altair DISK OPERATING SYSTEM DOCUMENTATION

First Printing, June, 1977

mits
2450 Alamo S.E. / Albuquerque, New Mexico 87106

TABLE OF CONTENTS

APPENDICES

ALTAIR DOS DOCUMENTATION

SECTION I
INTRODUCTION

1. INTRODUCTION

1-1. Introduction to This Manual

The Altair Disk Operating System (DOS) is a system for developing and running Assembly Language programs. It consists of a Monitor and several system programs. The parts of this manual describe the various components of the system.

Chapter 2--the Monitor. The Monitor provides control and disk file management for all of DOS. Monitor Input/Output routines are available to any program running under DOS.

Chapter 3--the Text Editor. The Editor (EDIT) creates, modifies and saves ASCII coded files. Typical Editor files include Assembly Language programs and data.

Chapter 4--the Assembler. The Assembler (ASM) converts symbolic Assembly Language programs into relocatable machine code modules.

Chapter 5--the Linking Loader. The Linking Loader (LINK) loads the relocatable object code modules into memory, assigns addresses to symbols and resolves external references.

Chapter 6--Debug. Debug is a versatile symbolic debugging program. With Debug, the programmer can interrupt execution of a program, examine and modify the contents of register and memory locations.

Chapter 7--Miscellaneous System Programs.

Console (CNS) transfers command of the Monitor from one terminal device to another.

Initialize (INIT) allows the system parameters (amount of memory, number of disks, etc.) to be changed without reloading the system.

1-2. Loading and Initializing DOS

When the computer is first turned on, there is nothing of value in the semiconductor read/write memory. Therefore, before DOS can be used, the Monitor must be loaded from disk. This requires another program, the loader. The loader may reside in read-only memory or may be loaded from paper tape or cassette.

A. Systems with a Disk Boot Loader PROM mounted in the proper slot of a PROM Memory Card have the loader program readily available in non-volatile memory. Use the following procedure to load DOS with the DBL PROM:

1. Turn on the power to the computer, disk drives and peripherals.
2. Raise STOP and RESET simultaneously and then release them.
3. Raise switches A15-A8 and lower switches A7-AØ.
4. Actuate EXAMINE.
5. Make sure the DOS diskette is mounted in disk drive 0, that the door is closed and the disk has come up to speed (approximately 5 seconds).
6. Enter sense switch settings for the terminal I/O board from Table 1-A.
7. Press RUN.

DOS should start up and print MEMORY SIZE? For the remainder of the initialization procedure, see Section C below.

B. For systems without the DBL PROM, the loading procedure involves entering a bootstrap loader from the computer front panel, running it to load a disk loader program from paper tape or cassette and then running that loader to load the Monitor from disk. The procedure for doing this is as follows:

1. Turn on the power to the computer and peripheral devices.
2. Raise the STOP and RESET switches simultaneously and then release them.
3. Make sure the terminal is on-line (on a Teletype™, this means the mode switch is set to LINE).

Now enter the proper loader program for the device through which the loader tape is to be entered. The bootstrap loaders are in Appendix F.

The bootstrap loaders are entered on the front panel switches A7 - A0. Each switch has two positions, up and down. By convention, up is designated as 1 and down as 0. Therefore, the eight switches represent one byte of data. Each group of three switches, starting from the right, can represent the digits 0 through 7. The leftmost two switches represent the digits 0 through 3. For example, to enter the octal number 315, the switches A0 through A7 are set to correspond to the following table:

Switch	A7	A6	A5	A4	A3	A2	A1	A0
Position	up	up	down	down	up	up	down	up
Octal Digit	3			1			5	

The data bytes of the loader programs are shown in octal and are to be entered on A0 - A7 in this manner. To enter the programs:

4. Put switches A0 - A15 in the down position.
5. Raise EXAMINE.
6. Put the first loader program data byte in switches A0 - A7.
7. Raise DEPOSIT.
8. Put the next data byte in A0 - A7.
9. Depress DEPOSIT NEXT
10. Repeat steps 8 and 9 for each successive data byte until the loader is completely entered.

Now check the loader to make sure it has been entered correctly:

11. Put switches A0 - A15 in the down position.
12. Raise EXAMINE.
13. Check to see that the lights D0 - D7 correspond to the correct data byte for the first location. A light on indicates 1; off means 0. The rightmost three lights correspond to the rightmost octal digit. The next three lights represent the middle digit and the leftmost two lights represent the left digit.
    If the data byte is correct, go to step 16.
    If the data byte is not correct, go to step 14.
14. Put the correct value in switches A0 - A7.
15. Depress DEPOSIT.
16. Depress EXAMINE NEXT.
17. Check each successive byte by repeating steps 13 - 16 until the whole loader is checked.
18. If there were any incorrect bytes, check the whole loader again to see that they were corrected.

Now the paper tape or cassette labelled DISK LOADER can be read. For the paper tape version, put the tape in the reader and make sure it is positioned on the leader. The leader is the section of tape at the beginning with a series of 302_8 characters (3 of

8 holes punched). For the cassette version, put the cassette in the reader and make sure it is completely rewound.

19. Put switches A0 - A15 in the down position.
20. Raise EXAMINE.
21. Enter the proper sense switch settings for the load and terminal devices in switches A8 - A15. The rightmost four switches contain the load device setting, and the leftmost switches contain the setting for the terminal devices. Table 1-A shows both the octal sense switch setting and the load and terminal switches to be raised for each standard Altair system peripheral. If a device is used for interface to the terminal, the switches in the "Terminal Switches" column must be raised. If the device interfaces the peripheral through which DOS is being loaded, the "Load Switches" are raised.

	Sense Switch Setting	Terminal Switches	Load Switches	Channels
2SIO (2 stop bits)	0	None	None	20,21
2SIO (1 stop bit)	1	A12	A8	20,21
SIO	2	A13	A9	0,1
ACR	3	A13,A12	A9,A8	6,7
4PIO	4	A14	A10	40,41, 42,43
PIO	5	A14,A12	A10,A8	4,5
Non-Standard terminal	14			
No terminal	15			

22. Start the loading process. If the load device is connected to the computer through an 88-SIO A, B or C or an 88-PIO board, start the tape reader and then press the RUN switch on the computer front panel. For the 2SIO or 4PIO boards, press RUN and then start the reader. For the ACR, rewind and start the cassette. Listen to the signal from the tape (through an auxiliary earphone). When the steady tone changes to a warble, press RUN on the computer.

If the checksum loader detects a loading error, it turns on the Interrupt Enable light and stores the ASCII code of an error letter in memory location 0. The error letter is also transmitted over all terminal data channels. If a terminal is connected to one of these ports, it prints the error letter. The error letters are as follows:

C Checksum error. If the checksum on the DOS disk file does not equal the checksum generated by the loader, C error results. The error may not occur if the diskette is loaded again. If it does occur three times consecutively, the loader tape or diskette is at fault and must be replaced.

M Memory error. Data from the disk does not store properly. The location at which the error occurred is stored at locations 1 and 2 absolute.

O Overlay error. An attempt was made to load data over the loader.

I Invalid Load Device. The setting of the sense switches is incorrect.

C. When the Monitor has been loaded correctly, it responds with the first initialization question.

MEMORY SIZE?

Here the programmer may specify the amount of memory, in bytes, to be used by DOS. Typing a carriage return or zero causes DOS to use all of the read/write memory in the system. The next question is

INTERRUPTS?

Typing Y enables input interrupts and Typing N or carriage return disables them. If interrupts are enabled, special characters may be used to control program execution.

NOTE
Input interrupt features may be used only if the input interface board is strapped to accept interrupts. See Section 2-2 for information on I/O interrupts. If interrupts are not strapped, the answer to the INTERRUPTS? question must be N.

The next question is

HIGHEST DISK NUMBER?

to which the programmer responds with zero if there is one disk in the system, 1 if there are two disks and so on. The next question is

HOW MANY DISK FILES?

to which the programmer responds with the number of disk files (both sequential and random) to be open simultaneously. Responding with a carriage return sets the number of files at zero. Finally, DOS asks

HOW MANY RANDOM FILES?

Again, the programmer responds with a number or with a carriage return, which specifies zero random files.

To save time, especially when a slow terminal is in use, all of the initialization answers can be entered at once with the parameters separated by spaces. For example:

MEMORY SIZE? 0 Y 1 2 0

tells DOS that

1. it is to use all available memory,
2. input interrupts are enabled,
3. there are two disk drives in the system,
4. two sequential and
5. no random disk files are to be open at any given time.

When DOS has been properly initialized, it prints the following prompt message

DOS MONITOR VER x.x

.

The Monitor prints a period to indicate that it is now ready to receive commands.

1-3. Program Development Procedure

DOS is designed to allow the translation of an Assembly language program on paper to an operating Machine Language program with a minimum of time and effort. The process involves entering the Assembly language program into a disk file with the Text Editor, translating the file to Machine language with the Assembler and loading the program into memory with the Linking Loader.

Before the process can proceed, the disks in use must be mounted with the MNT command. To mount disk 0, the following command is used:

```
. MNT 0 <cr>
```

where <cr> means carriage return. Other disks may be mounted in the same command by typing their numbers after the zero, separated by spaces.

Mounting the disk(s) tells DOS the location of all the files and free space on each disk. If an attempt is made to run a program before the disk on which it is stored is mounted, a PROGRAM NOT FOUND error will result.

1. The first step in program development is to enter the program into a disk file with the Text Editor. The Editor is loaded from disk and run by the following command:

   ```
   .EDIT<cr>
   ```

   When it is loaded, it prints

   ```
   DOS EDITOR VER x.x
   ENTER FILE NAME
   ```

   to which the user replies with the name of the file to be entered or edited. The editor then prints

   ```
   ENTER DEVICE NUMBER
   ```

   which is answered with the number of the disk drive where the file is stored.

