM  GET HI-BYTE VALUE
4655 GOSUB 4700         : REM  POKE HI
4660 RETURN
4670 REM
4680 REM
4700 REM  ---------- SUBROUTINE: POKEBYTE ----------
4701 REM
4702 REM  POKES A INPUT BYTE (D) INTO NEXT AVAILABLE
4703 REM  MEMORY BUFFER LOCATION THEN INCREMENTS THE
4704 REM  POINTERS FOR NEXT POKING.
4705 REM
4706 REM    ENTRY : D = DATA BYTE
4707 REM    RETURN: S=S+1 & U=U+1
4708 REM  -----------------------------------------
4709 REM
4710 POKE S,D           : REM  POKING
4720 S=S+1 : U=U+1      : REM  INCREMENTS MEMORY POINTERS
4730 RETURN
```

```
4740 REM
4750 REM
5000 REM  ---------- SUBROUTINE: GETDATA ----------
5001 REM
5002 REM  GET A CHARACTER FILED FROM THE REMAINING
5003 REM  STATEMENT AND RETURN WITH ITS DECIMAL VALUE
5004 REM  IN D OR ERROR IN R.
5008 REM  ----------------------------------------
5009 REM
5010 GOSUB 900                   : REM  GET DATA CHARTERS
5012 IF R=1 THEN RETURN          : REM  EXIT WITH NO DATA
5015 M=X-K : A$=G$ : C=1         : REM  FOR ARITH.ONLY
5020 GOSUB 5500                  : REM  CHECK IF ARITHMETIC
5025 IF K<M GOTO 5100            : REM  YES, GO CALCULATING
5030 IF LEFT$(G$,1)="'" GOTO 5200:REM ASCII DATA
5032 REM
5035 REM  -- NESTED SUBROUTINE FOR ARITH.OPERATION --
5038 REM
5040 GOSUB 8600                  : REM  CHECK IF SYMBOL
5042 IF T<Y THEN D=T(T): RETURN: REM  EXIT OF SYMBOL
5045 IF RIGHT$(G$,1)="H" GOTO 5300 :  REM  HEX DATA
5050 IF RIGHT$(G$,1)="B" GOTO 5400 :  REM  BINARY DATA
5052 REM
5054 REM   CHECK EACH CHARACTER IF VALID DECIMAL
5055 REM
5060 FOR I=1 TO M: T=ASC(MID$(G$,I,1))
5065 IF T>57 OR T<48 THEN R=5:RETURN: REM  UNDEFINED SYMBOL
5070 NEXT I
5075 D=VAL(G$)                        : REM  ALL VALID NUMBERS
5080 IF D>65535 THEN R=7              : REM  ILLEGAL VALUE
5085 RETURN                           : REM  EXIT OF DECIMAL
5094 REM
5095 REM      --- ARITHMETIC OPERATION ---
5096 REM
5097 REM  DO ADDITION OR SUBTRACTION FROM LEFT TO RIGHT
5098 REM  * NOT ALLOWED TO HAVE SPACE BETWEEN SIGN AND OPERAND
5099 REM
5100 S(0)=0: V=1: C=2    : REM  INIZ.
5110 IF LEFT$(A$,1)="-" GOTO 5120: REM  HAS MINUS SIGN ALREADY
5115 A$="+"+A$: M=M+1: REM  DEFAULT NO SIGN TO PLUS SIGN
5120 GOSUB 5500: Q=M: M=K-C: REM  Q IS IMAGE OF M
5125 G$=MID$(A$,C,M): REM  C MARKS START OF CHAR.
5130 GOSUB 5040: REM  GET OPERAND VALUE
5135 IF R>0 THEN RETURN
5140 IF MID$(A$,C-1,1)="-" GOTO 5150: REM  SUBSTRACT?
5145 S(V)=S(V-1)+D: GOTO 5155
5150 S(V)=S(V-1)-D
5155 IF K<Q THEN C=K+1: V=V+1: M=Q: GOTO 5120: REM  MORE
5160 D=S(V): RETURN
5180 REM
5200 REM      --- ASCII DATA ---
5201 REM
5210 M=M-2                 : REM  NO COUNT ON 2 "'"
5215 REM
5220 REM   CHECK ASCII PERMITTING FLAG AND SYNTAX ERROR
5225 REM
5230 IF M>Z-1 OR RIGHT$(G$,1)<>"'" THEN R=6: RETURN
5235 REM
5240 G$=MID$(G$,2,M)       : REM  TAKE 2 "'" OFF
5245 REM
5250 REM   POKE EACH ASCII INTO MEMORY BUFFER
5255 REM
5260 FOR I=1 TO M: D=ASC(MID$(G$,I,1)): GOSUB 4700: NEXT I
5265 REM
5270 REM   SET ASCII MESSAGE FOR RETURN
5275 REM
```

