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

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. EXPANSION OF THE SDK-85 SYSTEM</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Basic System</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Expanded System</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Expansion Driver Circuits</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 External Expansion</td>
<td>9</td>
</tr>
<tr>
<td>3. DATA COMMUNICATION</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Hardware Design</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Software Structure</td>
<td>14</td>
</tr>
<tr>
<td>3.3 SDK-85 Communication Program</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1 TRANSM Routine</td>
<td>19</td>
</tr>
<tr>
<td>3.3.2 RECEIV Routine</td>
<td>21</td>
</tr>
<tr>
<td>3.3.3 RUN Routine</td>
<td>24</td>
</tr>
<tr>
<td>3.4 OSI-C4PMF Communication Program</td>
<td>24</td>
</tr>
<tr>
<td>3.4.1 TRANSM Subroutine</td>
<td>26</td>
</tr>
<tr>
<td>3.4.2 RECEIV Subroutine</td>
<td>32</td>
</tr>
<tr>
<td>3.4.3 RUN Subroutine</td>
<td>32</td>
</tr>
<tr>
<td>3.4.4 RESET Subroutine</td>
<td>35</td>
</tr>
<tr>
<td>4. EXECUTIVE SYSTEM DEVELOPMENT</td>
<td>36</td>
</tr>
<tr>
<td>4.1 Disk Operating System of OSI-C4PMF</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Development Software and Its Executive Program</td>
<td>36</td>
</tr>
<tr>
<td>5. EXTENDED MONITOR</td>
<td>44</td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>44</td>
</tr>
</tbody>
</table>
5.2 Command Format .............................................46
5.3 Main Program Structure .....................................48
5.4 Data Communication Command Routines ..................50
  5.4.1 DUMP Routine ............................................51
  5.4.2 GET Routine ..............................................51
  5.4.3 RUN Routine ..............................................53
  5.4.4 RESET Routine ...........................................54
  5.4.5 LINK Routine ............................................54
5.5 Display & Modification Command Routines .............56
  5.5.1 EXAM and PRINT Routines ..............................56
  5.5.2 SUBSTITUTE Routine ....................................57
  5.5.3 INSERT Routine ..........................................59
  5.5.4 ERASE Routine ..........................................62
  5.5.5 MOVE Routine ..........................................64
  5.5.6 SEE/SET Routine .........................................66
5.6 File Maintenance Command Routines .......................68
  5.6.1 SAVE Routine ............................................68
  5.6.2 LOAD Routine ............................................70
  5.6.3 CHAIN Routine ..........................................70
  5.6.4 CREATE Routine .........................................72
5.7 Subroutines ..................................................75
6. TEXT EDITOR ...............................................82
  6.1 General Description .......................................82
  6.2 Main Program Structure ...................................84
  6.3 Non-File Mode Related Command Routines ...............86
    6.3.1 INPUT Routine ..........................................88
7.4.1 ORG Operation ................................................. 132
7.4.2 EQU Operation .................................................. 132
7.4.3 DS Operation .................................................... 134
7.4.4 DW Operation .................................................... 134
7.4.5 DB Operation .................................................... 137
7.5 Subroutines .......................................................... 137
7.5.1 ISOLATE Subroutine ........................................... 140
7.5.2 GETDATA Subroutine ......................................... 140
7.5.3 POKWORD and POKEBYTE Subroutines ..................... 146
7.6 The Listing Program ................................................ 148
8. SYSTEM OPERATIONS ............................................... 151
8.1 Initialization ....................................................... 151
8.2 Edit Source File ................................................... 154
8.3 Assemble Source File .............................................. 155
8.4 Operations of Extended Monitor ................................. 157
  8.4.1 Insertion of an RET ........................................... 157
  8.4.2 Save Object Code File ........................................ 161
  8.4.3 Load Program to SDK-85 for Execution ...................... 162
  8.4.4 Get Result from SDK-85 ....................................... 163
8.5 Modify Program .................................................... 166
9. SUMMARY AND FUTURE DEVELOPMENTS ............................ 166
  9.1 Summary .......................................................... 166
  9.2 Future Developments ............................................. 167
    9.2.1 Double-Disk System Expansion ......................... 167
    9.2.2 Hardwired Interrupt ....................................... 168
REFERENCES .......................................................... 170
APPENDIX

A. CROSS ASSEMBLER ERROR CODE INTERPRETATION..........................171
B. SDK-85 DATA COMMUNICATION PROGRAM.................................172
C. OSI-C4PMF DATA COMMUNICATION PROGRAM............................176
D. ENHANCEMENT SYSTEM EXECUTIVE PROGRAM..............................180
E. SDK-85 EXTENDED MONITOR PROGRAM..................................181
F. TEXT EDITOR PROGRAM......................................................187
G. 8085 CROSS ASSEMBLER PROGRAM......................................191
H. ASSEMBLED FILE LISTING PROGRAM (SCRIBE)............................203
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 SDK-85 Development System Functional Block Diagram</td>
<td>4</td>
</tr>
<tr>
<td>2.1 SDK-85 Expansion Driver Circuit Diagram</td>
<td>8</td>
</tr>
<tr>
<td>2.2 SDK-85 External Expansion Circuit Diagram</td>
<td>10</td>
</tr>
<tr>
<td>2.3 SDK-85 Expanded System Memory Map</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Flowchart for Main Program Structure of SDK-85</td>
<td>17</td>
</tr>
<tr>
<td>3.2 SDK-85 Command Table Structure</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Flowchart for Subroutine DATAIN</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Flowchart for Subroutine EMPTY</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Flowchart for SDK-85 Routine TRANSM</td>
<td>22</td>
</tr>
<tr>
<td>3.6 Flowchart for SDK-85 Routine RECEIV</td>
<td>23</td>
</tr>
<tr>
<td>3.7 Flowchart for SDK-85 Routine RUN</td>
<td>25</td>
</tr>
<tr>
<td>3.8 OSI-C4PMF Data Communication Program Memory Map</td>
<td>27</td>
</tr>
<tr>
<td>3.9 Flowchart for OSI-C4PMF Subroutine BEGIN</td>
<td>28</td>
</tr>
<tr>
<td>3.10 Flowchart for OSI-C4PMF Subroutine CHKSUM</td>
<td>29</td>
</tr>
<tr>
<td>3.11 Flowchart for OSI-C4PMF Subroutine SETUP</td>
<td>29</td>
</tr>
<tr>
<td>3.12 Flowchart for OSI-C4PMF Subroutine TRANSM</td>
<td>31</td>
</tr>
<tr>
<td>3.13 Flowchart for OSI-C4PMF Subroutine RECEIV</td>
<td>33</td>
</tr>
<tr>
<td>3.14 Flowchart for OSI-C4PMF Subroutine RUN</td>
<td>34</td>
</tr>
<tr>
<td>3.15 Flowchart for OSI-C4PMF Subroutine RESET</td>
<td>34</td>
</tr>
<tr>
<td>4.1 OSI-C4PMF Disk Operating System Memory Map</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Disk Track Use Assignment</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Overall Software Development Structure</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Flowchart for System Executive Program</td>
<td>41</td>
</tr>
</tbody>
</table>
4.5 Flowchart for Executive Routines.................................42
5.1 Memory Map for Extended Monitor.................................45
5.2 Command Summary for Extended Monitor..........................47
5.3 Main Program Structure of Extended Monitor.....................49
5.4 Flowchart for Routine DUMP........................................52
5.5 Flowchart for Routine GET........................................52
5.6 Execution Sequence of Routine RUN...............................54
5.7 Execution Sequence of Routine RESET.............................54
5.8 Flowchart for Routine LINK........................................55
5.9 An Example for Displaying Form................................57
5.10 Flowchart for Routine EXAM and PRINT..........................58
5.11 Flowchart for Routine SUBSTITUTE...............................60
5.12 Flowchart for Routine INSERT....................................61
5.13 Flowchart for Routine ERASE.....................................63
5.14 Flowchart for Routine MOVE......................................65
5.15 Flowchart for Routine SEE/SET..................................67
5.16 Flowchart for Routine SAVE......................................69
5.17 Flowchart for Routine LOAD......................................71
5.18 Flowchart for Routine CHAIN.....................................73
5.19 Flowchart for Routine CREATE..................................74
5.20 Flowchart for Subroutine PARSE................................77
5.21 Flowchart for Subroutine SCAN................................78
5.22 Flowchart for Extended Monitor Subroutine DISPLAY............79
5.23 Flowchart for Subroutine GETFILE.................................80
5.24 Flowchart for Subroutine SHOW..................................81
5.25 Flowchart for Subroutine CALCPAGE...............................81
6.1 Command Summary for Editor.................................85
6.2 Flowchart for Editor Main Program Structure...............87
6.3 Flowchart for Routine INPUT...................................89
6.4 Flowchart for Routine LIST and PRINT.........................92
6.5 Flowchart for Routine DELETE.................................94
6.6 Flowchart for Routine FILE.................................97
6.7 Flowchart for Routine CALL.................................99
6.8 Flowchart for Subroutine SHRINK..........................102
6.9 Flowchart for Subroutine RECOVER.........................103
6.10 Flowchart for Subroutine PUTID.........................104
6.11 Flowchart for Subroutine DISPLAY.........................104
6.12 Flowchart for Subroutine STEND.......................105
6.13 Flowchart for Subroutine GETPOSITION..................106
7.1 The Assembler Workspace Memory Map....................108
7.2 Flowchart for Build Tables............................108
7.3 The Standard 8080/8085 Assembler Delimiters............111
7.4 Flowchart for The Initialization Procedure............114
7.5 Flowchart for The First Field Scan Procedure..........116
7.6 Flowchart for The Second Field Scan Procedure.........119
7.7 Flowchart for The Error Display Procedure...............121
7.8 Flowchart for The Ending Procedure.....................123
7.9 The Register Array and Register-pair Array............124
7.10 Base Opcodes & Arithmetic Expression Table for Register-related Instructions.........................126
7.11 Flowchart for One-byte Instructions Translation......128
7.12 Flowchart for Two-byte Instructions Translation.......129
7.13 Flowchart for Three-byte Instructions Translation
7.14 Flowchart for ORG Operation
7.15 Flowchart for EQU Operation
7.16 Flowchart for DS Operation
7.17 Flowchart for DW Operation
7.18 Flowchart for DB Operation
7.19 Flowchart for Subroutine ISOLATE
7.20 Flowchart for Subroutine GETDATA
7.21 Flowchart for Arithmetic Operation
7.22 Flowchart for ASCII Operation
7.23 Flowchart for Hex Operation
7.24 Flowchart for Binary Operation
7.25 Execution Sequence of Subroutine POKEBYTE
7.26 Flowchart for Subroutine POKWORD
7.27 Generalized Flowchart for File Listing Program Scribe
8.1 An Example Program
8.2 Source File of the Example Program
8.3 Listing File of the Example Program
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 divided 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 RTS (Request-To-Send) output pins are connected to each other's 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 RD and WR output pins of the 8085 CPU to generate an active-low signal for reading from or writing to the ACIA. The R/W input pin of the ACIA is connected to the 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 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
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>82C9</td>
<td>TRANSMIT COMMAND BYTE (4F)</td>
</tr>
<tr>
<td>82CA</td>
<td>ENTRY ADDRESS OF ROUTINE</td>
</tr>
<tr>
<td>82CB</td>
<td>TRANSMIT (8258)</td>
</tr>
<tr>
<td>82CC</td>
<td>RECEIVE COMMAND BYTE (49)</td>
</tr>
<tr>
<td>82CD</td>
<td>ENTRY ADDRESS OF ROUTINE</td>
</tr>
<tr>
<td>82CE</td>
<td>RECEIVE (827A)</td>
</tr>
<tr>
<td>82CF</td>
<td>RUN COMMAND BYTE (52)</td>
</tr>
<tr>
<td>82D0</td>
<td>ENTRY ADDRESS OF ROUTINE</td>
</tr>
<tr>
<td>82D1</td>
<td>RUN (828A)</td>
</tr>
<tr>
<td>82D2</td>
<td>RESET COMMAND BYTE (45)</td>
</tr>
<tr>
<td>82D3</td>
<td>ENTRY ADDRESS OF RST 1 IN</td>
</tr>
<tr>
<td>82D4</td>
<td>MONITOR (0008)</td>
</tr>
</tbody>
</table>