   Assume that an Assembly language program called SAMP is entered into a file on disk drive 0. The Editor is run with the following command:

   ```
   .EDIT SAMP 0 <cr>
   ```

   The file name (SAMP) and device number (disk 0) can be entered in the EDIT command to avoid the necessity of asking the file name and device number. The Editor searches disk drive 0 for a file name SAMP to edit. If it finds no such file, it prints the following messages:

CREATING FILE

00100

00100 is the number of the first line of the file. Now, all that is necessary is to enter the lines of the program.

```
00100   LDA             IER         LOAD MULTIPLIER<cr>
00110   LHLD            CAND        LOAD MULTIPLICAND<cr>
  .
  .
  .
```

After each carriage return, the next line number is generated automatically so that the next line can be entered. This process continues until all the lines of the program have been entered.

```
00340   PROD DB  0,0 <cr>
00350   END <cr>
00360   <cr>
```

To stop the generation of line numbers, type a null line (just a <cr>). The Editor prints an asterisk (*) to indicate it is ready to accept new commands. To check the file in order to make sure it has been entered without error, type

```
*P
```

This prints all of the lines on the current page with their line numbers. In this example, there is only one page (see paging commands, p. 40 , for an explanation of program pages), so the P command prints the whole file. The output appears as follows:

```
*P
00100           LDA     IER
00110           LHLD    CAND
00120   SHFTR   RAR
00130   SHFTR   RAR
          .
          .
          .
00240   CAND    DB      64
00250   PROD    DB      0,0
```

Suppose the line at 120 was inadvertantly entered again at line 130. To eliminate one of them, use the D (for Delete) command.

```
*D 130 <cr>
*
```

It is not necessary to type the leading zeros in the line number. To add another line between number 100 and 110, use the I (for Insert) command.

```
*I 100
00105 ;          A COMMENT LINE <cr>
00107 <cr>
```

The line number specified is that of the existing line immediately before the desired position of the new line. The Editor generates a line number halfway between the two existing lines. After typing the new line, a <cr> causes another number to be generated halfway between the inserted line and the next existing line. New lines can be inserted in this manner until there is no more room. Insertion of new lines is stopped by typing a null line.

When the file is in satisfactory form, the Editor is exited by typing the following command:

```
*E
```

This makes all of the changes, closes all of the files properly and provides a backup file. The backup file is the edited file as it appeared before the latest series of changes were made. If the edited file is unusable for some reason, the backup may be used to replace it.

2. When the program has been entered into a disk file with the Editor, it may be submitted to the Assembler for translation into machine language.

The Assembler is loaded and run with the following command:

```
.ASM <cr>
```

The Assembler prints

```
DOS ASM VER x.x
ENTER FILE NAME
```

The user enters the name of the Assembly language program file and a <cr>. The Assembler then prints

ENTER DEVICE NUMBER

to which the user replies with the number of the disk drive on which the file resides and a <cr>.

At this point, the Assembler proceeds immediately to assemble the program in the specified file. In our example, we can type

.ASM SAMP 0 <cr>

to avoid having the computer ask for the file name and drive number.

The Assembler produces a file with the machine language program and a listing. The listing is that of the source code (the input to the Assembler) along with other pertinent information. The Assembler listing of our sample program appears as follows:

SAMP LISTING

```
ØØØØØØ  Ø72  ØØØØ33'  ØØØ1ØØ         LDA    IER     LOAD MULTIPLIER
ØØØØØ3  Ø52  ØØØØ34'  ØØØ11Ø         LHLD   CAND    LOAD MULTIPLICAND
ØØØØØ6  Ø37           ØØØ12Ø  SHFTR  RAR            SHIFT 'ER RIGHT
ØØØØØ7  322  ØØØØ24'  ØØØ13Ø         JNC    SCAN    JUMP IF NO CARRY
ØØØØ12  Ø77           ØØØ135         CMC            TURN OFF CARRY
ØØØØ13  353           ØØØ14Ø         XCHG           SAVE 'CAND IN C,D
ØØØØ14  Ø52  ØØØØ36'  ØØØ15Ø         LHLD   PROD    LOAD PROD IN H,L
ØØØØ17  Ø31           ØØØ16Ø         DAD    D       ADD 'CAND TO PROD
ØØØØ2Ø  Ø42  ØØØØ36'  ØØØ17Ø         SHLD   PROD    STORE PROD
ØØØØ23  353           ØØØ18Ø         XCHG           RESTORE 'CAND
ØØØØ24  051           ØØØ19Ø  SCAN   DAD    H       SHIFT LEFT
ØØØØ25  322  ØØØØØ6'  ØØØ2ØØ         JNC    SHFTR   REPEAT IF NOT FINISHED
ØØØØ3Ø  3Ø3  ØØØØØØ   ØØØ225         JMP    ØØØ     JUMP TO MONITOR WHEN
ØØØØ33                ØØØ228  ;                      FINISHED
ØØØØ33  Ø4Ø           ØØØ23Ø  IER    DB     32
ØØØØ34  2ØØ  ØØØ      ØØØ24Ø  CAND   DB     128,Ø
ØØØØ36  ØØØ  ØØØ      ØØØ25Ø  PROD   DB     Ø,Ø
ØØØØ4Ø                ØØØ26Ø         END
```

The rightmost four columns are the source listing. Note that there is not much room for comments at the end of the line. If the comments are too long for the allotted space, the excess is printed on the next line and operation is not affected.

The next column to the left is the Text Editor's line number. The next two columns are the octal representation of the object code (the output of the Assembler). If the source instruction does not produce a machine instruction (END, for example), this column is left blank. If the source instruction defines the contents of memory (DB or DW, for example), those contents appear in the object code column. Source instructions that produce object code instructions (LDA, for example) are represented by the octal instruction code and the address of the operand. Addresses followed by an apostrophe are to be relocated. Their actual addresses are not determined until the program is loaded into memory.

Finally, the leftmost column is a list of the relative addresses of the object code instructions and memory areas. If a letter precedes the address, it indicates an error. The letter designates the nature of the error and the position indicates the address where the error occurred. A list of error letters and their meanings is in section 4-4, p. 71.

If an error is detected by the Assembler, it can be corrected by reentering the Text Editor and making the necessary changes. The ability to pass programs rapidly from the Text Editor to the Assembler and back makes DOS an extremely effective tool for writing and debugging Assembly language programs.

3. Finally, the Linking Loader is used to load the program into memory and execute the program. The Linking Loader is loaded typing the following command:

    . LINK <cr>

When the Linking Loader starts, it prints

    DOS LINK VER 1.0
    *

To load the sample program, type

    *L SAMP 0 <cr>

If the file name and drive number had been omitted, LINK would have asked for them. This command causes LINK to load our file into memory beginning at location 24000_8. Other starting addresses can be specified (see Linking Loader, L command, p.

76), but the default value is adequate for our purposes. The following command causes the program to be executed:

*X <cr>

This command causes control to be passed to whatever program begins at location 24000_8. Again, other starting addresses can be specified (see Linking Loader, X command, p. 51). If the program does not run as expected (and that is not improbable), the program bugs can be tracked down by Debug. For a description of the use of Debug, see Section 6, p. 83.

1-4. Notation and Definitions

In the specification of command formats and examples, the following notation conventions are used:

< >	Angle brackets enclose information that must be supplied by the user
[ ]	Square brackets enclose information that is optional and may be specified by the user.
<cr>	Carriage return (ASCII 013) on most terminals, <cr> is typed with the Return key.
<space>	a space (ASCII code 032)
Control/x	where x is a character, is typed by holding down the Control key while typing the character.

In examples, characters output by the computer are underlined. Information typed by the user is presented exactly as it is to be typed. All punctuation and spacing must be observed.

The following definitions are used throughout this manual:

byte	eight bits of binary information. Memory locations each contain 1 byte of information and the ASCII code uses 1 byte to represent 1 character.
file	set of information accessible to a program by name or number. Program modules, data blocks and information transferred to or from I/O devices may all be considered to be files. In this manual, files are divided into two broad classes: Sequential and Random.

A Sequential file is organized as a string of bytes of information. From any point in a sequential file, only the next byte may be accessed directly. Data bytes are written after the last existing byte of the file. Sequential files can be divided into two types, depending upon how the data bytes are interpreted:

a) ASCII files in which each byte represents a character according to the American Standard Code for Information Interchange (see Appendix A for a table of ASCII codes) and

b) binary files in which the binary data are taken as such with no code conversions applied. Two special types of binary files are distinguished from other binary files by their contents. Absolute files are those which conform to the Absolute Tape Dump format in Appendix B. The Monitor's SAV command produces absolute files. Relocatable files conform to the relocatable object code module format in Section 5-3. The Assembler produces relocatable files which the Linking Loader can then load into memory.

Random files are organized as a series of records, each of which may be accessed separately from the rest. Each record has a unique number which may be used to read, modify or write on any record in the file at any time.

The various system programs follow certain conventions for file names. See section 2-7 for an explanation of these conventions. Appendix E shows an example of the use of files in a DOS program.

program — an ordered set of machine and/or Assembler instructions that direct the computer to perform a given series of operations. The two major classes of programs are system programs and user programs.

a) system programs are stored on disk in absolute binary files and thus may be loaded and run simply by typing the program's name to the Monitor. System programs run in memory immediately above the Monitor and below user programs.

b) user programs are those programs that run in high memory above the system programs. The usual procedure for developing user programs is to construct them from one or more relocatable code modules produced by the Assembler and linked together by the Linking Loader. For a discussion of relocatable modules, see Section 5-3, page 77.

prompt — When the Monitor or a system program takes control, it prints a message indicating which program is running and whether it is ready to receive commands. The Monitor prompts with a period (.) which precedes each command. Similarly, Editor and Linking Loader commands are typed after an asterisk (*). Debug and the Assembler prompt only once after the program is loaded.

The Monitor also prompts the programmer when insufficient information has been given in a command. For example, if the programmer types

```
.MNT <cr>
```

the computer prints

```
ENTER DEVICE NUMBER
```

Typing the number and a carriage return causes the command to be executed.

1-5. DOS Input Conventions

All input to DOS (as from a terminal) is handled through the Monitor's input routine. This routine has several properties which set constraints on the form of input.

All 128 ASCII characters are accepted by the input routine except characters of the form Control/x where x is any letter. Some Control/ characters are used to control the input routine and the rest are ignored.

<cr> terminates a line. The input buffer is cleared and subsequent input is taken as a new line. <line feed> is considered an input character.

The input buffer accepts the first 72 characters as one line of input. If more than 72 characters are input in a line, the contents of the buffer are discarded and a new line is begun.

Special characters include the following:

a) Rubout deletes the last character in the buffer. When Rubout is typed, a backslash (\) and the last character in the buffer are printed. Each successive Rubout prints the previous character. Typing another character prints another backslash and the character. All of the characters between the backslashes are deleted. If Rubout is typed with no characters in the buffer, a <cr> is printed.

b) Control/U deletes the current contents of the input buffer.

c) Control/R displays the current contents of the input buffer.
   Example:

   ```
   EXAMPLE LENE\ENE\INE <Control/R>
   EXAMPLE LINE
   ```

   Typing three rubouts deleted the characters between the backslashes. Typing Control/R displayed the final appearance of the line.

d) Control/I is a tab character. When a tab is printed, spaces are printed so that the next character is printed at the start of the next 8 space column.