```
5280 G$="'" : RETURN
5290 REM
5300 REM     --- HEXADECIMAL DATA ---
5301 REM
5302 H$=LEFT$(G$,M-1)             : REM  GET RID OF TAIL "H"
5305 REM
5306 REM    CHECK EACH CHARACTER IF VALID HEX
5308 REM
5310 FOR I=1 TO M-1 : T=ASC(MID$(H$,I,1))
5315 IF T<48 OR  T>70 GOTO 5350
5320 IF T>57 AND T<65 GOTO 5350
5330 NEXT I
5340 IF I>5 THEN R=7 : RETURN    : REM  4 DIGITS AT MOST
5345 GOSUB 8000 : RETURN         : REM  GET DEC.AND EXIT
5350 R=6 : RETURN                : REM  ERROR EXIT
5360 REM
5400 REM     --- BINARY DATA ---
5401 REM
5410 M=M-1 : G$=LEFT$(G$,M) : REM  GET RID OF TAIL "B"
5415 REM
5420 REM    CHECK EACH CHARACTER IF 1 OR 0
5425 REM
5430 FOR I=1 TO M : T=ASC(MID$(G$,I,1))
5435 IF T<48 OR T>49 GOTO 5350 : REM  SHARE WITH HEX
5440 NEXT
5450 IF I>9 THEN R=7 : RETURN  : REM  8 DIGITS AT MOST
5455 GOSUB 8200 : RETURN       : REM  ERROR FREE EXIT
5470 REM
5480 REM
5500 REM  ---------- SUBROUTINE: CHKSIGN ----------
5501 REM
5502 REM  SCANNING A$ FOR PLUS OR MINUS SIGN
5503 REM  CALLED BY ARITHMETIC OPERATION ONLY
5504 REM
5505 REM    ENTRY : C= POSITION OF STARTING
5506 REM    RETURN: K= POSITION OF SIGN OR ENDING
5507 REM  -----------------------------------------
5508 REM
5510 FOR K=C TO M
5520 T=ASC(MID$(A$,K,1))
5530 IF T<>43 AND T<>45 THEN NEXT K
5540 RETURN
5550 REM
5560 REM
6000 REM  ---------- SUBROUTINE: RECOVER ----------
6001 REM
6002 REM  RECOVER THE INSTRUCTION MNEMONICS, BASE OPCODES,
6003 REM  AND THE DIRECTIVES FOR TABLE BUILD-UP
6005 REM
6006 REM   ENTRY : A= POSITION OF NEXT CHARACTER
6007 REM   RETURN: T$=CHARACTER   T=BASE OP CODE
6008 REM  -----------------------------------------
6009 REM
6010 T$="" :  REM  INIZ
6020 T=PEEK(A): A=A+1: IF T=0 GOTO 6040:  REM  END FOR CHAR.
6030 T$=T$+CHR$(T): GOTO 6020:  REM  RECOVER CHAR.
6040 T=PEEK(A): RETURN
6050 REM
6060 REM
8000 REM  ---------- SUBROUTINE: HEX-DEC ----------
8001 REM