**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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>009B</td>
<td>LOCAL MEMORY POINTER LOWBYTE</td>
</tr>
<tr>
<td>009C</td>
<td>LOCAL MEMORY POINTER HIGHBYTE</td>
</tr>
<tr>
<td>5E00</td>
<td>6502 ASSEMBLY LANGUAGE PROGRAM</td>
</tr>
<tr>
<td>5EDC</td>
<td></td>
</tr>
<tr>
<td>5EDD</td>
<td>IMAGE OF MPLO</td>
</tr>
<tr>
<td>5EDE</td>
<td>IMAGE OF MPH</td>
</tr>
<tr>
<td>5EDF</td>
<td>BYTE COUNTER LOWBYTE</td>
</tr>
<tr>
<td>5EBO</td>
<td>BYTE COUNTER HIGHBYTE</td>
</tr>
<tr>
<td>5EE1</td>
<td>START ADDRESS LOWBYTE</td>
</tr>
<tr>
<td>5EE2</td>
<td>START ADDRESS HIGHBYTE</td>
</tr>
<tr>
<td>5EE3</td>
<td>CHECKSUM LOWBYTE</td>
</tr>
<tr>
<td>5EE4</td>
<td>CHECKSUM HIGHBYTE</td>
</tr>
<tr>
<td>5EE5</td>
<td>MESSAGE BYTE</td>
</tr>
<tr>
<td>5EE6</td>
<td>TRANSMIT COMMAND BYTE (4F)</td>
</tr>
<tr>
<td>5EE7</td>
<td>RECEIVE COMMAND BYTE (49)</td>
</tr>
<tr>
<td>5EE8</td>
<td>RUN COMMAND BYTE (52)</td>
</tr>
<tr>
<td>5EE9</td>
<td>RESET COMMAND BYTE (45)</td>
</tr>
</tbody>
</table>

**FIGURE 3.8** OSI-C4PMF Data Communication Program Memory Map
**MESSAGE CODE**

**CLEARED**

**ENTER**

**Check STATUS**

**N**

**SDK-85 Ready?**

**Y**

**Transmit Command Byte**

*** POINTED BY Y**

**Message Code =01**

**Get Response**

**Y**

**Verified ?**

**N**

**Message Code =02**

**Increment SP By 2**

*** SKIP BEGIN RETURN ADDR FOR DIRECT RETURN TO BASIC**

**RETURN**

---

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

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

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 ',', '"', and ':' 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.
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
<table>
<thead>
<tr>
<th>STELLAR FIELD</th>
<th>SPECIFICATION FIELD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dumps contents of XXX through YYYY to SDX-85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dumps contents of XXX through end of simulated memory to SDX-85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gets contents of XXX through YYYY from SDX-85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gets contents of XXX through end of simulated memory from SDX-85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orders SDX-85 to execute program starting at location XXX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orders SDX-85 to execute program in simulated memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Displays contents of XXX through YYYY on screen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Displays entire contents of simulated memory on screen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inserts capacity for D (0-9) bytes starting at address XXX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erases D (0-9) bytes starting at address XXX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moves contents of XXX through YYYY to locations starting at ZZZZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Displays current range of simulated memory / Sets new range</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creates new file name in user file directory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saves current buffer contents to disk under specified file name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loads specified file from disk to buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chains specified file with current file in buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exits Extended Monitor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** (1) XXX, 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.
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
* RETURN ADDR
IN NS & CONTENT IN D

ENTER

Call SCAN

Error?
Y
N

Substitute Content of NS w/D

N

NS > DN?
Y

Update DN & DN Bytes w/NS

Y

End of Buffer?
N

NS = NS + 1

Y

Substitute?

N

Go To Display Error

Read New Content

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

**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
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
Set Flag for Screen

Line-Count = 0 ?

Has Specification Field ?

Initialize SEARCH Flag

Call STEND
Return Start & End Positions

Any Error?

Call DISPLAY

E7

E3

E4

E7

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

- Load Buffer via Track 37 or 29
- Initialization
- Initialize a Line
- Recover Character-Count
  - = 0? (End Mark)
    - Y: E7
    - N
      - Buffer End? (Y: Load Buffer via Track 38 or 30, N: N)
      - Recover Character to Line
        - Y: Init Buffer Point
        - N
          - End of Line? (Y: Call PUTID to Get Line Number, N: N)
            - Inc Line-Count & Acc Data-Count
  - N
- * DEPEND ON FILE MODE FLAG

---

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

* REPEAT-COUNT IS REPRESENTED BY ASCII A - Z
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
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
Call ISOLATE
Second Field

Has Character ?

'ORG' or 'END' ?

Directive Operation (EQU, DS, DB, DW)

Operand Error ?

Evaluate PROG CNTR

Address Error ?

Set Err Code

* SYNTAX ERROR

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.

<table>
<thead>
<tr>
<th>REGISTER TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY SUBSCRIPTS</td>
</tr>
<tr>
<td>REGISTER SYNTAX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REGISTER-PAIR TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY SUBSCRIPTS</td>
</tr>
<tr>
<td>REGISTER-PAIR SYNTAX</td>
</tr>
</tbody>
</table>

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:

$$4 + (S \times 8)$$ ; 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
<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>ARITHMETIC EXPRESSION</th>
<th>REGISTERS USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r1, r2</td>
<td>OPCODE = 64+(s*8)+32</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>INR r</td>
<td>OPCODE = 4+(s*8)</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>DCR r</td>
<td>OPCODE = 5+(s*8)</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>ADD r</td>
<td>OPCODE = 128+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>ADC r</td>
<td>OPCODE = 136+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>SUB r</td>
<td>OPCODE = 144+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>SBB r</td>
<td>OPCODE = 152+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>ANA r</td>
<td>OPCODE = 160+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>XRA r</td>
<td>OPCODE = 168+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>ORA r</td>
<td>OPCODE = 176+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>CMP r</td>
<td>OPCODE = 184+s</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>RST 0-7</td>
<td>OPCODE = 199+(0-7)*8</td>
<td>NON</td>
</tr>
<tr>
<td>POP rp</td>
<td>OPCODE = 193+(s*16)</td>
<td>B, D, H, PSW</td>
</tr>
<tr>
<td>PUSH rp</td>
<td>OPCODE = 197+(s*16)</td>
<td>B, D, H, PSW</td>
</tr>
<tr>
<td>STAX rp</td>
<td>OPCODE = 2+(s*16)</td>
<td>B, D</td>
</tr>
<tr>
<td>LDAX rp</td>
<td>OPCODE = 10+(s*16)</td>
<td>B, D</td>
</tr>
<tr>
<td>INX rp</td>
<td>OPCODE = 3+(s*16)</td>
<td>B, D, H, SP</td>
</tr>
<tr>
<td>DCX rp</td>
<td>OPCODE = 11+(s*16)</td>
<td>B, D, H, SP</td>
</tr>
<tr>
<td>DAD rp</td>
<td>OPCODE = 9+(s*16)</td>
<td>B, D, H, SP</td>
</tr>
<tr>
<td>MVI r, D8</td>
<td>OPCODE = 6+(s*8)</td>
<td>B, C, D, E, H, L, M, A</td>
</tr>
<tr>
<td>LXI rp, D16</td>
<td>OPCODE = 1+(s*16)</td>
<td>B, D, H, SP</td>
</tr>
</tbody>
</table>

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 accommodate 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

* RETURN IF HIT '+', '-', OR FIELD END
BEI'WEEN THE PRESENT SIGN AND THE NEXT SIGN

Call GETDATA (Nested GETDATA)

* CLEAR SUM & MARK THE FIRST SIGN's POSITION TO 1

* BETWEEN THE PRESENT SIGN AND THE NEXT SIGN

* MARK THE NEXT SIGN OR EXPRESSION END

Check + ~-- ...Operand t

Subtract Add from to S~ S~

End of Expression?

RETURN

FIGURE 7.21 Flowchart for Arithmetic Operation
Count The Number of Characters

Check with ASCII Flag

* MAY BE 0, 1, 2, OR NO LIMIT

Allowed?

Has End Quote Mark?

Start From Leftmost Char

Convert ASCII, Call POKEBYTE

End of Characters?

Set Up ASCII Message

RETURN

Illegal Form Error (Err Code=6)