The following special characters are recognized if input interrupts are enabled (see p. 22).

Control/S Causes execution of a program to pause until Control/Q is typed. This can be used to pause during a listing or to pause during execution of a program to examine intermediate values.

Control/Q causes execution to resume after a Control/S. Control/Q has no effect if no Control/S has been typed.

Control/C causes execution of a program to be suspended and control to be passed to the Monitor. During the execution of certain I/O operations (Mount, Open, Kill, etc.), Control/C does not terminate execution until the operation is completed.

Control/O prevents output from the computer. Execution proceeds normally, but no output is generated until either another Control/O is typed or another command is requested by the Monitor or Editor. Example: Suppose the following Editor command is typed:

```
*P
00100   LDA   IER
00200   LHLD  CAND
<Control/O>
*
```

The Print command action is completed, but no output appears on the terminal until the Editor's prompt asterisk appears, requesting another command.

Other constraints are imposed by the system programs in use and are discussed in the descriptions of the Editor, Assembler, Debug and miscellaneous programs. Some of the standards which apply to all of the system programs are as follows:

a) All commands must be typed in upper case.

b) The fields of the command are separated by delimiters. These delimiters include space, tab, comma, semicolon and colon. Colons are used specifically to separate multiple commands on a single line.

ALTAIR DOS DOCUMENTATION
SECTION II
MONITOR

2. THE MONITOR

2-1. Introduction to the Monitor

The Monitor is the control center of the DOS system. It is used to load and execute system and user programs and to execute Input/Output routines for all of the system's peripheral devices.

The Monitor is loaded first to load and execute all the other system components. It remains in memory at all times, passing control back and forth to system and user programs and providing I/O services.

The Monitor's device-independent Input/Output system reduces programming effort. The programmer could write a different input or output routine for each I/O device used by a program. But these device handler routines are incorporated into the Monitor, so the programmer can perform the desired information transfer simply by calling the Monitor. Monitor Calls are described in detail in Appendix C.

When DOS has been loaded and initialized, the Monitor starts up and prints the following message.

```
DOS MONITOR VER x.x
.
```

This message is also printed when the Monitor is entered from another program. The period indicates that the Monitor is ready to receive commands.

2-2. Input from the Console

Input from the console keyboard is handled by a central Monitor routine regardless of the system program that is running at the time. This routine provides the following special characters and functions.

Rubout	deletes the last character in the input buffer. Typing Rubout causes a backslash (\) and the last character in the buffer to be printed. Subsequent Rubouts print the immediately previous character in the buffer. When a character other than Rubout is typed, a second backslash and the character are printed. All the characters between the backslashes are deleted.
Backarrow (←)	same as Rubout

Control/R causes the current contents of the input buffer to be printed on the console. Example:

```
EXEMPLE LINE\ENIL ELPME\AMPLE<Control/R>
EXAMPLE
```

In this example, typing Rubout 10 times deleted the characters between the backslashes; typing Control/R displays the current appearance of the line.

Control/U clears the input buffer.

<cr> terminates a line of input. The current contents of the line buffer are passed to the program and the line buffer is cleared.

If input interrupts are enabled, the following special character functions are available:

Control/C suspends execution of the current program and returns control to the Monitor.

Control/S temporarily suspends execution of a program until Control/Q is typed.

Control/Q causes execution of a program to be resumed after a Control/S

Control/O allows execution to proceed normally, but prevents output to the terminal. No output is printed until another Control/O is typed or another command is requested by the Monitor or Editor.

To enable interrupts on the older I/O interface boards (PIO, SIO A, B, C), install a jumper from the IN interrupt line to PINT or, if the Vector Interrupt board is in use, to VI7.

On newer interface boards (2SIO, 4PIO), install the jumper between PINT or VI7 to the interrupt request line for the input channel. DOS automatically assures that input interrupts are enabled.

For more information, see the manual for the interface board in use.

2-3. Monitor Commands

The Monitor is directed to perform its functions by commands. The general form of a Monitor command is as follows:

<command code> [<field> <field> . . .]

where the command code is the three letter designation of the command to be performed and the fields are the required operands for the specific command. The fields are separated by spaces, tabs or other legal delimiters. If insufficient information is given in the operand fields for a given command, the Monitor asks for the missing information and will not proceed until the information is typed. If the Monitor cannot execute the requested command, it prints an error message which indicates the reason the command could not be executed.

The following abbreviations and definitions are used in the descriptions of the Monitor commands:

delimiter	characters that separate the fields in a command. Legal delimiters are <space>, tab (Control/I), comma, semicolon and colon.
device	number of the device to be used in the command action. The Monitor at present supports only floppy disk drives in the commands, so the term "device" is interchangeable with the term "drive number."
file	name of the data or program file on which the command action is to be performed.
list	a series of device numbers or file names separated by delimiters.

Table 2-A. Monitor Commands

Command	Function
DEL <file><device>	deletes the named file from the indicated device.
DIN <device><list>	initializes the listed disk drives by writing the track and sector number in each sector. Zeros are written into each byte of each sector, destroying any existing files and marking each sector as free. The DOS disk is initialized at the factory and must not be initialized again. Doing so will destroy all system programs as well as user files.

Command	Function
DIR <device>	Prints a directory of the files on the indicated device. See section 2-7 for an explanation of the file name conventions.
DSM <device list>	Dismounts the disks on the listed device or devices. A disk must be dismounted before it is removed from a drive. Failure to do so may cause file link errors the next time the disk is read.
LOA <file><device>	Loads the named file into memory from the specified device. The file must be an absolute binary file. The LOA command automatically adds # to the file name.
MNT <device list>	Mounts the disks on the specified devices. The MNT command causes the system to read each specified diskette and creates a table of unused space. When files are created or modified, the system checks the table for unused sectors. This command must be given before the files on a disk may be accessed.
REN <old name> <new name> <device>	Renames the file <old name> on the specified device to have a name <new name>.
RUN <file><device>	Loads the named file from the specified device and runs it. The file must be an absolute binary file. A # sign is automatically added to the file name.
SAV <file><device> <1st location> <last location><sa>	Contents of memory from the first location to the last location are saved as an absolute binary file With the specified name. A # sign is automatically added to the file name. Any subsequent RUN command causes execution to begin at <sa>.

If the input to the Monitor is not one of these commands, the Monitor searches disk drive 0 for an absolute program file which has a name corresponding to the input. If such a file is found, it is loaded and run. The following system programs are run in this manner:

ASM	Assembler - see chapter 4
EDIT	Text Editor - see chapter 3
DEBUG	Debug package - see chapter 6
LINK	Linking Loader - see chapter 5
INIT	Disk initialization program - see chapter 7
CNS	Console - see chapter 7. Console allows the Monitor command console to be changed to another terminal.

Drive 0 must be mounted before running these programs.

2-4. Monitor Error Messages

When the Monitor detects an error in the execution of a command or a Monitor Call, it prints an error message and terminates execution of the operation. In the case of an error in a Monitor Call, the error message is printed and control returns to the calling program.

A Monitor error message contains the following information:

Error Code	the error codes are given in Table 2-B
File Number	the number of the file that was being accessed when the error occurred
RQCB Address	the address of the Request Control Block of the Monitor Call that caused the error.
Opcode	the operation code of the Monitor Call that caused the error
Return Address	the address to which control would have returned had the error not occurred.

Table 2-B. Error Codes

Error Code	Meaning
1	FILE TABLE ENTRY MISSING

The file table contains entries for thirteen disk files (numbered 0 - 12) and four other I/O files (0 - 3). If a file number other than these is encountered, an error occurs.

2 DEVICE NOT IN PHYSICAL DEVICE TABLE

The following devices are listed in the physical device table:

Teletype or Teletype compatible terminal
Audio Cassette
High-Speed Paper Tape Reader
Floppy Disk

An attempt to transfer information to or from another device causes an error.

3 HANDLER NOT IN HANDLER TABLE

An attempt was made to perform an invalid operation on an I/O device, for example, to output to a paper tape reader.

4 BOARD NOT IN I/O TABLE

The following I/O boards are in the I/O table:

2SIO
SIO A, B, and C
4PIO
PIO

Use of other boards is not supported.

5 SHORT DATA TRANSFER

The end of data transfer came before the specified number of bytes was read or written.

6 CHECKSUM ERROR

When a program is loaded, the Monitor keeps a running sum of all the bytes in each record. The least significant byte of this sum is the checksum. At the end of the record, it is compared with the checksum byte in the record. If there is a discrepancy between them, an error has occurred in loading the program and the Checksum Error message is printed.

7 MEMORY ERROR

An attempt was made to write into a bad memory location. This could be a non-functioning read/write memory location or a location in read-only memory.

10 BAD FILE NUMBER

A bad file number is one which has not been opened or which is greater than the number of files allocated at initialization.

11 FILE LINK ERROR

During a disk file read, a sector was read which did not belong to the file. A FILE LINK ERROR often occurs after a disk has been removed from a drive without being dismounted first.

12 I/O ERROR

A checksum error occurred in 18 successive disk read operations. A checksum error on a disk read causes the disk controller automatically to re-read the sector. A Disk I/O Error indicates that

the error is a permanent defect in the file, disk or disk drive.

13 BAD FILE MODE

A sequential operation was attempted on a random file or vice versa.

14 DEVICE NOT OPEN

An attempt was made to input or output a file through a device which had not been opened to that file.

15 DEVICE NOT ENABLED

The door of a disk drive has not been closed, or the motor of the drive has not had time to come up to full speed.

16 DEVICE ALREADY OPEN

An attempt was made to mount a disk which has already been mounted.

17 INTERNAL ERROR

DOS became confused. Please report the circumstances of this error to the MITS, Inc. Software Department.

20 OUT OF RANDOM BLOCKS

All sectors allotted for random files have been filled.

21 FILE ALREADY OPEN

An open operation was attempted on a file that was already open.

22 FILE NOT FOUND

The file name referred to was not found on the specified device.

23 TOO MANY FILES

An attempt was made to create a file when the disk directory was already full.

24 MODE MISMATCH

A command that expected a character string operand received a number, or vice-versa. This error often occurs when the quotation marks are left out of a character string in a command.

25 END OF FILE

During a read operation, an end of file mark was encountered before the read operation was complete.

26 DISK FULL

All of the sectors of the disk have been used.