8002 REM  CONVERT INPUT HEX TO DECIMAL OUT
8003 REM
8004 REM    ENTRY : H$       RETURN : D
8005 REM  -----------------------------------------
```

```
8010 J=LEN(H$) : D=0
8020 FOR I=1 TO J : T=ASC(MID$(H$,J+1-I,1)
8030 S1=D+16^(I-1)*(T-55)
8035 IFT<64THEND=S2
8040 S2=D+16^(I-1)*(T-48)
8045 RETURN
8050 IF T>64 THEN D=S1
8060 IF T<64 THEN D=S2
8070 NEXT
8080 RETURN
8085 REM
8090 REM
8100 REM  ---------- SUBROUTINE: DEC-HEX ----------
8101 REM
8102 REM  CONVERT INPUT DECIMAL TO 4 DIGITS HEX
8103 REM
8104 REM     ENTRY : D        RETURN : H$
8105 REM  -----------------------------------------
8108 REM
8110 D(0)=D
8120 FOR I=1 TO 4
8130 D(I)=INT(D(I-1)/16) : P(I)=D(I-1)-D(I)*16 : J=I
8135 IF INT(D(I))=0 GOTO 8140
8138 NEXT I
8140 FOR I=1 TO J
8145 E$(J+1-I)=CHR$(48+P(I))
8150 IF P(I)>9 THEN E$(J+1-I)=CHR$(55+P(I))
8155 NEXT I
8160 H$=""
8165 FOR I=1 TO J : H$=H$+E$(I) : NEXT I
8170 IF J=4 THEN RETURN
8175 H$="0"+H$ : J=J+1 : GOTO 8170
8180 REM
8185 REM
8200 REM  ---------- SUBROUTINE: BIN-DEC ----------
8201 REM
8202 REM  CONVERT BINARY INPUT TO DECIMAL OUT
8203 REM  -----------------------------------------
8204 REM
8210 D=0
8220 FOR I=1 TO M
8230 D=D+2^(I-1)*VAL(MID$(G$,M+1-I,1))
8240 NEXT I
8250 RETURN
8260 REM
8270 REM
8500 REM  ------- SUBROUTINE: SEARCH MNE -------
8501 REM
8502 REM  COMPARE G$ WITH ALL ENTRIES OF THE INSTRUCTION MNEMONIC
8503 REM  TABLE.  RETURN Z AND T
8505 REM  -------------------------------------
8508 REM
8510 FOR T=0 TO 79 : IF G$<>B$(T) THEN NEXT T
8515 REM
8520 IF T<46 THEN Z=1 : RETURN : REM  1-BYTE
8530 IF T<57 THEN Z=2 : RETURN : REM  2-BYTE
8540 IF T<80 THEN Z=3 : RETURN : REM  3-BYTE
8550 Z=0 : RETURN              : REM  NOT FIND
8560 REM
8570 REM
8600 REM  ----- SUBROUTINE: SYMBOL SEARCH -----
8601 REM
8602 REM  COMPARE G$ WITH DEFINED SYMBOL TABLE.
8603 REM  RETURN WITH T.
8605 REM  ------------------------------------
```