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.

```plaintext
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 151
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 ; Points to LENGTH
15 LXI H,LENGTH ; B = data counter
20 MOV B,M
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-0 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
<table>
<thead>
<tr>
<th>ADDR</th>
<th>OP</th>
<th>DATA</th>
<th>SEQ</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
<td>; Addition of a string of data bytes</td>
</tr>
<tr>
<td>9000</td>
<td>10</td>
<td></td>
<td></td>
<td>ORG 9000H</td>
</tr>
<tr>
<td>9000</td>
<td>21</td>
<td>0E90</td>
<td>15</td>
<td>LXI H,LENGTH ; Points to LENGTH</td>
</tr>
<tr>
<td>9004</td>
<td>46</td>
<td>20</td>
<td>25</td>
<td>MOV B,M ; B = data counter</td>
</tr>
<tr>
<td>9005</td>
<td>23</td>
<td>30</td>
<td>NEXT:</td>
<td>INX H ; Points to data byte</td>
</tr>
<tr>
<td>9006</td>
<td>86</td>
<td>35</td>
<td></td>
<td>ADD M ; Addition</td>
</tr>
<tr>
<td>9007</td>
<td>05</td>
<td>40</td>
<td></td>
<td>DCR B ; Data end?</td>
</tr>
<tr>
<td>9008</td>
<td>C2</td>
<td>0590</td>
<td>45</td>
<td>JNZ NEXT ; No, adds the next byte</td>
</tr>
<tr>
<td>900B</td>
<td>32</td>
<td>0020</td>
<td>50</td>
<td>STA 2000H ; Yes, stores the sum</td>
</tr>
<tr>
<td>900E</td>
<td>02</td>
<td>55</td>
<td>LENGTH:</td>
<td>DB 2 ; 2 data bytes follows</td>
</tr>
<tr>
<td>900F</td>
<td>01</td>
<td>60</td>
<td></td>
<td>DB 01H, 02H ; Data bytes</td>
</tr>
<tr>
<td>9010</td>
<td>02</td>
<td></td>
<td>65</td>
<td>END</td>
</tr>
</tbody>
</table>

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 OF 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    ADDITION
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 ADDITION. The user is able to save this program by typing "SA ADDITION".

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.

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.
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 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

### APPENDIX A - CROSS ASSEMBLER ERROR CODE INTERPRETATION

<table>
<thead>
<tr>
<th>CODE</th>
<th>INTERPRETATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OPERATION CODE SYNTAX ERROR</td>
</tr>
<tr>
<td>2</td>
<td>MULTIPLE SYMBOL DEFINITION</td>
</tr>
<tr>
<td>3</td>
<td>SYMBOL TABLE OVERFLOW (MAXIMUM 100 ENTRIES)</td>
</tr>
<tr>
<td>4</td>
<td>NON-ASCENDING ORG SEQUENCE</td>
</tr>
<tr>
<td>5</td>
<td>UNDEFINED SYMBOL</td>
</tr>
<tr>
<td>6</td>
<td>ILLEGAL OPERAND FORM</td>
</tr>
<tr>
<td>7</td>
<td>ILLEGAL OPERAND VALUE</td>
</tr>
<tr>
<td>8</td>
<td>UNNECESSARY/ILLEGAL OPERAND</td>
</tr>
<tr>
<td>9</td>
<td>NO END DIRECTIVE</td>
</tr>
</tbody>
</table>
APPENDIX B - SDK-85 DATA COMMUNICATION PROGRAM

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

ADDR OF DATA SEG SOURCE STATEMENT
1;******************************************************************************************
2:
3;SDK-85 DATA COMMUNICATION PROGRAM
4;-----------------------------------------------
5:
6;This program resides permanently in the SDK-85 EPROM memory locations starting from address 8227H. It accepts commands from the OSI-C4P system, and executes the corresponding command routines.
7:
8;******************************************************************************************
9:
10:
11:
12;Definitions:
13:
14:B227 15 BEGIN EQU B227H  ; Program starting address
0057 16 RESET EQU 01010111B  ; Pattern for ACIA master reset
0015 17 PROGRM EQU 00010101B  ; Pattern for programming ACIA
0008 18 MSKCTS EQU 00001000B  ; Mask pattern for CTS test
0001 19 MSKRPT EQU 00000001B  ; Mask pattern for RDRF test
0002 20 MSKTRE EQU 00000010B  ; Mask pattern for TDRE test
008E 21 STATUS EQU 8EH  ; ACIA Status Reg.(Read Only)
008E 22 CONTRL EQU 8FH  ; ACIA Control Reg.(Write Only)
008F 23 OSIC4P EQU 8FH  ; ACIA Transmit/Receive Reg.
24:
25:
26; ORG BEGIN
27:
28:
29;************************************************ Main Routine ***********************************
30:
31;Initialization
32:
33:B227 31 C220 33 LXI SP,20C2H  ; Initialize Stack Pointer
B22A 3E 57 34 MVI A,RESET  ; Master reset ACIA
B22C DA 8E 35 OUT CONTRL  ; Programming ACIA, RTS low
B22E 3E 15 36 MVI A,PROGRM
B230 3E 8E 37 OUT CONTRL  ; Master reset ACIA
38:
39;B232 DA 8E 39 NOTYET: IN STATUS  ; Status Reg. to A
B234 E6 0B 40 ANI MSKCTS  ; Check if OSI ready
B236 C2 32B2 41 JNZ NOTYET  ; No, check again
42:
43;Re-entry location for Command Routines
44;Wait command input from OSI-C4P
45:
46:B239 C0 0B92 46 WACOMD: CALL DATAIN  ; Yes, set command in
B23C 21 C9B2 47 LXI H,TABLE  ; Set Command Table Pointer
B23F 06 04 48 MVI B,A  ; Set Counter
B241 8E 49 NEXT: CMP M  ; Match?
B242 C3 50 INX H  ; Point to Command Routine addr HI
B243 C3 4F82 51 JZ FOUND  ; Yes, found it
B246 05 52 DCR B  ; Check if end of table
B247 C3 39B2 53 JZ WACOMD  ; Yes, invalid command
B24A 23 54 INX H  ; No, skip address bytes
B24B 23 55 INX H
B24C C3 41B2 56 JMP NEXT  ; Try next
57:
58;Command verification
59:
**Subroutine SETUP**

- **GO to execute the specified program.**
- **Subroutine SETUP gets starting address & byte-count from the OSI, and clears checksum.**

**Subroutine CHKSUM**

- **Add data to checksum low-byte**
- **Clear checksum low-byte**
- **Clear checksum hi-byte**
- **Set or reset Z flag**
- **Get starting address hi-byte**
- **Get starting address low-byte**

**Subroutine DATAIN**

- **Check the status of RDRF bit, and loads the received byte to Accumulator.**

**Subroutine EMPTY**

- **Load ACIA Status Register**
- **Is Transmit Data Register busy?**
- **Yes, keep checking**
- **No**

**Command Table**

- **TRANSMIT command byte**
- **RECEIVE command byte**
- **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  B227  RESET  0057  PROGRAM  0015  MSKCTS  0008  MSKPRF  0001
MSKTS  0002  STATUS  008E  CONTROL  008E  OSIC4P  008F  NOTYET  B232
WACMD  B239  NEXT  B241  FOUND  B24F  TRANSM  B255  NEXOUT  B25B
CHECK  B26A  RECEIV  B27A  NEXIN  B27D  RUN  B28A  SETUP  B297

OBJECT CODES

0 1 2 3 4 5 6 7 8 9 A B C D E F
B220  31 C2 20 3E 57 D3 8E 3E 15
B230  D3 8E D8 8E ES 08 C2 32 B2 CD 68 82 21 CB 82 06
B240  04 3E 23 CA 4F 92 05 CA 3B B2 23 23 C3 41 82 D3
B250  9F 7E 5F 23 7E 57 EB 9E CD 97 B2 CD 62 82 7E D3
B260  8F CD AC 82 C2 5B 82 CD C0 B2 78 93 8F CD 95 82
B270  89 C2 39 B2 79 D3 8F C3 3B B2 CD 97 B2 CD 95 82
B280  77 CD AC 82 C2 7D B2 C3 5A B2 CD 62 82 57 CD 95
B290  82 9F 11 39 B2 D5 E9 CD 6B B2 67 CD 95 82 6F CD
B2A0  B6 B2 57 CD 6B B2 5F 3E 00 47 4F C9 81 4F 79 CE
B2B0  00 23 1B 7A B3 C9 DB 8E ES 01 CA 6B B2 DB 8F C9
B2C0  DB 8E ES 02 CA C0 B2 C9 4F 5B B2 49 7A B2 52 8A
B2D0  82 45 08 00
SUBROUTINE TRANSM

; The following procedures are shared by TRANSM and RECEIVE. It compares SDK-85 checksum to OSI checksum, and generates status code to notify BASIC.

76 5E12 20C55E  CHECK JSR DATAIN  Get SDK-85 checksum lo-by
77 5E15 CDE35E  CMP CHKHI  Agree w/ OSI’s?
78 5E1B D00C  BNE ERRHI  No
79 5E1A B00FC  STA SDK85  Yes, request checksum lo
80 5E1D 20C55E  JSR DATAIN  Get SDK-85 checksum lo-by
81 5E20 CDE25E  CMP CHKLO  Agree w/ OSI’s?
82 5E23 D007  BNE ERRLO  No
83 5E25 60  RTS  Yes, return to BASIC

APPENDIX C - OSI-C4PMF DATA COMMUNICATION PROGRAM

;**************************************************************
; OSIC4P 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 3B & 3C. 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:
;**************************************************************

20 5E00=  START  = $5E00  Program starting location
21 009C=  MPH1  = $9C  Local memory pointer hi-byte
22 0099=  MPL0  = $9B  Local memory pointer lo-byte
23 FC00=  STATUS  = $FC00  Status Register of ACIA
24 FC00=  CONTRL  = $FC00  Control Register of ACIA
25 0099=  MSKCTS  = X00001000  Pattern for testing CTS
26 0001=  MSKRRF  = X00000011  Pattern for testing RDRF
27 0002=  MSKTRE  = X00000010  Pattern for testing TDRE
28 FC01=  SDK85  = $FC01  ACIA Trans/Receive Register

;**************************************************************
;Subroutine TRANSM ***************
;**************************************************************

34  ;TRANSM is called by BASIC to implement SEND command of
35  ;Extended Monitor. It orders the SDK-85 to
36  ;enter the receiving mode, and transmits the data
37  ;block specified by BASIC to the SDK-85.
38  ;
41 5E00 A001  LDY #$01  Y points RECEIVE command
42 5E02 20625E  JSR BEGIN  Return w/ SDK-85 entered
43  ; receiving mode
44 5E05 208A5E  JSR SETUP  Return w/ SDK-85 ready to
45  ; accept data bytes & Y=0
46 5E08 B19B  NEXOUT  LDA (MPL0),Y  Get a byte
47 5E0A 20D05E  JSR DATA0  Transmit the byte
48 5E0D 20A15E  JSR CHKSUM  Add CHECKSUM, inc pointer,
49  ; dec byte-count
50 5E10 D066  BNE NEXOUT  Data end? No, so for next
51  ; / Yes, check CHECKSUM
52  ;
55  ;The following procedures are shared by TRANSM and
56  ;RECEIVE. It compares SDK-85 checksum to OSI checksum,
57  ;and generates status code to notify BASIC.
58 5E12 20C55E  CHECK JSR DATAIN  Get SDK-85 checksum hi-by
59 5E15 CDE35E  CMP CHKHI  Agree w/ OSI’s?
60 5E1B D00C  BNE ERRHI  No