27 BAD RECORD NUMBER

An attempt was made to refer to a random file record that was not in the specified file.

30 FILE TABLE FULL

An attempt was made to have more than thirteen disk files or four I/O files open at one time.

31 Unused

32 TOO MANY OPEN DISK FILES

An attempt was made to open more disk files than were specified at initialization.

33 FILE ALREADY EXISTS

An attempt was made to name or rename a file with a name that already exists in the directory.

2-5. File Name Conventions

When a directory of disk files is listed by the DIR command, the file names are preceded by special characters that denote the file type. These characters and their meanings are as follows:

#	absolute binary files. Files with this character are produced by the Monitor's SAV command and are used as input by the LOA and RUN commands. System program names appear in the directory with a pound sign (#).
*	relocatable load module. These files are output by the Assembler and used as input by the Linking Loader.
%	listing file. The optional source listing from ASM carries this designation.
&	Editor source file. The output of the Editor carries this designation.
$	Editor backup file. When a file is modified by the Editor, the old, unmodified file is renamed to have this designation.

These characters are supplied automatically by the system programs and Monitor commands which create the files. Therefore, they need not be supplied by the programmer. For example, the command

```
.ASM MULTI 0
```

is used to assemble the file which appears in the directory as

```
&MULTI
```

Similarly, the command

```
.EDIT TEXT 0
```

creates a source file called &TEXT.

File names in the DEL and REN commands must appear exactly as they do in the directory. For example, the Editor backup file

```
$LETTER
```

may be deleted by

```
.DEL $LETTER
```

without affecting the source file &LETTER or any other file.

ALTAIR DOS DOCUMENTATION
SECTION III
TEXT EDITOR

3. THE TEXT EDITOR

3-1. Introduction

Although the Text Editor is primarily used to create and maintain Assembly Language program files, it can be used for any ASCII coded file. EDIT is a line-oriented Editor, in that its commands operate on lines of text which are addressable by number. Line numbers are assigned automatically as the file is being created. A special command allows automatic renumbering of lines. The Assembler ignores EDIT line numbers in its input file except when producing a source listing.

Once the system disk (on drive 0) has been mounted with the MNT command, EDIT may be loaded and run with the following command:

.EDIT <file><device>

where <file> is the name of the file to be created or modified, and <device> is the number of the disk where the file is stored. When EDIT prints an asterisk (*), it is ready to accept commands. EDIT requires at least 2 disk files to be allocated at initialization.

The Text Editor is designed to minimize memory usage by dividing files into pages. Only one page resides in memory at a time, while the rest of the file remains on disk. The number, length and content of pages are completely under the programmer's control. Access to the pages is sequential; the paging commands refer to the next page in the file. The B command always refers to the first page of the file, so the Editor can go back to the beginning of a multipage file from any point.

Edit commands are provided to add, delete and replace lines, find and substitute character strings and modify individual lines. The form of an EDIT command is as follows:

<x> <field>[<field>] <cr>

where x stands for the EDIT command letter in use, and field is a line number or character string, depending upon the command. The command letter and fields are separated by delimiters.

The EDIT commands operate on individual lines or on ranges of lines. A line is referenced by stating its number in an EDIT command. For example,

P 150

prints line 150 on the console. A range of lines is referenced by stating the beginning and ending lines of the range. Thus,

R 200 230

replaces lines 200 to 230, inclusive. All line and range references are to lines on the current page only. Before a line or range on another page may be referenced, that page must be loaded into memory.

3-2. Edit Commands

A. Inserting, Deleting and Replacing lines. The following commands insert, delete and replace whole lines:

I <number><increment><cr> Inserts a new line at <number> or the first available line after <number>. After the <cr>, EDIT prints <number> or, if there is already a line at <number>, the number of the first available line after <number>. All input up to the next <cr> is inserted as the new line. In the Insert mode, the Editor automatically assigns numbers to the lines as they are entered. If <increment> is not specified, the line number increment is that last used in an N command. If there has been no previous N command, the default increment is 10. After a line is typed and a carriage return entered, EDIT adds the increment and checks to see that the new line number is less than the next existing line number. If it is not, the increment is reduced to half the difference between

the previous line number and the next existing line number. This process is repeated until no new line numbers are possible. Then the Insert mode is exited and an asterisk is printed. When a file is being created by the Editor, there are no existing lines, so each line is numbered with the specified or default increment.

Example:

```
.EDIT TEST 0
DOS EDITOR VER 0.1
CREATING TEST
00100 THIS IS A TEST <cr>
00110 FILE SHOWING LINE <cr>
00120 NUMBER INCREMENTS <cr>
00130 <cr>
*
```

In this example, new line numbers were generated after every carriage return until a null line (a line with no characters before the carriage return) was typed. Then Insert mode was terminated and the prompt asterisk printed. In the following example, insertions are made into file TEST:

```
*I 110
00115 INSERT ONE <cr>
00117 INSERT TWO <cr>
00118 INSERT THREE <cr>
00119 INSERT FOUR <cr>
*
```

	In each case, the increment was halved, until it was not possible to insert another line.
D <1st number> [<2nd number>] <cr>	Deletes all lines from <1st number> to <2nd number>, inclusive. If <2nd number> is omitted, one line is deleted.
R <1st number> <2nd number> <cr>	Replaces the lines from <1st number> to <2nd number>, inclusive, with input from the console. After the <cr>, EDIT displays the number of the first line to be replaced. All input to the next <cr>, replaces the line. After the next <cr>, the number of the next line to be replaced is displayed. Typing a null line causes that line and the remaining lines in the range to be deleted. If <2nd number> is omitted, one line is replaced.

B. Finding a String. The following commands display the next occurrence of a character string:

F <string> <cr>	Finds the next occurrence of <string> on the current page. If <string> is found, the line in which it appears is printed. If it is not found, an asterisk is printed and EDIT is ready for further commands. The search begins on the line immediately after the current line.
S <string> <cr>	The same as F, except the search can extend over page boundaries.

C. In-Line Editing: the Alter Command. The Alter command allows adding, deleting or modifying characters within a line without affecting the other lines in the file. The format of the Alter command is as follows:

A <number> <cr>

where <number> is the number of the line to be altered. The Alter command allows the use of several subcommands which order changes to be made. The subcommand action begins with the next character to the right of the current position. Changes are made from left to right.

In the listing of subcommands below, 'n' preceding the subcommand letter means the subcommand may be preceded by a number which indicates the number of times the subcommand is to be repeated. For example:

3CABC

is equivalent to three subcommands

CA

CB

CC

in sequence.

The Alter subcommands are not echoed. When they are used, the only output from the computer is a display of the line as modified.

In the examples that follow, assume the following command has been executed:

A 100

where line 100 is in file TEST on page 35. The Alter subcommands are as follows:

Command	Explanation
n<space>	skips over and prints the next n characters in the line. Typing <space> displays 00100 T
nC<characters>	changes the next n characters in the line to the specified characters. Typing 3CHAT displays 00100 THAT
nD	deletes the next n characters. Typing D displays 00100 THAT and deletes the following space. The effect of the subcommand is not apparent until the next subcommand is executed.
H<string>	deletes the rest of the line and inserts the string in its place. The string is terminated either by <Escape> or by <cr>. (On some terminals, Altmode is used rather than Escape.) Terminating with <Escape> allows the Alter command to receive further subcommands. <cr> exits Alter mode. Typing H'S NO<Escape> displays 0100 THAT'S NO
I<string>	inserts the string before the next character. The string is terminated either by <Escape> (Altmode on some terminals) or by <cr>. Typing <Escape> allows further subcommands to be issued. Typing <cr> exits Alter mode. Typing ILINE <cr> displays

```
                                0100 THAT'S NO LINE
```

and exits Alter mode.

To demonstrate the remaining Alter subcommands, the command

```
*A 100 <cr>
```

is executed again. This command reenters Alter mode on the same line as before and moves the current position to the beginning of the line.

nK<character>	deletes everything up to (but not including) the nth occurrence of the character. If the character does not exist, or if there are fewer than n of them, the subcommand does nothing. Typing KO displays `0100` The effect of the subcommand is not apparent until the next subcommand is executed.
R<string>	replaces the next character with the string. The string is terminated by <Escape> or <cr>. Typing <cr> exits Alter mode. Typing RSOME <space> <Escape> displays `0100 SOME`
nS<character>	skips over and prints all characters up to, but not including, the nth occurrence of <character>. If no such character exists, or if there are fewer than n of them, the subcommand does nothing. Typing SN displays `0100 SOME LI`
X<string>	skips to the end of the line and inserts the string at that point. The string is terminated with <Escape> or <cr>. <Escape> allows further

subcommands to be issued. <cr> exits Alter mode. Typing X, THAT! <cr> displays

0100 SOME LINE, THAT!

When all of the desired changes have been ordered, Alter command mode is exited with one of the following subcommands:

<cr>	replaces the existing line with the line as modified and exits Alter mode.
Q	exits Alter mode, but makes none of the ordered changes. The changes are lost.

D. Paging commands. The amount of memory used by the Text Editor may be minimized by dividing the file to be edited into pages and loading one page into memory at a time. Pages are manipulated by the following commands:

B	Loads the first page of the file into memory. Note that after a B command is issued, the line number is unpredictable. An additional command (such as P <number>) is needed to refer to any specific line on the page.
C	Loads the next page of the file into memory and saves the current page on disk.
L	Loads the next page into memory and deletes the current page
W <number>	Writes the lines currently in memory from the first to <number> onto disk as a page.

E. Miscellaneous commands:

N <increment>	Renumbers all of the lines in the file. The difference between successive line numbers is <increment>.

	The first line number is always 100.
P [<first number> [<second number>]]	Prints all lines from the <1st number> to the <2nd number>, inclusive. If there is no second number, 1 line is printed. If no line numbers are given, the entire current page is printed.
E <file name> <device number>	As the Editor proceeds through the named file making changes, it copies the modified file into a temporary file called EDIT.TEM. When the E command is executed, the remaining unmodified lines of the file are copied into EDIT.TEM. This file is then assigned the name of the edited file. The first character of the original file name is changed to $. This provides a backup file. Any previous backup file is deleted. If a file name and device number are specified in the E command, EDIT proceeds to edit that file. Thus, another file may be edited without having to reload the Editor. If the file and device are not specified, control is passed to the Monitor.
Q <file name> <device number>	Q exits to the monitor without renaming any files. The changes made by the Editor are ignored. The Q command allows the user to abort an editing session without damaging any files. The file name and device number may be specified as in the E command to edit another file without having to reload the Editor.