```
8610 T$=LEFT$(G$,6) : REM  LOOK 6 CHARACTERS ONLY
8615 REM
8620 FOR T=0 TO Y   : REM  Y IS NUMBER OF DEFINED SYMBOL + 1
8630 IF T$<>T$(T) THEN NEXT T
8640 RETURN
8650 REM  ...............................................
8660 REM
8700 REM  **** ERROR DISPLAY PROCEDURE
8701 REM
8702 REM  DISPLAYS ERROR CODE AND LINE NUMBER.
8703 REM  ALWAYS BACK TO NEW LINE SCANNING.
8708 REM
8720 IF P=1 AND R>4 GOTO 115 : REM  PASS 1 DISPLAYS ERROR 1-4
8730 IF P=2 AND R<5 GOTO 115 : REM  PASS 2 DISPLAY ERROR 5-9
8735 E=E+1                   : REM  INCREMENT ERROR COUNTER
8740 IF O=2 THEN PRINT"Error #";R;" in line";L : GOTO 8760
8750 PRINT #1,"Error #";R;" in line";L
8760 IF R<9 GOTO 115 :  REM  NOT 'NO END ERR', GO NEXT LINE
8770 REM  'NO END ERR', ENTER THE ENDING PROCEDURE
8775 REM
8780 REM
9000 REM  **** ENDING PROCEDURE (OPERATION FOR 'END')
9001 REM
9002 REM  P=1 -ENTER PASS 2       P=2 -EXIT ASSEMBLER
9003 REM
9004 REM  EITHER 'END' OR HITS THE ENDING MARK
9005 REM  SET BY THE EDITOR, WILL END ASSEMBLING.
9006 REM  IF DURING ASSEMBLING, THERE IS ANY ERROR
9007 REM  HAPPENS, NO LISTING WILL BE PREPARED.
9009 REM
9010 IF S<22528 GOTO 9020 : REM  NOT EXCEEDS THE BUFFER YET
9012 PRINT: PRINT "Exceeds buffer capacity": GOTO 9040
9014 REM
9016 REM   CHECK PASS CONDITION
9018 REM
9020 IF P=2 GOTO 9025
9022 P=2: PRINT: PRINT E;"errors in PASS 1": PRINT
9023 PRINT "Continue PASS 2 .....": PRINT: GOTO 110
9024 REM
9025 PRINT: PRINT "End assembling.  Total";E;"errors"
9026 IF E=0 GOTO 9080
9030 REM
9035 REM   ERROR EXIT, REQUEST DESTINATION
9038 REM
9040 PRINT:INPUT"Go back to Editor for corrections (Y/N)";A$
9050 IF LEFT$(A$,1)="N" THEN RUN "BEXEC*"
9060 POKE 133,87 : RUN "EDIT"
9062 REM
9065 REM
9066 REM   ERROR-FREE EXIT, STORE OBJECT CODES TO DISK
9068 REM
9080 A=U : B=S               : REM  SAVE POINTERS
9085 S=21504                 : REM  PREPARE FOR ST.& END ADDR.
9090 D=U-(B-21508):GOSUB 4600:REM  POKE STARTING ADDRESS
9095 D=A-1:         GOSUB 4600:REM  POKE ENDING ADDRESS
9100 DISK!"SAVE 36,1=5400/4": REM  1K BUFFER TO TRACK 36
9110 PRINT "Done!": PRINT
9120 REM
9125 REM   REQUEST DESTINATION
9130 REM
9135 INPUT "Do you want a completed listing (Y/N)";A$
9140 IF LEFT$(A$,1)="Y" THEN RUN "SCRIBE"
9150 RUN "BEXEC*"
9160 REM
9170 REM
```

APPENDIX H - ASSEMBLED FILE LISTING PROGRAM (SCRIBE)