61 5E1A B00FC  STA SDK85  Yes, request checksum lo
62 5E1D 20C55E  JSR DATAIN  Get SDK-85 checksum lo-by
63 5E20 CDE25E  CMP CHKLO  Agree w/ OSI’s?
64 5E23 D007  BNE ERRLO  No
65 5E25 60  RTS  Yes, return to BASIC
85 5E2G ADE35E  ;
86 5E2G ADE35E  ERRHI LDA CHKHI Wrong checksum hi-byte
87 5E2G B9D1FC  STA SDK85 No need to send lo-byte
88 5E2C A903 ERRLO LDA #$03 Transmission error message
89 5E2E 8DE45E  STA MSG For informing BASIC
90 5E31 60 RTS
91 ;
92 ;*************** Subroutine RECEIV ***************
93 ; RECEIV is called by BASIC to implement CET command
94 ; of Extended Monitor. It orders the SDK-85 to enter
95 ; the transmission mode, and receives the data
96 ; block specified by BASIC from SDK-85 to the corre-
97 ; sponding simulated memory locations in OSI-C4P.
98 ;
99 81 5E32 A000  LDY #$00 Y points TRANSMIT command
100 5E34 20523E JSR BEGIN Return w/ SDK-85 entered
101 5E34 20523E ; TRANSMISSION mode
102 5E37 20BA5E JSR SETUP Return w/ SDK-85 ready to send data bytes, & Y=0
103 5E3A 20C55E NEXIN JSR DATAIN Get a byte from SDK-85
104 5E3D B19B STA (MPLO),Y Allocate the byte
105 5E3F 20A15E JSR CHKSUM Add checksum, Inc pointer, Dec byte-count
106 5E42 00D6 BNE NEXIN Data end?/ No, so for next
107 5E44 4C125E JMP CHECK Yes, so to check CHECKSUM
108 5E47 A002 ;
109 5E49 20523E ;*************** Subroutine RUN ***************
110 5E4A 20523E ; RUN is called by BASIC to implement RUN command of
111 5E4B 20523E ; Extended Monitor. It orders the SDK-85 to execute
112 5E4C A004 ; a user specified 8085 program.
113 5E4E B9D35E LDY #$02 Y points RUN command
114 5E51 B9D1FC JSR BEGIN Return w/ SDK-85 read
115 5E54 B8 ; to accept address
116 5E55 B9D35E LDA BYCLO-1,Y Get address hi-byte
117 5E58 20D05E STA SDK85 Send to SDK-85
118 5E5A 20D05E DEY Y points to STALO
119 5E5B 20D05E LDA BYCLO-1,Y Get address lo-byte
120 5E5E 60 JSR DATAO Send to SDK-85
121 5E60 60 RTS Return to BASIC
122 ;
123 ;*************** Subroutine RESET ***************
124 ; RESET is called by BASIC to implement RESET com-
125 ; mand of the Extended Monitor. It orders the SDK-
126 ; 85 to enter the System Monitor.
127 5E5C A003 LDY #$03 Y points RESET command
128 5E5E 20523E JSR BEGIN Return w/ SDK-85 reset
129 5E61 60 RTS Return to BASIC
130 ;
131 ;*************** Subroutine BEGIN ***************
132 ; BEGIN is called to send the command byte pointed
133 ; by the calling subroutine to SDK-85. If SDK-85
134 ; returns a wrong echo, execution is return to the
135 ; BASIC program.
136 ;
\begin{verbatim}
131 ;Check if SDK-8S ready to accept command
132 5E52 AD00FC BEGIN LDA STATUS Get ACIA STATUS Register
133 5E55 2508 AND MSKCTS Check CTS
134 5E57 F004 BEQ READY SDK-8S ready?/ Yes
135 5E69 A901 LDA #1 No, prepare Err Message
136 5E6B BDE45E STA MSG For informing BASIC
137 5E6E 84 RETURN TSK Point return to BASIC
138 5E6F 8E INX
139 5E70 8E INX
140 5E71 9A TKS Return to BASIC
141 5E72 80 RTS
142 5E73 8E55E READY LDA CMDTB,Y Get command byte
143 5E75 8D01FC STA SDK85 Send to SDK-8S
144 5E79 20C55E JSR DATAIN Get echo from SDK-8S
145 5E7C DSE55E CMP CMDTB,Y Right command?
146 5E7F F008 BEQ RIGHT Yes, go to RIGHT
147 5E81 A902 LDA #2 No, prepare Err Message
148 5E83 BDE45E STA MSG For informing BASIC
149 5E86 4CSE5E JMP RETURN Prepare return BASIC
150 5E89 80 RIGHT RTS Error-free return
151
152
153
154 ;************** Subroutine SETUP **************
155
156 ;SETUP sends the Starting address & Byte-count to
157 ;SDK-8S. It then loads the Memory pointer with it
158 ;Image. It returns to the calling major subroutine
159 ;with Checksum byte & DECIMAL bit cleared.
160
161 5E8A A004 SETUP LDY #4 Set Y as counter/pointer
162 5E8C B9D55E NEXT LDA BYCLO-1,Y Send to SDK-8S
163 5E8F 20D05E DEY More to send?/ Yes, next
164 5E92 BB JSR DATAO Send to SDK-8S
165 5E95 D0F7 BNE NEXT More to send?/ Yes, next
166 5E98 ADD55E LDA IMLO Memory pointer in Page 0
167 5E9B B598 STA MPLO
168 5E9D B59C LDA IMHI
169 5E9F ADD55E STA MPHl
170 5EAF DB CLD Clear DECIMAL bit
171 5EAO 80 RTS
172
173
174 ;************** Subroutine CHKSUM **************
175
176 ;CHKSUM accumulates checksum, increments Memory -
177 ;inter, decrements Byte-count
178
179 5EA1 18 CHKSUM CLC Clear Carry
180 5EA2 BDE25E ADC CHKLO Accumulate data byte
181 5EA5 BDE25E STA CHKLO
182 5EAA 9003 BCC MPBYT Test if Carry clear
183 5EAD EEE35E INC CHKHI Yes, propagate Carry
184 5EAF D002 MPBYT INC MPL0 No, inc Mem &r hi-byte
185 5EE1 EB8C BNE THEN Test if need inc lo-byte
186 5EE3 CCDE5E CPY BYCLO Yes
187 5EE7 8D00 THEN MPHI Test if need decremen
188 5EEA D003 BNE NODEC both BYCLO & BYCHI bytes
189 5EEB CDE5E DEC BYCHI No, only lo-byte
190 5EEC ADD55E NODEC DEC BYCLO Yes
191 5EEF ODE5E LDA BYCHI Prepare for testing end
192 5EF1 8D00 ORA BYCLO Set/reset zero bit
193 5EF4 80 RTS
194
195
\end{verbatim}
*************** Subroutine DATAIN ***************

Gets a data byte from ACIA and returns data in A.

*************** Subroutine DATAO ***************

Sends the data byte in A to ACIA for transmission.

This area is initialized by BASIC program.

Image of MPLO

Image of MPH

Byte-count lo-byte

Byte-count hi-byte

SDK-8S start addr lo-byte

SDK-8S start addr hi-byte

Checksum lo-byte

Checksum hi-byte

Message byte

*************** Command Table ***************

CMDBY .BYTE '0' TRANSMIT command byte

CMDBY .BYTE '1' RECEIVE command byte

CMDBY .BYTE 'R' RUN command byte

CMDBY .BYTE 'E' RESET command byte

END
APPENDIX D - ENHANCEMENT SYSTEM EXECUTIVE PROGRAM

24 REM SETUP INFLAG & OUTF<AG FROM DEFAULT
25 X=PEEK(10950): POKE 8993,X: POKE 8994,X
26 REM CHECK FOR E000 MEMORY
27 FOR SC=1TO30:PRINT:NEXT
28 IF PEEK(57OB8)=223 THEN POKE794,37
29 PRINT"SDK-B5 EXTENDED MONITOR & CROSS ASSEMBLER SYSTEM EXECUTIVE"
30 PRINT:PRINT "JULY 25, 1982 RELEASE": PRINT
31 GO TO 100
32 PRINT:PRINT: INPUT "SELECT FUNCTION (1-4)";A
33 ON A GOTO 500,800,300,10000
34 PRINT:PRINT "FUNCTIONS AVAILABLE:":PRINT:PRINT
35 PRINT "(1) EXTENDED MONITOR - INTERCHANGE, MODIFY, & FILE DATA"
36 PRINT "(2) EDITOR - EDIT THE 8080/8085 SOURCE LANGUAGE FILES"
37 PRINT "(3) ASM85 - ASSEMBLE THE 8080/8085 SOURCE LANGUAGE FILE"
38 PRINT "(4) FREE - FREE SYSTEM FOR USER PROGRAMMING"
39 GOTO 60
40 REM
41 REM ASM85 - ASSEMBLER
42 REM
43 REM CHANGES LOWER WORKING LIMIT TO $53FF
44 POKE 133,83
45 GOSUB 2000
46 RUN"ASM85"
47 REM
48 REM EXTENDED MONITOR
49 REM
50 REM CHANGES LOWER WORKING LIMIT TO $57FF
51 POKE 133,85
52 DISK!"CALL 5600=36.1": REM BRING ASSEMBLED DATA TO BUFFER
53 DISK!"CALL 5600=39.1": REM S302 PROG.IN
54 GOSUB 2000
55 RUN"OSI-85"
56 REM
57 REM EDIT
58 REM
59 REM CHANGES LOWER WORKING LIMIT TO $57FF
60 POKE 133,87
61 GOSUB 2000
62 RUN"EDIT"
63 REM
64 REM FREE THE SYSTEM FOR USER PROGRAMMING
65 REM
66 REM ENABLE "REDO FROM START"
67 POKE 2993.2B;POKE 2994.11
68 REM "DISABLE ", & ":"
69 POKE 2972.13; POKE 2976.13
70 RETURN
71 REM
72 REM ENABLE CONTROL-C
73 POKE 2073.173
74 REM PRINT "SYSTEM FREE": PRINT PRINT "11645 BYTES AVAILABLE"
75 REM
76 REM 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"
3 REM Display & define pseudo memory range and command array
4 BS=22016:GOSUB 30500:GOSUB 40000
5 REM Recover User Directory
6 DISK!"CALL 5F00=39,Z"
7 A=24320: FOR Y=1 TO 5: T$=""
8 N=PEEK(A):A=A+1:IF N>7 GOTO 100
9 FOR Y=1 TO N:T$=T$+CHR$(PEEK(A))$:A=A+1:NEXT Y
10 F$(X)=T$;P(X)=PEEK(A):A=A+1:NEXT X
11 GOTO 502: REM To start command recognition
12 GOTO 600: REM DUMP Routine entry
13 GOTO 700: REM RUN Routine entry
14 GOTO 800: REM RESET Routine entry
15 GOTO 900: REM EXAM Routine entry
16 GOTO 1000: REM SUBSTITUTE Routine entry
17 GOTO 1100: REM INSERT Routine entry
18 GOTO 1200: REM ERASE Routine entry
19 GOTO 1300: REM SAVE Routine entry
20 GOTO 1400: REM LOAD Routine entry
21 GOTO 1500: REM PRINT Routine entry
22 GOTO 1600: REM MOVE Routine entry
23 GOTO 1700: REM SEE/SET Routine entry
24 GOTO 1800: REM CREATE Routine entry
25 GOTO 1900: REM CHAIN Routine entry
26 GOTO 2000: REM GUIT Routine entry
27 FOR Y=X TO 5:F$(Y)="???":NEXT
28 REM
29 PRINT
30 INPUT "Command":A$
31 N=LEN(A$):T=ASC(LEFT$(A$,1»):IF T<65 OR T>SO GOTO 30000
32 FOR K=1 TO N
33 T=ASC(MID$(A$,K,1»)
34 IF T>64 AND T<91 THEN NEXT
35 CM$=LEFT$(LEFT$(A$,K-1),2)
36 J=N-(K-1):CHK=O
37 REM Check with Command Array entries
38 FOR X=1 TO 15
39 ON CHK GOTO 30000,30050,30100
40 REM Syntax error
41 REM
42 DISK!"GOSUB 10000: REM Call PARSE
43 ON CHK GOTO 30000,30050,30100,30300
44 ON CHK GOTO 30000: REM Syntax error
45 REM
46 REM ***** DUMP Command Routine
47 GOSUB 10000: REM Call PARSE
48 ON CHK GOTO 30000,30050,30100,30300
49 ON CHK GOTO 30000: REM Syntax error
50 REM
51 REM ***** GET Command Routine
52 GOSUB 10000: REM Call PARSE
53 ON CHK GOTO 30000,30050,30100,30300
54 REM Extend the end of simulating range if necessary
55 IF EN>DN THEN DN=EN:D=DN:F=2:GOSUB 20500
56 ON CHK GOTO 30000: REM Syntax error
57 REM
58 REM ***** RUN Command Routine
59 REM Use default value if no specification
60 IF J=O THEN NS=ST:GOTO 720
61 ON CHK GOTO 30000,30050,30100
62 ON CHK GOTO 30000: REM Use specification
63 ON CHK GOTO 30000,30050,30100
718 IF J-(K+3)<0 GOTO 30000
720 LO=71: GOTO 11500: REM To LINK