ALTAIR DOS DOCUMENTATION
SECTION IV
ASSEMBLER

4. THE ASSEMBLER

The Assembler is a system program that translates programs from Assembly Language into machine language. In principle, machine language can be used to write programs for the computer. A machine language program is one in which the instructions to the computer are represented by binary numbers one, two or three bytes long. The practical problems of machine language programming, however, make its use virtually impossible for all but the simplest programs. First, it is difficult to remember all of the binary machine language codes and enter them into the computer without error. Second, machine language requires the programmer to remember all of the addresses in the program and refer to them explicitly. Finally, if a machine language program does not work as desired, it is extremely difficult to determine what went wrong.

Assembly language programming is preferable to machine language programming because it avoids all of these difficulties. Machine instructions are referred to in Assembly language by mnemonics that are descriptive of the operation and that are relatively easy to remember. Addresses can be specified explicitly, but they can also be referred to symbolically. That is, a memory location can be given a label and referred to subsequently simply by mentioning that label. Finally, Assembly language provides the programmer with a complement of error messages that make the process of debugging much easier than in machine language programming.

The DOS Assembler translates Assembly Language to machine language by means of a two step process. In the first step, the Assembler reads the Assembly Language program and assigns addresses to all of the symbols. In the second step, the program is read again and the instructions are converted to their machine language equivalents. On this second pass through the program, the program may be listed on the terminal or in a disk file. If the Assembler detects an error in the program, the place where the error occurred is marked in the listing with a letter that indicates the nature of the error.

Once the system disk is mounted in drive 0, the Assembler is run by typing the following command to the Monitor:

. ASM <file name> <device> [<device type> <device number>]

where the <file name> is the name of the disk file that contains the

source program and <device> is the number of the drive where that file resides. If a <device type> is specified, an Assembler listing is written in a file on the specified device. If the <device type> is TTY, the listing is printed on the terminal; if the <device type> is FDS, it is sent to floppy disk. The name of the listing disk file is the file name in the ASM command preceded by a percent sign (%). The following message is printed on the terminal upon termination of the assembly:

xxxxx ERRORS DETECTED

where xxxxx is the number (in octal) of errors encountered in the program.

The machine language, object code module that results from the Assembler's action is written on the same disk as the source code. The name of the object code file is the <file name> preceded by an asterisk (*). For example, after the following command is executed:

.ASM SOURCE 0 FDS 1

the object code file is named *SOURCE and is written on disk 0. The listing of the source program is named %SOURCE and resides on disk 1.

When the assembly and listing are complete, the Assembler prints

ANY MORE ASSEMBLIES?

Typing "Y" causes the Assembler to start over and ask for the new file name, device number and listing file parameters. Thus, another file may be assembled without reloading the assembler. Typing N or <cr> exits the Assembler and returns control to the Monitor.

4-1. Statements

The fundamental unit of an Assembly Language program is the statement, whose form is as follows:

[label] <op-code> <operand> [,<operand>] [comment]

The label is a tag by which other statements in the program can refer to this statement. Not all statements in a program need to be labelled. Since program execution proceeds normally in order from the lowest memory location to the highest, statements that need to be executed in normal sequence need not carry labels. If, on the other hand, a statement needs to be executed out of normal order, it must carry a label. Such out-of-order execution is called branching and it is particularly important in programmed decision making and loops. Labels can also be used to refer

to memory locations for storing data. This use will be discussed more fully in section 4-2B below.

The op-code is the mnemonic of the machine instruction or Assembler pseudo-operation to be performed by the statement. Machine instruction op-codes are translated by the Assembler into machine language instructions. Assembler pseudo-ops are not translated, but direct the Assembler itself to allocate storage areas, set up special addresses, etc.

The op-code is followed by one or more operands, depending upon the nature of the instruction. An operand is an address - specified in any one of several manners - where the computer is to find the data to be operated upon. In the case of an ADC (add with carry) instruction, for example, the operand is the address of the location whose contents are to be added to the accumulator. In the MOV (above) instruction, the two operands are the addresses of the location from which a data byte is to be taken and to which it is to be moved.

Comment may be added to the end of a statement if they are separated from the rest of the statement by a semicolon. Comments are ignored by the Assembler, but they do appear in the Assembler listing and may thus be used by the programmer for documentation and explanation.

4-2. Addresses

A program is a series of statements that are stored in memory and executed either in the order in which they are stored or in sequence directed by statements in the program itself. The data operated upon by the program or used to direct the program's actions is stored in memory and referred to by the addresses of the locations in which it is stored. Therefore, addresses are used both to control execution of the program and to manipulate data. Much of the versatility of the Assembly Language programming system in DOS results from the various ways in which addresses may be represented and modified.

The DOS Assembler recognizes addresses in three major forms; constants, labels and address expressions.

A. Constants. A constant is an address that is stated explicitly as a number. For example, the instruction

```
JMP 23000
```

causes execution to proceed from the location whose address is 23000 decimal. A constant address may be expressed in octal, decimal or hexadecimal notation.

1. Octal address constants are strings of octal characters (0 - 7) whose first character is zero. The allowable range of values is -01777777 to 01777777.
   Examples:
   ```
   0377
   01345
   017740
   ```
2. Decimal address constants are strings of decimal digits (0 - 9) without a leading zero. The allowed range is -65536 to 65536. Examples:
   ```
   255
   1024
   23000
   ```
3. Hexadecimal address constants have the following form:
   ```
   X'hhhh'
   ```
   where h is any hexadecimal digit (0 - 9, A - F). The allowed range is -X'FFFF' to X'FFFF'. Examples:
   ```
   X'F000'
   X'2300'
   X'00F'
   ```
4. Character address constants have the following form:
   ```
   "xx"
   ```
   where x is any ASCII character except ("). The characters are translated into binary according to their ASCII codes and the resulting two-byte quantity makes up the address. Examp es:
   ```
   "A1"
   "BZ"
   "#"
   ```

B. Labels. When a statement is labelled, the label is entered into the symbol table in the Assembler along with the address of the statement. Any subsequent statement can then use the label to represent that address. Two types of labels can be used in the DOS Assembler; names and program points.

1. Names are strings of up to 6 alphanumeric characters. The first character must be a letter and the subsequent characters may be letters, numbers or dollar signs.
Examples:

```
SHIFT
LBL1
A$OUT
```

The usual use of labels is to refer to a statement by name. For example:

```
.
.
.
SHIFT      RAR
           JNC      SHIFT
.
.
.
```

The operand of the jump instruction tells the computer to branch back to the RAR (rotate right) instruction if there is no carry out of the shift. If there is a carry, execution proceeds with the next instruction after the jump.

Data bytes can bear labels as well. For example:

```
           ADC      ADDEND
ADDEND     DB       255
```

These instructions add the contents of location ADDEND to the accumulator with carry. In this example, the contents of ADDEND have the value 255 decimal.

For the purposes of clarity and ease of use, names should be systematically applied. That is, they should be logically related to the statements or data locations they represent and should be easily distinguishable from other names in the program.

Sometimes, short branches and loops require statements to be labelled, but those labels are not important to the whole program. Rather than filling up the symbol table with unique

names, the programmer may prefer to label those statements with program points.

2. Program points are special labels with the following form:

   .x

   where x is any letter. A letter may be used any number of times in a single program. Unlike names, program points may be referred to in two ways. The program point reference -x refers to the most recently encountered program point with letter x. The program point reference +x refers to the next program point in the program with the letter x. Therefore, while any number of statements may be labelled with the same program point, a statement may only refer to the two program points bracketting it in the program.

C. Address Expressions. The DOS Assembler allows addresses to be specified relative to other addresses. For example, to refer to the fourth location after the location labelled LOC, the following expression can be used:

   LOC+4

Expressions of this form are called address expressions. Address expressions may be comprised of any of the following:

   Name
   Constant
   Program point reference
   Address expression ± constant

The sixteen bit values of the names, constants, program point references and address expressions are combined and truncated to 16 bits to form the value of the final address expression.

Example:

```
SHIFT+5
+A-010
LOC+X'F'
```

D. Special Addresses. The DOS Assembler allows certain addresses to be referred to directly with special notation.

* indicates the present contents of the location counter. That is, * refers to the address of the current instruction or the current data address.

Registers may be addressed symbolically by name. Therefore, such instructions as

```
MOV     H,A
```

are interpreted to refer to the correct registers.

E. Addressing Modes. The addresses of statements or data locations are specified in one of five different modes. The DOS Assembler addressing modes are Absolute, Relative, Common, Data and External.

Absolute addresses are the actual hardware addresses of the designated locations. Address constants in themselves (not in address expressions) refer to absolute mode addresses. If an absolute mode address is specified, all of the other addresses in the program must be relocated to fit it.

Relative addresses are relocated by the action of the Linking Loader. Unless otherwise specified, all symbolic addresses (names, program points, address expressions) are in Relative mode. To calculate a Relative mode address, the Assembler calculates a displacement which the Linking Loader adds to a relocation base address when the program is loaded. In this way, the loader can load the program anywhere in memory and all the addresses bear the correct relation to each other.

An External mode address is one that refers to a location in another program. A name must be mentioned in an EXT statement before it can be used as an External mode address. External addresses allow a program to use routines or data in another program.

Data and Common mode addresses refer to separate blocks of memory locations that may or may not be contiguous with the programs which make the references. Data mode addresses are so designated by being mentioned in a DAT statement. Common mode items are designated by CMN statements. The difference between Common and Data addresses is that Data addresses may only be referenced by the program in which they are defined, whereas Common mode addresses are available to any program. In addition, several Common blocks can exist simultaneously and be referred to by name.

In an address expression, the constituent addresses may have different modes. Any mode expression combined with an Absolute mode address has the mode of the expression. The difference of two expressions of the same mode is of Absolute mode.

4-3. Op-Codes

Op-codes are of two types. One type, the machine codes, are the mnemonic expressions of the 8080 instructions. These op-codes and their associated operands are discussed in section A, below, which is reprinted from the Intel 8080 Microcomputer System Users' Manual. The Assembler can use any address expression to derive the required address for direct or immediate addressing instructions. Register instructions can use any address expression as long as its value is the address of a register (0 - 7 absolute). Before a register indirect mode instruction may be used, the register pair must be loaded with an address. Any address expression can be used to supply that address.