```
1 REM  SCRIBE - Listing Program for Assembled 8080/8085 Object File
2 REM
3 DISK!"CALL 5400=39,4": REM  Load reference table information
4 DIM B$(79),T$(100),T(100)
5 REM
6 REM  Recover the Mnemonic table only
10 A=21504
12 FOR X=0 TO 79
14 B$(X)=""
16 T=PEEK(A): A=A+1
18 IF T>0 THEN B$(X)=B$(X)+CHR$(T): GOTO 16
20 A=A+1
22 NEXT
25 REM
28 REM  Load the first track of source file and object code file
30 DISK!"CALL 5400=36,1":DISK!"CALL 5800=37,1"
32 REM
35 REM  Request listing destination & list the head message
40 B$="8080/8085  CROSS ASSEMBLER,   RELEASED 1982.  E.E. OHIO U."
45 C$="ADDR OP DATA  SEQ       SOURCE STATEMENT"
50 PRINT:PRINT:INPUT"List on printer instead of screen (Y/N)";A$
55 IF LEFT$(A$,1)="Y" THEN O=1: GOTO 65: REM PRINTER
60 O=2:PRINT:PRINTB$:PRINT:PRINT:PRINTC$:PRINT:GOTO100
65 PRINT#1:PRINT#1,B$:PRINT#1:PRINT#1:PRINT#1,C$:PRINT#1
70 REM
80 REM  Initialization
100 A=22528:S=21508:U=0:F=0:Y=0:F2=1:X$="  ":Y$="      "
101 REM
102 REM  Entry of recovering a source statement
105 D$="":R=0:P=0:N=PEEK(A):IF N=0 GOTO 700
110 A=A+1: GOSUB 950: REM  Update source buffer if need
112 I=0
115 T=PEEK(A): A=A+1: GOSUB 950: REM  Update source buffer if need
120 IF T<>32 GOTO 140
122 REM  Recover spaces
125 FOR X=1 TO PEEK(A)-64
130 D$=D$+" ": NEXT X
135 A=A+1:GOSUB 950:I=I+2:GOTO 115
138 REM  Recover non-space characters
140 D$=D$+CHR$(T):I=I+1:IF I<N GOTO 115
145 N=LEN(D$)
146 REM
148 REM  Isolate statement between line number and comments
150 FOR X=1 TO N: REM  Search comments
155 IF MID$(D$,X,1)<>";" THEN NEXT
160 N=X-1: I$=LEFT$(D$,N): REM  Exclude comments
165 FOR X=1 TO N: REM  Pass number characters
170 T=ASC(MID$(I$,X,1)): IF T>47 AND T<58 THEN NEXT
172 REM  X points the first non-number character
180 REM
190 REM  First field scanning
200 GOSUB 900:IF R=1 THEN A$=Y$:B$=X$:C$=Y$:GOTO 8700: REM  Comments
202 REM  Check if directives
205 IF G$="ORG" GOTO 500
210 IF G$="END" GOTO 9000
```

```
215 IF G$="DB"  GOTO 4500
220 IF G$="DS"  GOTO 3500
225 IF G$="DW"  GOTO 4000
228 REM  Check if mnemonics
230 GOSUB 8500: IF Z>0 GOTO 1000
232 REM  Rebuild symbol table
235 T$(Y)=LEFT$(G$,G):T(Y)=U:Y=Y+1
236 REM
238 REM  Second field scanning (must be either directive or mnemonic)
240 GOSUB 900: GOSUB 8500: IF Z>0 GOTO 1000: REM  Mnemonic
242 REM  Either one of the following
245 IF G$="EQU" GOTO 3000
250 IF G$="DS"  GOTO 3500
255 IF G$="DW"  GOTO 4000
260 GOTO 4500: REM  Then must be DB directive
265 REM
500 REM  ***** ORG Operation
505 REM  Scan source statement and evaluate the Program Counter
510 GOSUB 5000:IF F=0 THEN U=D:F=1:GOTO 530
520 S=S+(D-U):U=U+(D-U)
530 D=U:L=4:GOSUB 8100:A$=H$:B$=X$:C$=Y$:GOTO 8700
540 REM
700 REM  ***** EXTEND Routine
705 REM  Set Extended flag, reinitialize buffer w/ Extended source file
710 F2=2:DISK!"CALL 5800=29,1":A=22528:GOTO 105
720 REM
900 REM  ***** ISOLATE Subroutine