730 REM
750 REM ***** RESET Command Routine
760 LO=92: GOTO 11640: REM To LINK with only command
770 REM
810 DP=I: REM Set flag For screen display
815 GOSUB 10000: REM Call PARSE
920 ON CHK GOTO 30000, 30050, 30100, 30300
925 DS=NS: REM NS will be redefined
928 GOSUB 7000: REM Call DISPLAY
940 IF LEFT$(A$,1)<"Y" GOTO 504
850 8C=256: GOTO 828
860 REM
1000 REM ***** CREATE Command Routine
1010 REM Display the current Directory
1030 GOSUB 1200: INPUT "Are you sure (Y/N)U;B$"
1040 IF LEFT$(3$,1)="IN" GOTO 504
1050 PRINT: INPUT "Enter net...:FILE name":B$=LEFT$(B$,7)
1060 PRINT: INPUT "At which storage location (1-5)";A$
1070 T=VAL(A$): IF T=0 OR T>5 GOTO 1080
1075 REM Define Filename to Directory & display updated Directory
1080 F$(T)=B$: GOSUB 1200: INPUT "Create another file (Y/N)U;S$" GOTO 920
1090 REM
1100 REM ***** DISPLAYDIR Subroutine
1110 REM Display the current Directory on screen
1120 PRINT: PRINT "-- DIRECTORY --": PRINT: PRINT "LOC... FILE NAME": PRINT
1130 FOR X=1 TO 5: PRINT X; F$(X): NEXT
1140 PRINT: RETURN
1150 REM
1200 REM ***** SUBSTITUE Command Routine
1205 IF J=0 GOTO 30050: REM No specifications
1210 P=2: GOSUB 20200: ON CHK GOTO 30000, 30050, 30100, 30300
1220 IF NS-ST=2043 THEN PRINT "The end of buffer": GOTO 504
1230 NS=NS+1: D=NS: GOSUB 11200: REM For next prompt message
1240 PRINT: PRINT "Substitute "HEX$" with";
1250 INPUT A$: IF LEFT$(A$,1)="N" GOTO 504
1260 J=LEN(A$): GOSUB 20700: ON CHK GOTO 30000, 30050, 30100
1270 GOTO 1630
1280 REM
1300 REM ***** QUIT Command Routine
1305 REM Save the current Directory to disk before exiting
1310 A=24320:FOR X=1 TO S
1320 N=LEN(F$(X)):POKE A,N: A=A+1
1330 POKE A,ASC(MID$(F$(X),Y,1))":A=A+1:NEXT Y
1340 POKE A,P(X):A=A+1:NEXTX
1350 DISK!"SAVE 39,2=5FO0/1":"RUN BEXEC*".
1360 REM
1400 REM ***** INSERT Command Routine
1405 IF J=0 GOTO 30050
1410 P=1: GOSUB 20200: ON CHK GOTO 30000, 30050, 30100, 30300
1420 IF DN+D-ST>2043 OR NS>DN GOTO 30300
1430 REM Move block down
1440 BC=(DN-NS)+1: SA=SA+(DN-NS): F=1: GOSUB 20500
1450 REM Extend the end of simulating range
1460 DISK!"SAVE 39,2=5FO0/1":"RUN BEXEC*".
1470 REM
3200 REM ***** ERASE Command Routine
3205 IF J=0 GOTO 30050
3210 P=1: GOSUB 20200: ON CHK GOTO 30000,30050,30100,30300
3215 IF NS>DN GOTO 30300
3220 BC=(DN-NS-D)+1: IF BC<0 GOTO 30000
3225 REM Move data block up for deletion
3230 SA=5A+D: F=-1: D=-D: GOSUB 20500: GOTO 2450
3240 REM ***** CHAIN Command Routine
3510 GOSUB 8000: IF X>5 GOTO 30400
3515 REM Calculate the pages of the file in buffer
3520 GOSUB 15000: IF P+P(X)<>8 GOTO 30300
3525 REM Load the specified disk file
3530 T$=RIGHT$(STR$(X+30),2): SA=8S+4+(DN-ST)+1: D=SA:
3540 DISK:"CA 5600;"Fl=5600;/P$: GOTO 11690
3545 REM Extend the simulation range to include the disk file
3550 D=-4: SA=SA+4: F=-1: GOSUB 20500: GOTO 11890
3560 REM ***** SAVE Command Routine
4020 GOSUB 8000: IF X<>S GOTO 30400
4030 T$=RIGHT$(STR$(X+30),2): P(X)=P:
4040 DISK:"SA 5600;"Fl=5600;/P$: GOTO 11690
4050 REM ***** LOAD Command Routine
4520 GOSUB 8000: IF X<>S GOTO 30400
4530 T$=RIGHT$(STR$(X+30),2): D=15K!
4540 GOSUB 30500: GOTO 11890
4550 REM ***** PRINT Command Routine
4820 IF LEFT$(B$Fl)="N" GOTO 815
4830 PRINT: INPUT "Title" ; B$: PRINT#1,BS: GOTO 81S
4840 REM ***** MOVE Command Routine
5610 IF J=0 GOTO 30050
5620 GOSUB 20100: ON CHK GOTO 30000,30050,30100
5630 IF J-(K+3)=0 GOTO 30050
5640 MS=NS: J=J-(K+3): P=4: GOSUB 20200
5650 ON CHK GOTO 30000,30050,30100,30300
5660 EN=DN: BC=(EN-NS)+1: ID=MS-NS
5700 IF BC<(MS-ST)>2044 OR EN>DN OR NS<ST GOTO 30300
5710 REM Check move upward or downward
5720 IF NS<DN THEN F=-1: DISK:"CA 5600;"Fl=5600;/P$: GOTO 11690
5730 F=1:SA+SA+BC-1: DISK:"CA 5600;"Fl=5600;/P$: GOTO 11690
5740 REM For upward movement only
5750 IF EN<DN GOTO 11690: REM No need to reduce the end
5760 REM Change the end of simulation range
5770 DN=MS+BC-1: F=2:ID=DN:DISK:"CA 1000;"Fl=5600;/P$: GOTO 11690
5780 REM For downward movement only
5790 IF MS+BC-1<DN GOTO 11690
5800 GOTO 5710
5740 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
6470 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: ST:F=0: DISK:"CA 20500: GOTO 20600; DN=EN:ID=F=2: DISK:"CA 20600: GOTO 11690
6550 REM
6600 J=LEN(A$): P=4: GOSUB 20700: RETURN
**DISPLAY Subroutine**

7000 REM **** DISPLAY Subroutine
7010 NS=INT(DS/16)*16: BK=DS-NS:T=16-BK
7020 PRINT 7030 PRINT" 0 1 2 3 4 5 6 7 8 9 A B C D E F"
7040 DATA 0 1 2 3 4 5 6 7 8 9 A B C D E F"
7050 REM For the first row only
7060 REM Fill blanks
7100 FOR X=1 TO T
7110 IF BK=0 GOTO 7220
7120 DSP$=DSP$+BK: BK=9K-1: GOTO 7190
7130 IF BK=0 THEN GOSUB 7270
7140 NEXT X
7150 GOSUB 7270: NS=NS+16: IF BC=0 THEN RETURN
7160 REM For the rest of rows
7200 FOR X=1 TO 16
7210 DS=DS+1: SA=SA+1: BC=BC-1: IF BC=0 GOTO 7380
7220 NEXT X
7230 GOSUB 7270: IF BC=0 THEN RETURN
7240 NS=NS+16
7250 GOTO 7310
7260 REM
7270 GOSUB 7500: IF BC=0 THEN RETURN
7280 REM Parse the address specification Field to return either the specified value(s) or the default value(s)
7300 IF J<>0 GOTO 10040
7310 SA=BS+4: NS=ST: BC=(DN-ST)+1: EN=DN: GOTO 10150
10000 REM ***** PARSE Subroutine
10005 REM Parse the address specification Field to return either the 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 Digits From specification Field
10505 REM Get Digits From specification Field
10510 B$=RIGHT$(A$,J)
10520 FOR K=1 TO J
10530 T=ASC(MID$(B$,K,1))
10540 IF T>47 AND T<59 GOTO 10580
10550 IF T>64 AND T<71 GOTO 10580
10560 NEXT K
10570 CHK=1: RETURN
10580 DG$=MIDS(BS,!,<,I):T=!...EN(DG$):rF
10590 T>I THEN CHK=3:RETURN
10600 FOR N=1 TO !
10610 T=ASC(MID$(DG$,N,1»
10620 IF T<48 OR T>70 THEN CHK=3:RETURN
10630 IF T>84 THEN D=91
10640 IF T<64 THEN D=92
10650 NEXT I
10660 RETURN
10670 REM
10680 REM ***** HTOD Subroutine
10690 REM Convert input HEX$ to the equivalent decimal output in D
10700 L=LEN(HEX$):D=0
10710 FOR I=1 TO L
10720 N=L-I:T=ASC(MID$(HEX$,N,1»
10730 SI=D+IG~(r-l)*(T-55):S2=D+15*(I-1)*(T-48)
10740 IF T>G4 THEN D=51
10750 IF T<64 THEN 0=52
10760 NEXT I
10770 RETURN
10780 REM
10790 REM ***** DTOH Subroutine
10800 REM Convert the input D to the equivalent 4-digit hex in HEX$
10810 TD(O)=D
10820 FOR I=1 TO 4
10830 TD(I)=INT(TD(I-l)/18):TP(I)=TD(!-!)-TDCI)*15
10840 N=!:IF INT(TD(I»=O GOTO 11280
10850 NEXT I
10860 FOR I=1 TO N
10870 TE$(N+l-I)=CHR$(48+TPCI»
10880 IF TPC!»9 THEN TE$(N+1-I)=CHR$(S5+TP(I»
10890 NEXT I
10900 HEX$=""
10910 FOR I=1 TO N
10920 HEX$=HEX$+TE$(I»:NEXT
10930 IF N=4 THEN RETURN
10940 HEX$="0"+HEX$+N+1:GOTO 11370
10950 REM
10960 REM ***** LINK Routine - A linkage between BASIC & machine subs
10970 REM Place the SDK-85 starting address
10980 D=NS:GOSUB 11800:SH=DH:SL=DL:IF LO=71 GOTO 11630
10990 REM Place the byte-count
11000 D=8C:GOSUB 11800:POKE 24288,DH:POKE 24287,DL
11010 REM Place the OSI local starting address
11020 D=SA:GOSUB 11800:POKE 24295,DH:POKE 24294,DL
11030 POKE 24291,O:POKE 24292,0: REM Clear checksum bytes
11040 POKE 24293,SL:POKE 24290,0
11050 REM Zero MSG byte and Set up machine subroutine entry address
11060 POKE 24299.O:POKE 9855,LO:POKE 9856,94
11070 REM Call the corresponding machine subroutine
11080 POKE 64512.2:USR(1•):POKE 64512.2
11090 CHK=PSEK(24299): REM Check communication error status in MSG
11100 ON CHK GOTO 30120,30140,30160
11110 PRINT:"PRINT"Done":GOTO 504
11120 REM
11130 REM ***** SPLIT Subroutine
11140 REM Split input D to two decimal-byte, DH and DL
11150 GOSUB 11200:TS=HEX$
11160 HEX$=LEFT$(TS,2):GOSUB 11000:DH=D
11170 HEX$=RIGHT$(TS,2):GOSUB 11000:DL=D:RETURN
11180 REM
11190 REM ***** CALCPAGE Subroutine
11200 P=INT((DN-ST)+3)/256):IF P*256=(DN-ST)+3 THEN RETURN
11210 P=P+l:RETURN
11220 REM
STEND Subroutine

Gosub 11200: HEX = D*1000 + $100: RETURN

Gosub 11200: HEX = D*1000 + $100: RETURN

GETNS Subroutine

Get the specified starting address value in NS

SCAN Subroutine

Translate specification field with no default options

UPDN Subroutine

Move a block of data upward or downward

CHANGE Subroutine

Change simulating start or end boundary to D

GETDATA Subroutine

Get P-digit of data from specification field & return value

SHOW Subroutine

Define & display the simulating range by the 1st 4 bytes of

DEFINE Command Array Subroutine

DIM CT$(15)
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)
38 REM
38 REM Definitions
40 DIM I$(291),I(290):FOR X=1 TO S:READ T$:C$(X)=T$:NEXT
80 DATA "I","N","F","C","L","P","D","E","G"
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»: IF T<65 AND T>80 GOTO 20000
122 REM
125 REM Isolate the leftmost character of the syntax field
130 FOR K=1 TO N: T=ASC(MID$(A$,K,1»: IF T>G4 AND T<91 THEN NEXT
155 REM
160 REM Check with the command array
170 FOR K=1 TO S: IF M$<>C$(K) THEN NEXT
190 ON K GOTO 200, 500, 600, 800, 1000, 2000, 3000, 4000, 4500
195 GOTO 20000: REM Syntax error
198 REM
200 REM ***** INPUT Command Routine
205 PRINT
210 IF I=280 GOTO 20300: REM Test if reach maximum line limit
220 INPUT A$: N=LEN(A$): IF VAL(LEFT$(A$,1»=0 GOTO 120: REM May be command
222 REM
225 REM Shrink the entered line
230 X=I+1: I$(X)=A$: GOSUB 9000: IF R=1 GOTO 20400: REM Violate space limit
232 REM
235 REM Fill the line number array