CHAPTER 4
INSTRUCTION SET

A computer, no matter how sophisticated, can only do what it is "told" to do. One "tells" the computer what to do via a series of coded instructions referred to as a **Program**. The realm of the programmer is referred to as **Software**, in contrast to the **Hardware** that comprises the actual computer equipment. A computer's software refers to all of the programs that have been written for that computer.

When a computer is designed, the engineers provide the Central Processing Unit (CPU) with the ability to perform a particular set of operations. The CPU is designed such that a specific operation is performed when the CPU control logic decodes a particular instruction. Consequently, the operations that can be performed by a CPU define the computer's **Instruction Set**.

Each computer instruction allows the programmer to initiate the performance of a specific operation. All computers implement certain arithmetic operations in their instruction set, such as an instruction to add the contents of two registers. Often logical operations (e.g., OR the contents of two registers) and register operate instructions (e.g., increment a register) are included in the instruction set. A computer's instruction set will also have instructions that move data between registers, between a register and memory, and between a register and an I/O device. Most instruction sets also provide **Conditional Instructions.** A conditional instruction specifies an operation to be performed only if certain conditions have been met; for example, jump to a particular instruction if the result of the last operation was zero. Conditional instructions provide a program with a decision-making capability.

By logically organizing a sequence of instructions into a coherent program, the programmer can "tell" the computer to perform a very specific and useful function.

The computer, however, can only execute programs whose instructions are in a binary coded form (i.e., a series of 1's and 0's), that is called **Machine Code**. Because it would be extremely cumbersome to program in machine code, programming languages have been developed. There are programs available which convert the programming language instructions into machine code that can be interpreted by the processor.

One type of programming language is **Assembly Language.** A unique assembly language mnemonic is assigned to each of the computer's instructions. The programmer can write a program (called the **Source Program**) using these mnemonics and certain operands; the source program is then converted into machine instructions (called the **Object Code**). Each assembly language instruction is converted into one machine code instruction (1 or more bytes) by an **Assembler** program. Assembly languages are usually machine dependent (i.e., they are usually able to run on only one type of computer).

THE 8080 INSTRUCTION SET

The 8080 instruction set includes five different types of instructions:

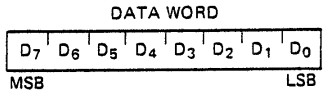
- **Data Transfer Group**—move data between registers or between memory and registers
- **Arithmetic Group** – add, subtract, increment or decrement data in registers or in memory
- **Logical Group** – AND, OR, EXCLUSIVE-OR, compare, rotate or complement data in registers or in memory
- **Branch Group** – conditional and unconditional jump instructions, subroutine call instructions and return instructions
- **Stack, I/O and Machine Control Group** – includes I/O instructions, as well as instructions for maintaining the stack and internal control flags.

Instruction and Data Formats:

Memory for the 8080 is organized into 8-bit quantities, called Bytes. Each byte has a unique 16-bit binary address corresponding to its sequential position in memory.

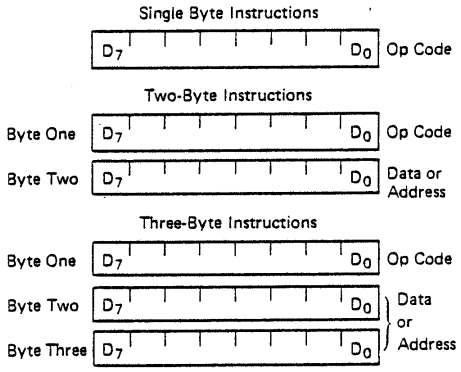
The 8080 can directly address up to 65,536 bytes of memory, which may consist of both read-only memory (ROM) elements and random-access memory (RAM) elements (read/write memory).

Data in the 8080 is stored in the form of 8-bit binary integers:



When a register or data word contains a binary number, it is necessary to establish the order in which the bits of the number are written. In the Intel 8080, BIT 0 is referred to as the **Least Significant Bit (LSB)**, and BIT 7 (of an 8 bit number) is referred to as the **Most Significant Bit (MSB)**.

The 8080 program instructions may be one, two or three bytes in length. Multiple byte instructions must be stored in successive memory locations; the address of the first byte is always used as the address of the instructions. The exact instruction format will depend on the particular operation to be executed.



Addressing Modes:

Often the data that is to be operated on is stored in memory. When multi-byte numeric data is used, the data, like instructions, is stored in successive memory locations, with the least significant byte first, followed by increasingly significant bytes. The 8080 has four different modes for addressing data stored in memory or in registers:

- Direct – Bytes 2 and 3 of the instruction contain the exact memory address of the data item (the low-order bits of the address are in byte 2, the high-order bits in byte 3).
- Register – The instruction specifies the register or register-pair in which the data is located.
- Register Indirect – The instruction specifies a register-pair which contains the memory address where the data is located (the high-order bits of the address are in the first register of the pair, the low-order bits in the second).
- Immediate – The instruction contains the data itself. This is either an 8-bit quantity or a 16-bit quantity (least significant byte first, most significant byte second).

Unless directed by an interrupt or branch instruction, the execution of instructions proceeds through consecutively increasing memory locations. A branch instruction can specify the address of the next instruction to be executed in one of two ways:

- Direct – The branch instruction contains the address of the next instruction to be executed. (Except for the 'RST' instruction, byte 2 contains the low-order address and byte 3 the high-order address.)
- Register indirect – The branch instruction indicates a register-pair which contains the address of the next instruction to be executed. (The high-order bits of the address are in the first register of the pair, the low-order bits in the second.)

The RST instruction is a special one-byte call instruction (usually used during interrupt sequences). RST includes a three-bit field; program control is transferred to the instruction whose address is eight times the contents of this three-bit field.

Condition Flags:

There are five condition flags associated with the execution of instructions on the 8080. They are Zero, Sign, Parity, Carry, and Auxiliary Carry, and are each represented by a 1-bit register in the CPU. A flag is "set" by forcing the bit to 1; "reset" by forcing the bit to 0.

Unless indicated otherwise, when an instruction affects a flag, it affects it in the following manner:

Zero: If the result of an instruction has the value 0, this flag is set; otherwise it is reset.

Sign: If the most significant bit of the result of the operation has the value 1, this flag is set; otherwise it is reset.

Parity: If the modulo 2 sum of the bits of the result of the operation is 0, (i.e., if the result has even parity), this flag is set; otherwise it is reset (i.e., if the result has odd parity).

Carry: If the instruction resulted in a carry (from addition), or a borrow (from subtraction or a comparison) out of the high-order bit, this flag is set; otherwise it is reset.

Auxiliary Carry: If the instruction caused a carry out of bit 3 and into bit 4 of the resulting value, the auxiliary carry is set; otherwise it is reset. This flag is affected by single precision additions, subtractions, increments, decrements, comparisons, and logical operations, but is principally used with additions and increments preceding a DAA (Decimal Adjust Accumulator) instruction.

Symbols and Abbreviations:

The following symbols and abbreviations are used in the subsequent description of the 8080 instructions:

SYMBOLS	MEANING
accumulator	Register A
addr	16-bit address quantity
data	8-bit data quantity
data 16	16-bit data quantity
byte 2	The second byte of the instruction
byte 3	The third byte of the instruction
port	8-bit address of an I/O device
r,r1,r2	One of the registers A,B,C,D,E,H,L
DDD,SSS	The bit pattern designating one of the registers A,B,C,D,E,H,L (DDD=destination, SSS=source):

DDD or SSS	REGISTER NAME
111	A
000	B
001	C
010	D
011	E
100	H
101	L

rp	One of the register pairs: B represents the B,C pair with B as the high-order register and C as the low-order register; D represents the D,E pair with D as the high-order register and E as the low-order register; H represents the H,L pair with H as the high-order register and L as the low-order register; SP represents the 16-bit stack pointer register.
RP	The bit pattern designating one of the register pairs B,D,H,SP:

RP	REGISTER PAIR
00	B-C
01	D-E
10	H-L
11	SP

rh	The first (high-order) register of a designated register pair.
rl	The second (low-order) register of a designated register pair.
PC	16-bit program counter register (PCH and PCL are used to refer to the high-order and low-order 8 bits respectively).
SP	16-bit stack pointer register (SPH and SPL are used to refer to the high-order and low-order 8 bits respectively).
r_m	Bit m of the register r (bits are number 7 through 0 from left to right).
Z,S,P,CY,AC	The condition flags: Zero, Sign, Parity, Carry, and Auxiliary Carry, respectively.
( )	The contents of the memory location or registers enclosed in the parentheses.
←	"Is transferred to"
∧	Logical AND
∀	Exclusive OR
∨	Inclusive OR
+	Addition
−	Two's complement subtraction
*	Multiplication
⟷	"Is exchanged with"
—	The one's complement (e.g., $(\overline{A})$)
n	The restart number 0 through 7
NNN	The binary representation 000 through 111 for restart number 0 through 7 respectively.

Description Format:

The following pages provide a detailed description of the instruction set of the 8080. Each instruction is described in the following manner:

1. The MAC 80 assembler format, consisting of the instruction mnemonic and operand fields, is printed in **BOLDFACE** on the left side of the first line.
2. The name of the instruction is enclosed in parenthesis on the right side of the first line.
3. The next line(s) contain a symbolic description of the operation of the instruction.
4. This is followed by a narative description of the operation of the instruction.
5. The following line(s) contain the binary fields and patterns that comprise the machine instruction.

6. The last four lines contain incidental information about the execution of the instruction. The number of machine cycles and states required to execute the instruction are listed first. If the instruction has two possible execution times, as in a Conditional Jump, both times will be listed, separated by a slash. Next, any significant data addressing modes (see Page 4-2) are listed. The last line lists any of the five Flags that are affected by the execution of the instruction.

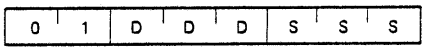
Data Transfer Group:

This group of instructions transfers data to and from registers and memory. **Condition flags are not affected** by any instruction in this group.

MOV r1, r2 (Move Register)

(r1) ← (r2)

The content of register r2 is moved to register r1.

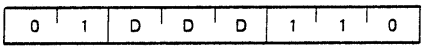


Cycles: 1
States: 5
Addressing: register
Flags: none

MOV r, M (Move from memory)

(r) ← ((H) (L))

The content of the memory location, whose address is in registers H and L, is moved to register r.

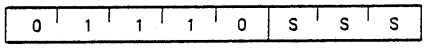


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

MOV M, r (Move to memory)

((H) (L)) ← (r)

The content of register r is moved to the memory location whose address is in registers H and L.