905 REM  Collect a field of characters from source statement
910 IF X>N GOTO 922
912 FOR K=X TO N: REM  Search valid start character
914 I=ASC(MID$(I$,K,1))
915 IF I>47 AND I<58 GOTO 925
916 IF I>64 AND I<91 GOTO 925
918 IF I=39 GOTO 925
920 NEXT
922 R=1:RETURN
925 X=K: REM  Mark the start position
930 FOR X=K TO N: REM  Search delimiter or line end
932 I=ASC(MID$(I$,X,1))
935 IF I=58 OR I=44 OR I=32 GOTO 946
940 NEXT
946 G$=MID$(I$,K,X-K):RETURN
948 REM
950 REM  ***** CHKBUFF Subroutine
955 REM  Check if the buffer needs the 2nd track file
960 IF A<24576 THEN RETURN
962 REM  Filetype flag designates the disk access
965 IF F2=1 THEN DISK!"CALL 5800=38,1":GOTO 970
968 DISK!"CALL 5800=30,1"
970 A=22528:RETURN
980 REM
1000 REM  ***** INSTRUCTION Collection Routine
1005 REM  Use the opcodes in opcode buffer
1010 D=U:L=4:GOSUB 8100:A$=H$
1020 GOSUB 4700:B$=H$
1030 IF Z>1 GOTO 1050
1040 C$=Y$:GOTO 8700
1050 IF Z>2 GOTO 1070
1060 GOSUB 4700:C$=H$+X$:GOTO 8700
1070 GOSUB 4600:C$=H$:GOTO 8700
1080 REM
3000 REM  ***** EQU Operation
3005 REM  Rebuild the definition to symbol table
3010 GOSUB 5000:T(Y-1)=D:L=4:GOSUB 8100:C$=H$:A$=Y$:B$=X$:GOTO 8700
3020 REM
```

```
3500 REM  ***** DS Operation
3505 REM  Increment Program Counter to a new setting
3510 GOSUB 5000:C=D:D=U:L=4:GOSUB 8100:A$=H$:D=C:GOSUB 8100
3520 C$=H$:B$=X$:S=S+C:U=U+C:GOTO 8700
3530 REM
4000 REM  ***** DW Directive Data-Collection Routine
4005 REM  Collect word(s) from the object code buffer
4010 GOSUB 5600:D=U:L=4:GOSUB 8100:A$=H$:GOSUB 4600:C$=H$
4020 B$=X$:P=1:GOSUB 8700
4030 C=C-1:IF C=0 GOTO 105
4040 D=U:L=4:GOSUB 8100:A$=H$:GOSUB 4600:C$=H$
4050 D$="":GOSUB 8700:GOTO 4030
4060 REM
4500 REM  ***** DB Directive Data-Collection Routine
4505 REM  Collect byte(s) from the object code buffer
4510 GOSUB 5600:D=U:L=4:GOSUB 8100:A$=H$:GOSUB 4700:C$=H$+X$
4520 B$=X$:P=1:GOSUB 8700
4530 C=C-1:IF C=0 GOTO 105
4540 D=U:L=4:GOSUB 8100:A$=H$:GOSUB 4700:C$=H$+X$
4550 D$="":GOSUB 8700:GOTO 4530
4560 REM
4600 REM  ***** GETWORD Subroutine
4605 REM  Get a word from the object code buffer
4610 GOSUB 4700:L$=H$:GOSUB 4700:H$=L$+H$:RETURN
4650 REM
4700 REM  ***** GETBYTE Subroutine
4705 REM  Get a byte from the object code buffer
4710 D=PEEK(S):L=2:GOSUB 8100:S=S+1:U=U+1:RETURN
4750 REM
5000 REM  ***** GETDATA Subroutine
5005 REM  Get an operand value from the field characters
5010 GOSUB 900:M=X-K:A$=G$:C=1
5015 GOSUB 5500:IF K<M GOTO 5100: REM  Arithmetic operand
5020 IF LEFT$(G$,1)<>"'" GOTO 5030
5025 D=ASC(MID$(G$,2,1)):RETURN: REM  ASCII operand(s)
5030 GOSUB 8600:IF T<Y THEN D=T(T):RETURN: REM  Symbol operand
5035 IF RIGHT$(G$,1)<>"H" GOTO 5045
5040 H$=LEFT$(G$,M-1):GOSUB 8000:RETURN: REM  Hexadecimal operand
5045 IF RIGHT$(G$,1)<>"B" GOTO 5055
5050 M=M-1:G$=LEFT$(G$,M):GOSUB 8200: REM  Binary operand
5055 D=VAL(G$):RETURN: REM  Decimal operand
5090 REM