240 GOSUB 5000: IF I=0 OR (I$<I(!) GOTO 300: REM No new "line"
242 REM
245 REM Sorting Procedures - either replacement or insertion
250 FOR Y=1 TO I: IF I$(Y)<I$(Y) GOTO 270
255 REM Replace line Y with the new line
260 C=C-LEN(I$(Y)-1: I$(Y)=I$(Y)-1: I$(Y)=I$(Y)+1: GOSUB 8000: I$(Y)=T$=T$
270 IF I$(X)>I$(Y) GOTO 290
275 REM Insert the new line at Y and reposition the rest of the lines
290 NEXTY
300 I=I+1
305 REM Test if data-count overflown
310 C=C+N+1: IFC 4096 GOTO 210
320 REM Adjust the file by deleting the highest-numbered line
320 C=C-LEN(I$<I(!)-1: I$(I)="": I=I-1: R=2: IFC 4096 GOTO 330
350 GOTO 20300: REM To inform the user that file ends
360 REM
300 REM ***** NEW Command Routine
310 PRINT:PRINT "Ok": RUN 30: REM Clear all variables
320 REM
60 REM ***** FILE Command Routine
605 PRINT:PRINT "Dumping...": A=22528: P=1: FOR X=1 TO I
608 REM Load the character-count of that line
610 N=LEN(I$(X)): POKEA, N+1: GOSUB 7000: REM Test if needed 2nd track
615 REM Load characters of that line
620 FOR X=1 TO S: ASC(MID$(I$(X),Y,1»: POKEA, T:=T:=GOSUB 7000: NEXTY
630 NEXTX: REM Continue the next line
632 REM
635 REM Install the file-end mark
188

G40

POKEA,0:IFP=1THENGOSUB750:GOTOD20500
645 REM Store the current buffer to proper file track
650 IFV=0THENPRINT"SAVE 38,1=8800/2":GOSUB20500
660 DISK:"SAVE 38,1=8800/2":GOTOD20500
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 GOSUB20500:RETURN
720 REM
750 REM ***** SAVEFIRST Subroutine
751 REM Filemode flag guides the buffer to be saved to track 37 or 29
755 IFV=OTHENDISK!IFSAVE 37,1=5800/8" :RETURN
760 DISK!"CALL 5800=37,1" :GOT0930
770 REM
800 REM ***** CALL Subroutine
801 REM Load either track 37 or 29 to buffer
802 IFV=0THENPRINT"CALL 5800=37,1":GOTOB10
804 DISK!"CALL 5800=29,1"
810 PRINT:PRINT"Recovering...";I=0;C=0;X=I;A=22528
820 B=1:
830 EM Test if the character-count byte is the end mark (0)
840 IFV=0THENPRINT:RETURN
850 IFV=OORJ=0GOTOI10: REM Do nothing when no specifications
860 O=2:GOSUB8000:0NRGOT02000
870 REM
900 REM ***** DELETE Subroutine
901 REM Load the 2nd track of corresponding file if necessary
905 IFV=2GOSUB20500:
910 DISK!"CALL 5800=38,1":GOT0930
920 A=22528:RETURN
930 REM
1000 REM ***** LIST Command Routine
1005 F=1: REM Set flag for screen display
1010 IFI=0GOT0110: REM Nothing to display
1020 IFJ=0GOSUB8000:0NRGOT02000,20200
1025 REM Call STEND to return the proper display range
1030 GOSUB8000:0NRGOT02000,20200
1050 REM
1080 REM
2000 REM ***** PRINT Command Routine
2010 F=2:GOSUB2010: REM Set flag for printer & Join LIST
2020 REM
3000 REM ***** DELETE Command Routine
3010 IFI=0GOSUB2010: REM Do nothing when no specifications
3015 REM Call STEND to return the exact deleting range
3020 GOSUB8000:0NRGOT02000,20200
3030 REM Prepare for deletion
3040 A=I-E;Y=I:IF(I-E)=F:FORi=STOEN;N=LEN(I$):C=C-N-1:NEXT
3050 REM Call MOVE for deletion and clear useless lines
3060 F=1;GOSUB8000:FORi=1TOY;i$(x)="":NEXT
3070 GOSUB20500
3110 REM
3120 REM
3150 REM ***** MOVE Subroutine
3150 REM Move a block of lines upward or downward
3155 REM
3200 REM
3250 REM ***** EXTEND Command Routine
3300 REM Go to clear all variables & enter Extend mode
3305 REM
4500 REM ***** GUIT Command Routine
4510 RUN"BE<EC": REM Exit Editor
4520 REM
5000 REM ***** PUTID Subroutine
5001 REM Isolate the line number & Place it to line number array
5010 FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT>47 AND T<58 THEN NEXT
5020 I(X)=VAL(LEFT$(I$(X),K-1)): RETURN
5030 REM
5999 REM ***** DISPLAY Subroutine
6000 PRINTFORX=STOE: GOSUB9600: IFF=1 THEN PRINT$: GOTO9020
6010 PRINT#:1,T$
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 FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT>47 AND T<58 THEN NEXT
7040 NEXT
7050 R=1: RETURN
7060 A=K: FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT>47 AND T<58 THEN NEXT
7070 REM Get specification value and start searching
7080 J=J-(K-1): L=VAL(MID$(B$,A,(K-A)): T=I: FOR X=1 TO I
7090 ONO GOTO 7100, 7180, 7180
7100 IFL(T)=I(X) THEN RETURN
7110 IFL(T)<>I(X) THEN RETURN
7120 A=K: FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT<>45 THEN NEXT
7130 IF J-K=0 THEN J=0: E=I: GOTO 7190
7140 IF J<>0 THEN GOTO 7190
7150 E=T
7160 IFF<>0 DO S=OTHE<EO: I=1: RETURN
7170 RETURN
7180 RE<EM
7190 REM
8000 REM ***** STEND Subroutine
8005 REM Interpret the specification field with default value(s)
8010 "$=RIGHT$(A$,J): FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT<>45 THEN NEXT
8020 IF T=4 THEN E=S: RETURN
8030 E=S: RETURN
8040 GOSUB87000: IFR<>0 THEN RETURN
8050 S=X: IFF=0 THEN S=RETURN
8100 "$=RIGHT$(A$,J): FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT>45 THEN NEXT
8110 IFF-X=0 THEN J=0: E=T: GOTO 8190
8120 A=K: FOR K=1 TO J: T=ASC(MID$(B$,K,1)): IFT>45 THEN NEXT
8130 IF J-K=0 THEN J=0: E=I: GOTO 8190
8140 IF F<>0 THEN E=S: RETURN
8150 S=X: RETURN
8160 REM
8170 IF K<>0 THEN S=RETURN
8180 RETURN
8190 REM
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$(X),A,(K-A))
9030 REM Test if line ends
9040 REM
9050 IFX=1 THEN I=N<ENN(I$): T$=N=LEN(T$): RETURN
9060 REM Search non-space character
9070 A=K: GOSUB9400: IF A<>2 THEN R=1: RETURN
9080 IFK<>1 THEN GOTO 9050: REM Ignore the spaces at the end
9090 REM Collect one space and a repeat-count
9100 T$=T$+" "+CHR$«K-A)+64): A=K: GOTO 9020
9110 REM
9199 REM ***** SEARCHSPACE Subroutine
9200 FOR K=ATON: IF MID$(I$(X),K,1)<>"THEN NEXT
9230 RETURN
9250 REM
9259 REM ***** SEARCHCHARAC Subroutine
9260 FOR K=ATON:IF MID$(I$(X),K,1)=" " THEN NEXT
9270 RETURN
9280 REM
9299 REM ***** RECOVER Subroutine
9300 REM Recover a line to its original shape
9310 T$="";A=1:N=LEN(I$(X))
9320 GOSUB 9200:T$=T$+MID$(I$(X),A,(K-A)):IF K=N THEN RETURN
9330 REM Recover the space from the repeat-count
9340 A=ASC(MID$(I$(X),K+1,1))-64:FOR B=1 TO A:T$=T$+" II"
9350 A=K+2:GOTO 9620
9360 REM
9379 REM Error Procedures
9380 PRINT:PRINT"?Syntax error";GOTO 210
9390 PRINT:PRINT"?Not in list";GOTO 210
9400 PRINT:PRINT"?Over 25 spaces";PRINT:R=0;GOTO 210
9410 PRINT:PRINT"?Done";GOTO 210
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.
14 REM
15 REM
16 REM
17 REM BRING IN ALL ASCII DATA FOR THE TABLES
18 REM AND BUILD TABLES
19 REM
20 DIM B$(79), C$(79), T$(100), T(100) : REM MAX. 100 SYMBOLS
21 A=21504 : REM INIZ MEMORY PTR
22 REM
23 REM BUILD INSTRUCTION AND BASE-OPCODE TABLES
24 FOR X=0 TO 79: GOSUB 6000: BS(X)=T$: C(X)=T: A=A+1: NEXT
25 REM
26 REM BUILD REGISTER TABLE (B,E,D,H,L,M,A)
27 FOR X=0 TO 7: GOSUB 6000: R$(X)=T$: NEXT
28 REM
29 REM BUILD REGISTER PAIR TABLE (B,D,H,L,M,A)
30 FOR X=0 TO 3: GOSUB 6000: RP$(X)=T$: NEXT
31 REM
32 REM BUILD DIRECTIVE TABLE
33 FOR X=1 TO 6: GOSUB 6000: D$(X)=T$: NEXT
34 REM
35 REM
36 REM USER SELECTS PRINTER OR SCREEN, SET DISPLAY FLAG (0)
37 REM
38 PRINT ; PRINT
39 INPUT "List errors on printer instead of screen (Y/N)";A$
40 IF LEFT$(A$,1)="Y" THEN D=1: GOTO 90 : REM PRINTER
41 D=2 : REM SELECT SCREEN
42 REM PRINT "This is a slow assembler!": PRINT
43 PRINT "Begin assem...": PRINT
44 REM
45 REM PASS 1 : SET UP MEMORY LAYOUT AND DEFINE SYMBOLS
46 REM PASS 2 : FILL MEMORY WITH OP-CODES AND DATA
47 REM
48 REM
49 REM **** PASS 1 ENTRY
50 REM
51 REM INITIATES FLAGS AND POINTERS
52 REM
53 P=1 : Y=0 : E=0
54 REM
55 REM **** PASS 2 ENTRY
56 REM
57 REM BRING THE 1ST SOURCE TRACK IN
58 REM
59 REM
60 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
121 REM RECOVER STATEMENT BEFORE ';' AND RECOVER ONE SPACE
122 REM ONLY EVEN IF THERE ARE SEVERAL SPACES
123 REM
124 FOR X=1 TO N: I=PEEK(A): A=A+1: GOSUB 950
125 IF X=59 GOTD 150 : REM STOP IF HITS SEMICOLON
126 I=I$+CHR$(I)
127 REM CHECK IF SPACE THEN SKIP REPEAT-COUNT
128 REM
129 IF I=32 THEN X=X+1 : A=A+1 : GOSUB 950
130 NEXT X
131 REM
132 REM ADJUST SOURCE MEMORY PTR FOR NEXT LINE
133 REM
134 IF X-1=N GO TO 160 : REM NO ADJUSTING NEEDED
135 A=A+(N-X) : GOSUB 950 : REM A POINTS THE START OF NEXT LN
136 REM
137 REM GET LINE NUMBER
138 REM
139 REM
140 N=LEN(I$)
141 FOR X=1 TO N
142 T=ASC(MID$(I$,X,1)) : IF T>47 AND T<58 THEN NEXT
143 L=VAL(MID$(I$,1,1)) : REM L IS LINE NUMBER
144 IF L>0 GOTD 115 REM IT’S AN INSTRUCTION, NO SCANNING IN PASS 1
145 REM
146 REM ENTER THE FIRST FIELD SCANNING PROCEDURE
147 REM
148 GOSUB 900 : REM GET THE 1ST FIELD OF CHAR.
149 IF R=1 GOTO 115 : REM NO CHAR. BACK FOR NEXT LINE
150 REM
151 REM CHECK IF IT’S DIRECTIVE ('EGU' IS NOT ALLOWED IN THE 1ST.FIELD)
152 REM
153 REM ONLY PASS 2 NEEDS SCANNING DATA FIELD
154 REM
155 IF P=2 THEN ON 2 GOTO 1000,2000,2500
156 S=S+Z : U=U+Z : REM NO EFFECT EVEN Z=0
157 IF Z>0 GOTO 115 : REM IT’S AN INSTRUCTION, NO SCANNING IN PASS 1
158 IF P=2 GOTO 240 : REM NO SYMBOL BE DEFINED IN PASS 2
159 REM
160 REM DEFINE SYMBOL (THE 1ST.FIELD AND PASS 1 ONLY)
161 REM
162 GOSUB 8500
163 REM
164 REM
165 GOSUB 900 : ON I GOTO 500,9000,290,3500,4000,4500
166 REM
167 REM CHECK IF IT’S INSTRUCTION
168 REM
169 REM
170 REM ENTER THE SECOND FIELD SCANNING PROCEDURE
172 REM
173 GOSUB 8900 : REM CHECK IF MULTI.DEFINED
174 IF T<Y THEN R=2 : GOTO 8700 : REM YES, ERROR!
175 IF Y>100 THEN R=3 : GOTO 8700 : REM SYMBOL TB OVERFLOW!
176 T*G=LEFT$X(G$+6) : REM TAKE FIRST 6