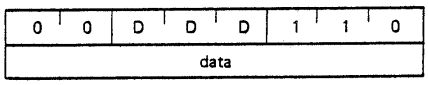


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

MVI r, data (Move Immediate)

(r) ← (byte 2)

The content of byte 2 of the instruction is moved to register r.

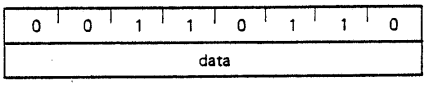


Cycles: 2
States: 7
Addressing: immediate
Flags: none

MVI M, data (Move to memory immediate)

((H) (L)) ← (byte 2)

The content of byte 2 of the instruction is moved to the memory location whose address is in registers H and L.



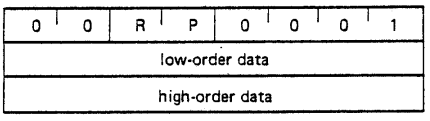
Cycles: 3
States: 10
Addressing: immed./reg. indirect
Flags: none

LXI rp, data 16 (Load register pair immediate)

(rh) ← (byte 3),

(rl) ← (byte 2)

Byte 3 of the instruction is moved into the high-order register (rh) of the register pair rp. Byte 2 of the instruction is moved into the low-order register (rl) of the register pair rp.

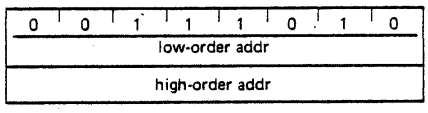


Cycles: 3
States: 10
Addressing: immediate
Flags: none

LDA addr (Load Accumulator direct)

(A) ← ((byte 3)(byte 2))

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register A.

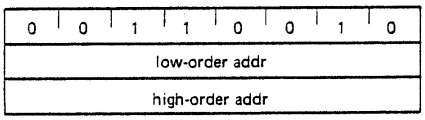


Cycles: 4
States: 13
Addressing: direct
Flags: none

STA addr (Store Accumulator direct)

((byte 3)(byte 2)) ← (A)

The content of the accumulator is moved to the memory location whose address is specified in byte 2 and byte 3 of the instruction.



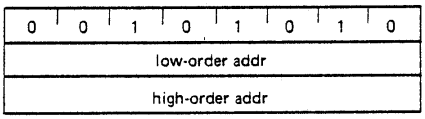
Cycles: 4
States: 13
Addressing: direct
Flags: none

LHLD addr (Load H and L direct)

(L) ← ((byte 3)(byte 2))

(H) ← ((byte 3)(byte 2) + 1)

The content of the memory location, whose address is specified in byte 2 and byte 3 of the instruction, is moved to register L. The content of the memory location at the succeeding address is moved to register H.



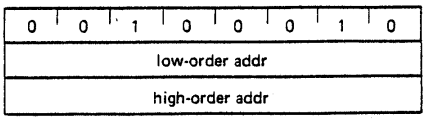
Cycles: 5
States: 16
Addressing: direct
Flags: none

SHLD addr (Store H and L direct)

((byte 3)(byte 2)) ← (L)

((byte 3)(byte 2) + 1) ← (H)

The content of register L is moved to the memory location whose address is specified in byte 2 and byte 3. The content of register H is moved to the succeeding memory location.

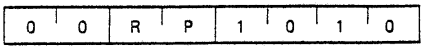


Cycles: 5
States: 16
Addressing: direct
Flags: none

LDAX rp (Load accumulator indirect)

(A) ← ((rp))

The content of the memory location, whose address is in the register pair rp, is moved to register A. Note: only register pairs rp=B (registers B and C) or rp=D (registers D and E) may be specified.

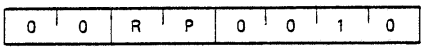


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

STAX rp (Store accumulator indirect)

((rp)) ← (A)

The content of register A is moved to the memory location whose address is in the register pair rp. Note: only register pairs rp=B (registers B and C) or rp=D (registers D and E) may be specified.



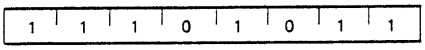
Cycles: 2
States: 7
Addressing: reg. indirect
Flags: none

XCHG (Exchange H and L with D and E)

(H) ⟷ (D)

(L) ⟷ (E)

The contents of registers H and L are exchanged with the contents of registers D and E.



Cycles: 1
States: 4
Addressing: register
Flags: none

Arithmetic Group:

This group of instructions performs arithmetic operations on data in registers and memory.

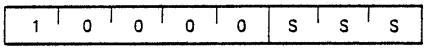
Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Carry, and Auxiliary Carry flags according to the standard rules.

All subtraction operations are performed via two's complement arithmetic and set the carry flag to one to indicate a borrow and clear it to indicate no borrow.

ADD r (Add Register)

(A) ← (A) + (r)

The content of register r is added to the content of the accumulator. The result is placed in the accumulator.

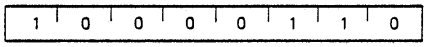


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

ADD M (Add memory)

(A) ← (A) + ((H) (L))

The content of the memory location whose address is contained in the H and L registers is added to the content of the accumulator. The result is placed in the accumulator.

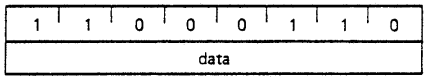


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

ADI data (Add immediate)

(A) ← (A) + (byte 2)

The content of the second byte of the instruction is added to the content of the accumulator. The result is placed in the accumulator.

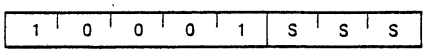


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

ADC r (Add Register with carry)

(A) ← (A) + (r) + (CY)

The content of register r and the content of the carry bit are added to the content of the accumulator. The result is placed in the accumulator.

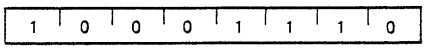


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

ADC M (Add memory with carry)

(A) ← (A) + ((H) (L)) + (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the accumulator. The result is placed in the accumulator.

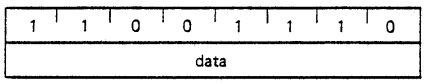


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

ACI data (Add immediate with carry)

(A) ← (A) + (byte 2) + (CY)

The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

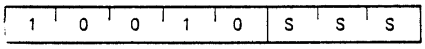


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

SUB r (Subtract Register)

(A) ← (A) – (r)

The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.

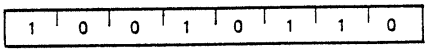


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

SUB M (Subtract memory)

(A) ← (A) – ((H) (L))

The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.

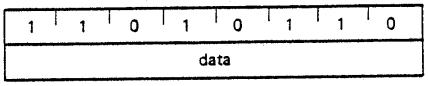


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

SUI data (Subtract immediate)

(A) ← (A) – (byte 2)

The content of the second byte of the instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.

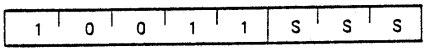


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

SBB r (Subtract Register with borrow)

(A) ← (A) – (r) – (CY)

The content of register r and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

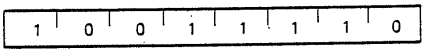


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

SBB M (Subtract memory with borrow)

(A) ← (A) – ((H) (L)) – (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

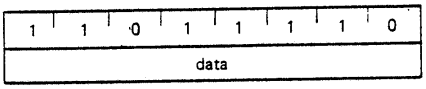


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

SBI data (Subtract immediate with borrow)

(A) ← (A) – (byte 2) – (CY)

The contents of the second byte of the instruction and the contents of the CY flag are both subtracted from the accumulator. The result is placed in the accumulator.

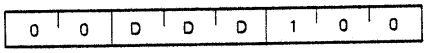


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

INR r (Increment Register)

(r) ← (r) + 1

The content of register r is incremented by one. Note: All condition flags **except CY** are affected.

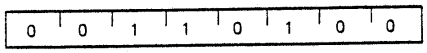


Cycles: 1
States: 5
Addressing: register
Flags: Z,S,P,AC

INR M (Increment memory)

((H) (L)) ← ((H) (L)) + 1

The content of the memory location whose address is contained in the H and L registers is incremented by one. Note: All condition flags **except CY** are affected.

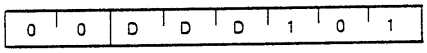


Cycles: 3
States: 10
Addressing: reg. indirect
Flags: Z,S,P,AC

DCR r (Decrement Register)

(r) ← (r) – 1

The content of register r is decremented by one. Note: All condition flags **except CY** are affected.

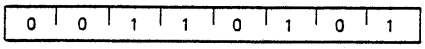


Cycles: 1
States: 5
Addressing: register
Flags: Z,S,P,AC

DCR M (Decrement memory)

((H) (L)) ← ((H) (L)) − 1

The content of the memory location whose address is contained in the H and L registers is decremented by one. Note: All condition flags **except CY** are affected.

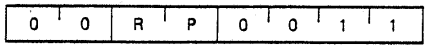


Cycles: 3
States: 10
Addressing: reg. indirect
Flags: Z,S,P,AC

INX rp (Increment register pair)

(rh) (rl) ← (rh) (rl) + 1

The content of the register pair rp is incremented by one. Note: **No condition flags are affected.**

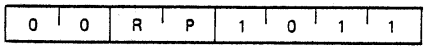


Cycles: 1
States: 5
Addressing: register
Flags: none

DCX rp (Decrement register pair)

(rh) (rl) ← (rh) (rl) − 1

The content of the register pair rp is decremented by one. Note: **No condition flags are affected.**

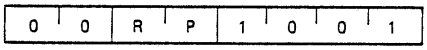


Cycles: 1
States: 5
Addressing: register
Flags: none

DAD rp (Add register pair to H and L)

(H) (L) ← (H) (L) + (rh) (rl)

The content of the register pair rp is added to the content of the register pair H and L. The result is placed in the register pair H and L. Note: **Only the CY flag is affected.** It is set if there is a carry out of the double precision add; otherwise it is reset.



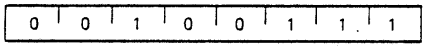
Cycles: 3
States: 10
Addressing: register
Flags: CY

DAA (Decimal Adjust Accumulator)

The eight-bit number in the accumulator is adjusted to form two four-bit Binary-Coded-Decimal digits by the following process:

1. If the value of the least significant 4 bits of the accumulator is greater than 9 or if the AC flag is set, 6 is added to the accumulator.
2. If the value of the most significant 4 bits of the accumulator is now greater than 9, or if the CY flag is set, 6 is added to the most significant 4 bits of the accumulator.

NOTE: All flags are affected.



Cycles: 1
States: 4
Flags: Z,S,P,CY,AC

Logical Group:

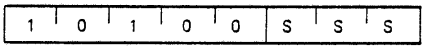
This group of instructions performs logical (Boolean) operations on data in registers and memory and on condition flags.