5100 REM  Arithmetic Procedures
5105 V=0:W=0
5110 Q=M:M=K-C:G$=MID$(A$,C,M):GOSUB 5030
5120 IF V=0 THEN S(0)=D:GOTO 5150
5125 IF MID$(A$,C-1,1)="-" GOTO 5135
5130 S(V)=S(V-1)+D:GOTO 5140
5135 S(V)=S(V-1)-D
5140 IF W=0 GOTO 5150
5145 D=S(V):RETURN
5150 V=V+1:C=K+1:M=Q:GOSUB 5500:IF K>M THEN W=1
5155 GOTO 5110
5160 REM
5500 REM  ***** SEARSIGN Subroutine
5505 REM  Search if any '+' or '-' sign in the source statement
5510 FOR K=C TO M
5520 T=ASC(MID$(A$,K,1)):IF T<>43 AND T<>45 THEN NEXT
5530 RETURN
5540 REM
5600 REM  ***** COUNTOPERAN Subroutine
5602 REM  Count the number of succeeding operand(s)
5605 C=0
5610 GOSUB 900:IF R>0 THEN RETURN
5620 IF LEFT$(G$,1)="'" GOTO 5650
5630 C=C+1:GOTO 5610
```

```
5650 C=C+((X-K)-2):GOTO 5610
5660 REM
8000 REM  ***** HEX-DEC Subroutine
8010 J=LEN(H$):D=0
8020 FOR I=1 TO J
8030 T=ASC(MID$(H$,J+1-I,1))
8040 S1=D+16^(I-1)*(T-55):S2=D+16^(I-1)*(T-48)
8050 IF T>64 THEN D=S1
8060 IF T<64 THEN D=S2
8070 NEXT
8080 RETURN
8090 REM
8100 REM  ***** DEC-HEX Subroutine
8110 D(0)=D
8115 FOR I=1 TO 4
8120 D(I)=INT(D(I-1)/16):P(I)=D(I-1)-D(I)*16
8125 J=I:IF INT(D(I))=0 GOTO 8135
8130 NEXT
8135 FOR I=1 TO J
8140 E$(J+1-I)=CHR$(48+P(I))
8145 IF P(I)>9 THEN E$(J+1-I)=CHR$(55+P(I))
8150 NEXT
8155 H$="":FOR I=1 TO J
8160 H$=H$+E$(I):NEXT
8165 REM   MAKE UP L DIGITS
8170 IF J=L THEN RETURN
8175 H$="0"+H$:J=J+1:GOTO 8170
8180 REM
8200 REM  ***** BIN-DEC Subroutine
8210 D=0: FOR I=1 TO M
8220 D=D+2^(I-1)*VAL(MID$(G$,M+1-I,1))
8230 NEXT
8240 RETURN
8250 REM
8500 REM  ***** CHKMNEMONIC Subroutine
8505 REM  Check with mnemonic table to see it is instruction mnemonic
8510 FOR T=0 TO 79
8520 IF G$<>B$(T) THEN NEXT
8530 IF T<46 THEN Z=1:RETURN
8540 IF T<57 THEN Z=2:RETURN
8550 IF T<80 THEN Z=3:RETURN
8560 Z=0:RETURN
8570 REM
8600 REM  ***** SEARSYMBOL Subroutine
8605 REM  Compare with symbol table entries
8610 T$=LEFT$(G$,6)
8620 FOR T=0 TO Y
8630 IF T$<>T$(T) THEN NEXT
8640 RETURN
8650 REM
8700 REM  ***** DISPLAY Subroutine
8705 REM  Organize a print statement for listing
8710 D$=A$+" "+B$+" "+C$+"  "+D$
8720 GOSUB 8800:IF P=0 GOTO 105
8730 RETURN
8740 REM
8800 REM  ***** PRINT Subroutine
8805 REM  Print a statement to either screen or printer
8810 IF O=2 THEN PRINT D$:RETURN
8820 PRINT#1,D$:RETURN
8840 REM
9000 REM  ***** END Operation
9005 REM  Print the symbol table entries and exit
9010 A$=Y$:B$=X$:C$=Y$:P=1:GOSUB 8700
9500 A$="SYMBOL TABLE:":IF O=2 THEN PRINT:PRINTA$:PRINT:GOTO9505
9502 PRINT#1:PRINT#1,A$:PRINT#1
```

```
9505 K=0
9510 D$=""
9520 FOR X=1 TO 5
9525 T$(K)=T$(K)+" ":IF LEN(T$(K))<7 GOTO 9525
9530 D=T(K):L=4:GOSUB 8100:D$=D$+T$(K)+H$+X$:K=K+1
9540 IF K<Y THEN NEXT
9550 GOSUB 8800:IF K<Y GOTO 9510
9600 PRINT:PRINT"OK":PRINT
9610 INPUT"Do you want to go to Loader (Y/N)";A$
9620 IF LEFT$(A$,1)="N" GOTO 9640
9630 POKE 133,85:DISK!"CALL 5600=36,1"
9635 DISK!"CALL 5E00=39,1":RUN"OSI-85"
9640 RUN"BEXEC*"
```