177 T=Y+1 : REM DEFINE VALUE (CURRENT ADDR.)
178 Y=Y+1 : REM INCREMENT SYMBOL PTR
179 REM
180 REM **** ENTER THE SECOND FIELD SCANNING PROCEDURE
182 REM
183 GOSUB 900 : IF R=1 GOTO 8700 : REM NO CHAR. SYNTAX ERROR
REM
REM CHECK IF IT'S DIRECTIVE ('ORG' & 'END' ARE NOT ALLOWED TO BE PRESENTED)
REM
GOSUB 980: ON I GOTO 280, 280, 3000, 3500, 4000, 4500
REM
REM CHECK IF IT'S INSTRUCTION
REM
GOSUB 8500
REM IF P=1 AND Z>0 THEN S=S+Z: U=U+Z: GOTO 115
ON Z GOTO 1000, 2000, 2500: REM PASS2 OR NON-MNE AT PASS1
R=1: GOTO 8700: REM CAN NOT RECOGNIZE
REM
REM
REM SET MEMORY POINTER TO NEW VALUE.
REM NEW START LESS THAN LAST ORG IS NOT ALLOWED.
REM ----------------------------------------
REM =1: REM SET FLAG TO INDICATE ASCII ARE NOT ALLOWED
GOSUB 5000: REM GET DATA FIELD VALUE
IF R>0 GOTO 8700
REM
REM CHECK NEW START VALUE
REM
T=D-U: IF T<0 THEN R=4: GOTO 8700: REM NOT ALLOWED
REM
REM CHECK IF IT'S THE FIRST ORG
IF F=1 THEN U=D: F=2: GOTO 560: REM THE 1ST
REM
REM THE OTHERS
REM
GOTO 1090: REM TO CHECK ERROR AND EXIT
REM
REM
REM ------------ DIRECTIVE: ORG ------------
REM
REM... INSTRUCTIONS...
REM... INSTRUCTIONS...
REM... INSTRUCTIONS...
REM... INSTRUCTIONS...
REM... INSTRUCTIONS...
REM
REM
REM ------------ CHECK EXTEND -----------
REM
REM CHECK EXTEND LAG. IF IT WAS SET, THEN
REM IT'S NO 'END' ERROR OTHERWISE SET FLAG
REM ENTER EXTEND MODE.
REM
REM ----------------------------------------
REM
REM IF F2=2 THEN R=9: L=0: GOTO 8700: REM NO 'END'
REM
F2=2: REM SET EXTEND FLAG
DISK! "CALL 5800=29,1"
A=22528: GOTO 115: REM RESET PTR AND CONTINUE
REM
REM
REM ---------- SUBROUTINE: ISOLATE ---------
REM
REM SCANNING 1$ UNTIL HITS THE DELIMITER THEN
REM RETURNS WITH CHARACTERS OR ERROR MESSAGE.
REM
REM ENTRY: X= THE POSITION OF START
REM
REM RETURN X= THE POSITION OF DELIMITER
REM
REM R= ERROR CONDITION
REM
REM
REM
REM
REM
REM IF X>N GOTO S21: 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=1 TO N : I=ASC(MIDS(IS$,[I$.K],1))
916 IF I>47 AND I<59 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(MIDS(IS$,[X$.X],1))
930 IF I=58 OR I=44 OR I=32 GOTO 948 : REM HITS DELIMITER
932 NEXT X
942 REM
946 G$=MIDS(IS$,[X$.X-K]) : RETURN
947 REM
948 REM
949 REM ------- SUBROUTINE: CHKBUFF -------
950 REM
951 REM CHECK IF NEEDS TO BRING THE 2ND TRACK TO
952 REM BUFFER. IF SO, RESET SOURCE MEMORY PTR.
954 REM -----------------------------
955 REM
960 IF A<24576 THEN RETURN : REM NO NEED
962 IF F2=1 THEN
965 DISK"CALL 5800=38,1": GOTO 970
967 DISK"CALL 5800=30,1" : REM EXTENDED MODE
970 A=22528+(A-24576) : RETURN
971 REM
972 REM
973 REM ------- SUBROUTINE: CMPDIR -------
974 REM COMPARE CHARACTERS (G$) WITH
975 REM TABLE. RETURN WITH I (1-7)
979 REM -----------------------------
985 REM
990 FOR I=1 TO G: IF G$<>D$(I) THEN NEXT
992 RETURN
993 REM
994 REM
995 REM ------- ONE-BYTE INSTRUCTION -------
1000 REM FILLS MEMORY BUFFER WITH OPCODE.
1003 REM ENTER WITH T POINTS THE FOUND MNEMONIC
1004 REM -----------------------------
1008 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 POkses OPCODE
1088 REM
1087 REM ENTRY OF CHECKING UNNECESSARY (EXTRA) OPERAND
1088 REM
1090 GOSUB 800 : IF R=1 GOTO 115 : REM NO MORE
1095 R=B : 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
1140 D=B+T : GOTO 1085 : REM EXIT OF 'MOV' AND A&L
1155 REM .....................................................
1160 REM
1170 REM ENTRY OF 'RST' (OPCODE=B+(0-7)*8)
1171 REM
1180 GOSUB 5000 : REM GET DATA (0-7)
1185 IF R=0 OR D=7 THEN R=8 : GOTO 8700 : REM ILLEGAL
1190 D=B*D+8 : GOTO 1085 : REM REENTER ONE-BYTE
1205 REM .....................................................
1210 REM
1220 REM ENTRY OF 'POP' & 'PUSH' (OPCODE=B+RP*16)
1225 REM
1230 GOSUB 1900 : REM B=B+RP*16 BACK
1235 IF R=0 GOTO 8700 : REM DATA FIELD ERROR
1240 GOTO 1080 : REM EXIT OF 'POP' & 'PUSH' AND RP FAMILY
1250 REM .....................................................
1260 REM
1270 REM ENTRY OF 'INR' & 'DCR' (OPCODE=B+R*8)
1275 REM
1280 GOSUB 1900 : REM B=B+R*8 BACK
1285 IF R=0 GOTO 8700
1290 GOTO 1080 : REM REENTER ONE-BYTE
1300 REM .....................................................
1310 REM
1320 REM ---------- SUBROUTINE: CHKRGTR ----------
1321 REM
1322 REM GET NEXT FIELD OF CHARACTERS AND COMPARE
1323 REM WITH REGISTERS TABLE. RETURN WITH T POINTS
1324 REM THE FOUND REGISTER, OR ERROR BACK.
1325 REM -----------------------------------------
1326 REM
1327 GOSUB 900 : IF R=0 GOTO 1720
1332 R=R RETURN : REM NO CHAR.OR NOT MATCH ERROR
1333 REM PUT SP BACK
1335 GOTO 1080 : REM EXIT OF 'POP' & 'PUSH' AND RP FAMILY
1339 REM .....................................................
1340 REM
1345 REM ENTRY OF 'INR' & 'DCR' (OPCODE=B+R*8)
1349 REM
1350 GOSUB 1900 : REM B=B+R*8 BACK
1355 IF R=0 GOTO 8700
1360 GOTO 1080 : REM REENTER ONE-BYTE
1365 REM .....................................................
1370 REM
1375 REM ------- SUBROUTINE: CHKRGTR -------
1380 REM
1381 REM GET NEXT FIELD OF CHARACTERS AND COMPARE
1382 REM WITH REGISTERS TABLE. RETURN WITH T POINTS
1383 REM THE FOUND REGISTER, OR ERROR BACK.
1384 REM -----------------------------------------
1385 REM
1386 GOSUB 900 : IF R=0 GOTO 1720
1391 R=R RETURN : REM NO CHAR.OR NOT MATCH ERROR
1396 REM
1397 REM COMPARE WITH TABLE
1398 REM
1400 FOR T=0 TO 3 : IF GS=R$(T) THEN RETURN
1405 NEXT T
1410 GOTO 1715 : REM CAN NOT FIND
1415 REM
1420 REM
1800 REM ---------- SUBROUTINE: GETRP ----------
1801 REM
1802 REM GET REGISTER-PAIR VALUE (B=0,D=1,H=2,SP OR PSW=3)
1804 REM RETURN WITH B=BASE+RP*16
1805 REM -----------------------------------------------------
1806 REM
1810 GOSUB 900 : IF R=1 GOTO 1840 : REM NO CHAR. ERROR
1815 REM
1816 REM COMPARE WITH TABLE
1817 REM
1820 FOR T=0 TO 3
1825 IF G$(T)=RP$(T) THEN B=B+T*16 : RETURN : REM FOUND
1835 NEXT T
1840 R=8 : RETURN : REM CANT FIND
1845 REM
1855 REM
2000 REM ---------- TWO-BYTE INSTRUCTIONS ----------
2001 REM
2002 REM FILLS MEMORY BUFFER WITH OPCODE AND 1-BYTE DATA. ENTER WITH T POINTS THE POSITION OF THE MNEMONIC IN THE TABLE.
2005 REM ---------------------------------------------
2010 B=C(T) REM GET BASE OPCODE
2020 IF T>46 GOTO 2070 REM NOT 'MVI'
2025 REM
2030 GOSUB 1900 REM B=8+R*8
2035 REM
2040 GOSUB 1900 : REM B=8+R*8
2050 IF R=0 GOTO 8700
2055 REM
2060 REM REENTRY OF ALL 2-BYTES
2065 REM
2070 D=0 : GOSUB 4700 : REM POKE OPCODE
2080 GOSUB 5000 : REM GET OPERAND
2090 IF R=1 THEN R=6 : REM NO OPERAND ERROR
2100 IF R=0 GOTO 8700 : REM OTHER ERROR
2110 IF G$="""" GOTO 1090:REM ASCII DATA BEEN POKE 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
2300 REM ---------- THREE-BYTE INSTRUCTIONS ----------
2501 REM
2502 REM FILLS MEMORY BUFFER WITH OPCODE AND 2-BYTE DATA (ADDRESS). ENTER WITH T POINTS THE FOUND MNEMONIC.
2505 REM -----------------------------------------------------
2510 B=C(T) REM GET BASE OPCODE
2520 IF T>57 GOTO 2580 : REM NOT 'LXI'
REM SET FLAG TO PREVENT ASCII
GOSUB 5000 : REM GET D
IF R>O GOTO 8700 : REM ERROR
GOSUB 5000 : REM ERROR
IF R>0 GOTO 8700 : REM NO DATA
GOTO 8700 : REM ILLEGAL VALUE
IF D>65535 OR D<-2048 THEN R=7 : GOTO 8700 : REM ILLEGAL VALUE
GOSUB 4700 : REM POKE 2-BYTE
GOTO 1090 : REM EXIT
GOSUB 4700 : REM POKE OPCODE
GOSUB 5000 : REM GET 2-BYTE DATA
IF R=1 THEN R=6 : GOTO 8700 : REM NO ACTION AT PASS 2
Z=2 : REM SET FLAG TO PERMIT 1 ASCII
GOSUB 5000 : REM GET OPERAND IN DECIMAL
IF P=2 GOTO 115 : REM NO ACTION AT PASS 2
IF G$="" THEN R=1 : GOTD 8700 : REM ASCII HAD BEEN POKE
GOSUB 3070 : REM OTHER ERRORS
R=1 : GOTO 8700
Z=1 : REM SET FLAG TO PREVENT ASCII
GOSUB 5000 : REM GET D
IF R>0 GOTO 8700 : REM ERROR
REM GIVES VALUE TO THE SYMBOL JUST DEFINED
IF D>65535 OR D<-2048 THEN R=7 : GOTO 8700 : REM ILLEGAL VALUE
REM OPERAND CAN BE A DECIMAL, HEX, BINARY, DEFINED SYMBOL, OR AN ASCII DATA.
REM EXECUTES AT PASS 1 ONLY, ALL ERRORS WILL BE DISPLAYED IN ERROR 1.
REM REENTRY OF ALL 3-BYTES
D=B GOSUB 4700 : REM POKE 2-BYTE
GOTO 1090 : REM EXIT
REM RESERVES D BYTES OF MEMORY BUFFER
Z=1 : REM SET FLAG TO PREVENT ASCII
GOSUB 5000 : REM GET D
IF R>0 GOTO 8700 : REM ERROR
REM WORD FORM CAN BE EITHER DECIMAL, HEX, BINARY, OR SYMBOL. NO ASCII
REM WILL BE ACCEPTED.
Z=1 : REM SET FLAG TO PREVENT ASCII
GOSUB 5000 : REM GET FIRST DATA (WORD)
IF R>0 GOTO 8700
REM REENTRY OF THE NEXT WORD (IF MORE THAN ONE IN A LINE)
IF D>65535 OR D<-2048 THEN R=7 : GOTO 8700 : REM ILLEGAL VALUE
ENTRY: D=

RETURN: S=S+2, U=U+2

ENTRY: 0

RETURN: 8=5+1 & U=U+1

GETS DATA 8YTES FOLLOWING THE DB AND FILLS THOSE DATA INTO MEMORY.
DATA FORM CAN BE A COMBINATION OF DECIMAL, HEX, BINARY, DEFINED SYMBOL, AND A STRING OF ASCII.
AS LONG AS, BYTE VALUE IN THE RANGE OF -127 TO 255

GET HI-BYTE VALUE POKE HI POKE INCREMENTS MEMORY POINTERS
RELEASE FLAG TO ALLOW ASCII
GET DATA POKE WORD CHECK IF MORE NO MORE, EXIT ERROR EXIT
REM REM REM REM ----------------- DIRECTIVE: DB ----------------­
REM REM REM REM -------------------- SUBROUTINE: POKEWORD ----------------­
REM REM SPLIT S INPUT D TO 2 BYTES AND POKE S LOW, HIGH BYTE INTO MEMORY BUFFER IN SEQUENCE.
REM ENTRY: D= DATA WORD RETURN: S=S+2, U=U+2

------------------------------------------
REM
GOSUB 8100 : REM CONVERTS D TO 4 DIGITS HEX
GOSUB 8000 : REM GET LOW-BYTE VALUE
GOSUB 4700 : REM POKE LOW
H$=LEFT$(T$,2)
GOSUB 3000 : REM GET HI-BYTE VALUE
GOSUB 4700 : REM POKE HI
RETURN

------------------------------- SUBROUTINE: POKEBYTE ------------------------
POKES A INPUT BYTE (D) INTO NEXT AVAILABLE MEMORY BUFFER LOCATION THEN INCREMENTS THE POINTERS FOR NEXT POKING.
ENTRY: D = DATA BYTE RETURN: S=S+1 & U=U+1
------------------------------------------
POKE S, D : REM POKEING INCREMENTS MEMORY POINTERS
S=S+1 : U=U+1
POKE EACH ASCII INTO MEMORY

CHECK EACH CHARACTER IF Valid DECIMAL

CARRYING OUT ARITHMETIC PERMISSIBILITY AND SYNTAX ERROR

N C "T

ALL VALID NUMBERS

ILLEGAL VALUE

EXIT FOR C

NO COUNT ON 2 ""

CHECK ASCII PERMITTING FLAG AND SYNTAX ERROR

IF M>Z-1 OR RIGHT$(G@.1)<>"" THEN R=S: RETURN

REM TAKE 2 "" OFF

POKE EACH ASCII INTO MEMORY BUFFER

FOR I=1 TO M: D=ASC(MID$(G@.1)): GOSUB 4700: NEXT I

SET ASCII MESSAGE FOR RETURN
G$="" : RETURN
REM
--- HEXADECIMAL DATA ---
H$=LEFT$(G$,M-1) : REM GET RID OF TAIL "H"
REM
CHECK EACH CHARACTER IF VALID HEX
REM
FOR I=1 TO M-1 : T=ASC(MID$(H$,I,1))
IF T<48 OR T>70 GOTO 5350
IF T<48 OR T>70 GOTO 5350
NEXT I
IF I'S THEN R=7 : RETURN : REM 4 DIGITS AT MOST
GOSUB 8000 : RETURN : REM GET DEC.AND EXIT
R=G : RETURN : REM ERROR EXIT
REM
--- BINARY DATA ---
M=M-1 : G$=LEFT$(G$,M) : REM GET RID OF TAIL "B"
CHECK EACH CHARACTER IF 1 OR 0
FOR I=1 TO M : T=ASC(MID$(G$,I,1))
IF T<48 OR T>49 GOTO 5350 REM SHARE WITH HEX
NEXT I
IF I'S THEN R=7 : RETURN : REM 8 DIGITS AT MOST
GOSUB 8200 : RETURN : REM ERROR FREE EXIT
REM
---------- SUBROUTINE: CHKSIGN ----------
SCANNING A$ FOR PLUS OR MINUS SIGN CALLED BY ARITHMETIC OPERATION ONLY
ENTRY: C= POSITION OF STARTING RETURN: K= POSITION OF SIGN OR ENDING
-----------------------------------------
FOR K=C TO M
T=ASC(MID$(A$,K,1))
IF T<43 AND T>45 THEN NEXT K
RETURN
REM
---------- SUBROUTINE: RECOVER ----------
RECOVER THE INSTRUCTION MNEMONICS, BASE OPCODES, AND THE DIRECTIVES FOR TABLE BUILD-UP
ENTRY: A= POSITION OF NEXT CHARACTER RETURN: T$=CHARACTER T=BASE OP CODE
-----------------------------------------
T$="" : REM INIZ
IF T=0 GOTO 6040 : REM END FOR CHAR.
T$=T$+CHR$(T) : GOTO 6020 : REM RECOVER CHAR.
T$=PEEK(A) : RETURN
REM
---------- SUBROUTINE: HEX-DEC ----------
REM ----- SUBROUTINE: SYMBOL SEARCH ----- 
REM COMPARE G$ WITH DEFINED SYMBOL TABLE.
REM RETURN WITH T.
REM -------------------------------------
RETURN

REM ----- SUBROUTINE: SEARCH MNE -------
REM COMPARE G$ WITH ALL ENTRIES OF THE INSTRUCTION MNEMONIC
REM TABLE. RETURN Z AND T
REM --------------------------------------
RETURN

REM FOR T=O TO 79 : IF G$>B$(T) THEN NEXT T
REM
REM IF T<45 THEN z=1 : RETURN : REM 1-BYTE
REM IF T<57 THEN Z=2 : RETURN : REM 2-BYTE
REM IF T<60 THEN Z=3 : RETURN : REM 3-BYTE
REM
REM RETURN : REM NOT FIND

REM ----- SUBROUTINE: SYMBOL SEARCH ----- 
REM COMPARE G$ WITH DEFINED SYMBOL TABLE.
REM RETURN WITH T.
REM --------------------------------------
ERROR-FREE EXIT, STORE OBJECT CODES TO DISK

PREPARE FOR STORE OBJECT CODES TO DISK

ERROR DISPLAY PROCEDURE

DISPLAYS ERROR CODE AND LINE NUMBER.

ALWAYS BACK TO NEW LINE SCANNING.

ERROR CHECK PASS CONDITION

ERROR FREE EXIT, REQUEST DESTINATION

ERROR FREE EXIT, STORE OBJECT CODES TO DISK

ERROR FREE EXIT, STORE OBJECT CODES TO DISK
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=39.1":DISK!"CALL 5900=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 SEG SOURCE STATEMENT"
50 PRINT:PRINT:INPUT"List on printer instead of screen (Y/N)";P$
55 IF LEFT$(A$,1)="Y" THEN GOTO 1120: REM PRINTER GO
60 O=2:PRINT:PRINTB$:PRINT:PRINTC$:PRINT:GOTO100
65 PRINT#1:PRINT#1,B$:PRINT#1:PRINT#1:PRINT#1:PRINT#1
70 REM
80 REM Initialization
100 A=22528:S=21508:U=0:F=0:Y=0:F2=0:X$=If
101 REM
102 REM Entry of recovering a source statement
105 D$="8080/8085 CROSS ASSEMBLER, RELEASED 1982. E.E. OHIO U."
110 A=A+1:GOSUB 950: REM Update source buffer if need
112 IF T<>0 GOTO 115
115 T=PEEK(A); A=A+1:GOSUB 950: REM Update source buffer if need
120 IF T<32 GOTO 140
122 REM Recover spaces
125 FOR X=1 TO PEEK(A)-64
130 D$=D$+CHR$(T):I=!+1:IF I<N GOTO 115
135 A=A+1:GOSUB 950:I=I+2:GOTO 115
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: I=LEFT$(D$,N): REM Exclude comments
165 FOR X=1 TO N: REM Pass number characters
170 T=ASC(MID$(D$,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 B700: REM Comments
202 REM Check if directives
205 IF G$="ORG" GOTO 500
210 IF G$="END" GOTO 900
204

215 IF G$="DE" GOTO 4500
220 IF G$="DS" GOTO 3500
225 IF G$="DN" GOTO 4000
228 REM Check if mnemonics
230 GOSUB 8500: IF Z>0 GOTO 1000
232 REM Rebuild symbol table
233 $$(Y)=LEFT*(G$,G);T(Y)=U;Y=Y+1
236 REM
238 REM Second field scanning (must be either directive or mnemonic)
240 GOSUB 8300: GOSUB 8500: IF Z>0 GOTO 1000: REM Mnemonic
242 REM Either one of the following
245 IF G$="EGU" GOTO 3000
250 IF G$="DS" GOTO 3500
255 IF G$="DW" GOTO 4000
250 GOTO 4500: REM Then must be DB directive
255 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:1=4:GOSUB B100: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 S800=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 (=ASC(MIDS$(K,1))
915 IF I>47 AND I<58 GOTO 925
916 IF I>64 AND I<91 GOTO 925
918 IF I=39 GOTO 925
920 NEXT
922 R=1:RETURN
925 X=K: REM Mark the start position
930 FOR K=X TO N: REM Search delimiter or line end
932 (=ASC(MIDS$(X,1))
935 IF I>58 OR I=44 OR I=32 GOTO 946
940 NEXT
946 G$=MIDS$(K,X-K):RETURN
948 REM
950 REM ***** CHKBUFF Subroutine
955 REM Check if buffer needs the 2nd track file
960 IF A<247 THEN RETURN
965 IF F2=1 THEN DISK!'CALL S800=38,1'':GOTO 970
968 DISK!'CALL S800=30,1'
970 A=22528:RETURN
980 REM
1000 REM ***** INSTRUCTION Collection Routine
1005 REM Use the opcodes in opcode buffer
1010 D=U:1=4:GOSUB B100: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$:GOTO 8700
1080 REM
3000 REM ***** EQU Operation
3005 REM Rebuild the definition to symbol table
3010 GOSUB 5000:T(Y-1)=D:1=4:GOSUB B100:C$=H$:A$=Y$:B$=X$:GOTO 8700
3020 REM
3500 REM ***** DS Operation
3505 REM Increment Program Counter to a new setting
3520 $=H$:B=X$:S+S+C:U=U:C=GO TO 8700
3530 REM
4000 REM ***** DW Directive Data-Collection Routine
4005 REM Collect word(s) from the object code buffer
4020 C$=H$:S=X$:S=S+C:U=U+C:GOTO 8700
4030 REM
4040 REM ***** DB Directive Data-Collection Routine
4045 REM Collect byte(s) from the object code buffer
4060 REM
5000 REM ***** GETDATA Subroutine
5005 REM Get an operand value from the field characters
5010 GOSUB 5000:M=X-K:A$=G':C=1
5015 GOSUB 5500:IF K<M GOTO 5100: REM Arithmetic operand
5020 IF LEFT'(A$,1)<>'IF'GOTO 5030
5025 D=ASC(MID$(A$,2,1)»:RETURN: REM ASCII operand(s)
5030 GOSUB 8600:IF T<Y THEN D=T(T):RETURN: REM Symbol operand
5035 IF RIGHT$(A$,1)<>'IF'GOTO 5045
5040 M=M-1:G$=LEFT$(A$,M-1):GO SUB 8100:REM Binary operand
5045 D=VAL(G$):RETURN: REM Decimal operand
5050 REM
5060 REM ***** SETWORD Subroutine
5065 REM Set a word from the object code buffer
5070 D=PEEK(S):L=2:GOSUB 8100:S=S+1:U=U+1:RETURN
5080 REM
5090 REM ***** SETBYTE Subroutine
5095 REM Set a byte from the object code buffer
5100 D=PEEK(S):L=2:GOSUB 8100:S=S+1:U=U+1:RETURN
5110 REM
5120 REM ***** SEARSIGN Subroutine
5125 REM Search if any '+' or '-' sign in the source statement
5130 FOR K=C TO M
5135 T=ASC(MID$(A$,K-l,1))IF T<43 AND T<45 NEXT RETURN
5140 IF W=O 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
5170 REM ***** COUNTOPA Subroutine
5175 REM Count the number of succeeding operand(s)
5180 C=0
5185 IF LEFT$(A$,1)='' GOTO 5650
5190 C=C+1:GOTO 5610
C=C+((X-K)-2):GOTO 5610

REM

**** HEX-DEC Subroutine
J=LEN(H$):D=0
FOR I=1 TO J
T=ASC(MID$(H$,J+1-1,1))
SI=D+16^(I-1)*(T-55):S2=D+16^(I-1)*(T-49)
IF T>64 THEN D=SI
IF T<64 THEN D=S2
NEXT
RETURN

REM

**** DEC-HEX Subroutine
D(0)=D
FOR I=1 TO 4
D(I)=INT(D(I-1)/16):P(I)=D(I-1)-D(I)*16
NEXT
H$="":FOR I=1 TO J
E$(I)=CHR$(48+P(I))
IF P(I)<3 THEN E$(I)=CHR$(55+P(I))
NEXT
H$="":FOR I=1 TO J
H$=H$+E$(I):NEXT
GOTO 8135

REM

***** BIN-DEC Subroutine
0=0: FOR I=1 TO M
D=D+2^(I-1)*VAL(MIO$(G$,M-I,1))
NEXT
RETURN

REM

***** CHKMNEMONIC Subroutine
REM Check with mnemonic to see it is instruction mnemonic
FOR T=0 TO 79
IF G$<>B$(T) THEN NEXT
IF T<46 THEN Z=1:RETURN
IF T<57 THEN Z=Z:RETURN
IF T<80 THEN Z=3:RETURN
Z=0:RETURN
NEXT

REM

***** SEARSYMBOL Subroutine
REM Compare with symbol table entries
T$=LEFT$(G$,S)
FOR T=0 TO Y
IF T$<>T$(T) THEN NEXT
RETURN
NEXT
RETURN

REM

***** DISPLAY Subroutine
REM Organize a print statement for listing
DS=A$+-B$+-C$+-D$
GOSUB 8800:IF P=0 GOTO 105
RETURN

REM

***** PRINT Subroutine
REM Print a statement to either screen or printer
IF 0=2 THEN PRINT D$:RETURN
PRINT#1,D$:RETURN

REM

***** END Operation
REM Print the symbol table entries and exit
IF 0=2 THEN PRINT:PRINT:PRINT:GOTO 9050
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:DOSUB 8100:D$=D$+T$(K)+H$+X$;K=K+1
9540 IF K<Y THEN NEXT
9550 DOSUB 8900:IF K<Y GOTO 9510
9600 PRINT:"Ok":PRINT
9610 INPUT"Do you want to go to Loader (Y/N)?";A$
9620 IF LEFT$(A$,1)="N" GOTO 9640
9630 POKE 133,8$:DISK!="CALL 5600=39,1"
9635 DISK!="CALL 5600=39,1":RUN"C$-85"
9640 RUN"BEXEC*"