Unless indicated otherwise, all instructions in this group affect the Zero, Sign, Parity, Auxiliary Carry, and Carry flags according to the standard rules.

ANA r (AND Register)

(A) ← (A) ∧ (r)

The content of register r is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**

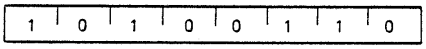


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

ANA M (AND memory)

(A) ← (A) ∧ ((H) (L))

The contents of the memory location whose address is contained in the H and L registers is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**

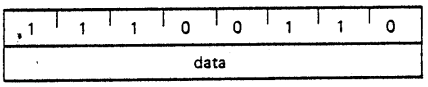


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

ANI data (AND immediate)

(A) ← (A) ∧ (byte 2)

The content of the second byte of the instruction is logically anded with the contents of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

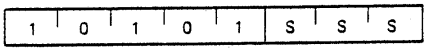


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

XRA r (Exclusive OR Register)

(A) ← (A) ∀ (r)

The content of register r is exclusive-or'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

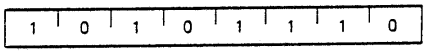


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

XRA M (Exclusive OR Memory)

(A) ← (A) ∀ ((H) (L))

The content of the memory location whose address is contained in the H and L registers is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

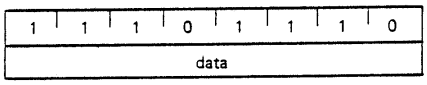


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

XRI data (Exclusive OR immediate)

(A) ← (A) ∀ (byte 2)

The content of the second byte of the instruction is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

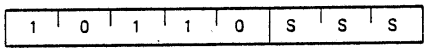


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

ORA r (OR Register)

(A) ← (A) V (r)

The content of register r is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

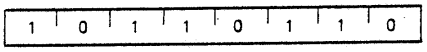


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

ORA M (OR memory)

(A) ← (A) V ((H) (L))

The content of the memory location whose address is contained in the H and L registers is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

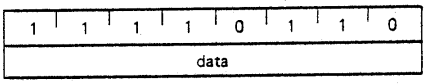


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

ORI data (OR Immediate)

(A) ← (A) V (byte 2)

The content of the second byte of the instruction is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

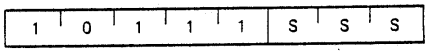


Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

CMP r (Compare Register)

(A) − (r)

The content of register r is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A) = (r). The CY flag is set to 1 if (A) < (r).**

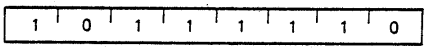


Cycles: 1
States: 4
Addressing: register
Flags: Z,S,P,CY,AC

CMP M (Compare memory)

(A) − ((H) (L))

The content of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. The Z flag is set to 1 if (A) = ((H) (L)). The CY flag is set to 1 if (A) < ((H) (L)).

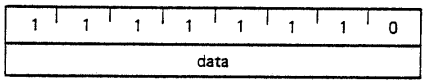


Cycles: 2
States: 7
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

CPI data (Compare immediate)

(A) − (byte 2)

The content of the second byte of the instruction is subtracted from the accumulator. The condition flags are set by the result of the subtraction. The Z flag is set to 1 if (A) = (byte 2). The CY flag is set to 1 if (A) < (byte 2).



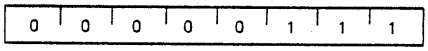
Cycles: 2
States: 7
Addressing: immediate
Flags: Z,S,P,CY,AC

RLC (Rotate left)

$(A_{n+1}) \leftarrow (A_n)$; $(A_0) \leftarrow (A_7)$

$(CY) \leftarrow (A_7)$

The content of the accumulator is rotated left one position. The low order bit and the CY flag are both set to the value shifted out of the high order bit position. **Only the CY flag is affected.**



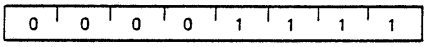
Cycles: 1
States: 4
Flags: CY

RRC (Rotate right)

$(A_n) \leftarrow (A_{n-1})$; $(A_7) \leftarrow (A_0)$

$(CY) \leftarrow (A_0)$

The content of the accumulator is rotated right one position. The high order bit and the CY flag are both set to the value shifted out of the low order bit position. **Only the CY flag is affected.**



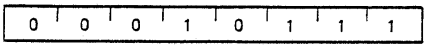
Cycles: 1
States: 4
Flags: CY

RAL (Rotate left through carry)

$(A_{n+1}) \leftarrow (A_n)$; $(CY) \leftarrow (A_7)$

$(A_0) \leftarrow (CY)$

The content of the accumulator is rotated left one position through the CY flag. The low order bit is set equal to the CY flag and the CY flag is set to the value shifted out of the high order bit. **Only the CY flag is affected.**



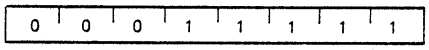
Cycles: 1
States: 4
Flags: CY

RAR (Rotate right through carry)

$(A_n) \leftarrow (A_{n+1})$; $(CY) \leftarrow (A_0)$

$(A_7) \leftarrow (CY)$

The content of the accumulator is rotated right one position through the CY flag. The high order bit is set to the CY flag and the CY flag is set to the value shifted out of the low order bit. **Only the CY flag is affected.**

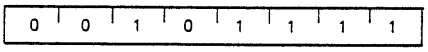


Cycles: 1
States: 4
Flags: CY

CMA (Complement accumulator)

$(A) \leftarrow (\overline{A})$

The contents of the accumulator are complemented (zero bits become 1, one bits become 0). **No flags are affected.**

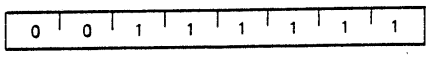


Cycles: 1
States: 4
Flags: none

CMC (Complement carry)

(CY) ← $(\overline{CY})$

The CY flag is complemented. **No other flags are affected.**

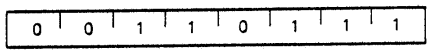


Cycles: 1
States: 4
Flags: CY

STC (Set carry)

(CY) ← 1

The CY flag is set to 1. **No other flags are affected.**



Cycles: 1
States: 4
Flags: CY

Branch Group:

This group of instructions alter normal sequential program flow.

Condition flags are not affected by any instruction in this group.

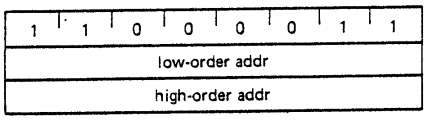
The two types of branch instructions are unconditional and conditional. Unconditional transfers simply perform the specified operation on register PC (the program counter). Conditional transfers examine the status of one of the four processor flags to determine if the specified branch is to be executed. The conditions that may be specified are as follows:

CONDITION	CCC
NZ – not zero (Z = 0)	000
Z – zero (Z = 1)	001
NC – no carry (CY = 0)	010
C – carry (CY = 1)	011
PO – parity odd (P = 0)	100
PE – parity even (P = 1)	101
P – plus (S = 0)	110
M – minus (S = 1)	111

JMP addr (Jump)

(PC) ← (byte 3) (byte 2)

Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.



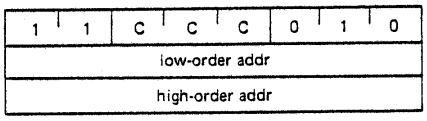
Cycles: 3
States: 10
Addressing: immediate
Flags: none

Jcondition addr (Conditional jump)

If (CCC),

(PC) ← (byte 3) (byte 2)

If the specified condition is true, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, control continues sequentially.



Cycles: 3
States: 10
Addressing: immediate
Flags: none

CALL addr (Call)

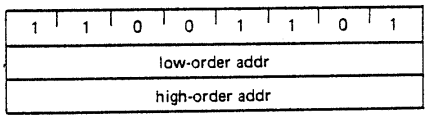
((SP) – 1) ← (PCH)

((SP) – 2) ← (PCL)

(SP) ← (SP) – 2

(PC) ← (byte 3) (byte 2)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

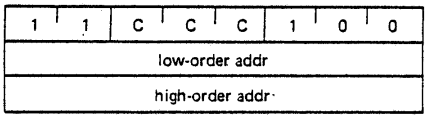


Cycles: 5
States: 17
Addressing: immediate/reg. indirect
Flags: none

Ccondition addr (Condition call)

If (CCC),
((SP) − 1) ← (PCH)
((SP) − 2) ← (PCL)
(SP) ← (SP) − 2
(PC) ← (byte 3) (byte 2)

If the specified condition is true, the actions specified in the CALL instruction (see above) are performed; otherwise, control continues sequentially.

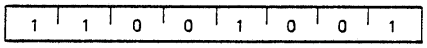


Cycles: 3/5
States: 11/17
Addressing: immediate/reg. indirect
Flags: none

RET (Return)

(PCL) ← ((SP));
(PCH) ← ((SP) + 1);
(SP) ← (SP) + 2;

The content of the memory location whose address is specified in register SP is moved to the low-order eight bits of register PC. The content of the memory location whose address is one more than the content of register SP is moved to the high-order eight bits of register PC. The content of register SP is incremented by 2.

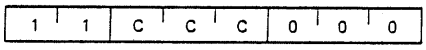


Cycles: 3
States: 10
Addressing: reg. indirect
Flags: none

Rcondition (Conditional return)

If (CCC),
(PCL) ← ((SP))
(PCH) ← ((SP) + 1)
(SP) ← (SP) + 2

If the specified condition is true, the actions specified in the RET instruction (see above) are performed; otherwise, control continues sequentially.

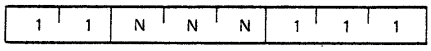


Cycles: 1/3
States: 5/11
Addressing: reg. indirect
Flags: none

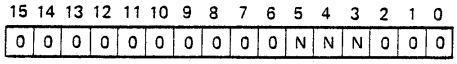
RST n (Restart)

((SP) − 1) ← (PCH)
((SP) − 2) ← (PCL)
(SP) ← (SP) − 2
(PC) ← 8 * (NNN)

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two. Control is transferred to the instruction whose address is eight times the content of NNN.



Cycles: 3
States: 11
Addressing: reg. indirect
Flags: none

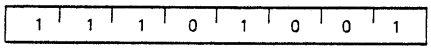


Program Counter After Restart

PCHL (Jump H and L indirect – move H and L to PC)

(PCH) ← (H)
(PCL) ← (L)

The content of register H is moved to the high-order eight bits of register PC. The content of register L is moved to the low-order eight bits of register PC.



Cycles: 1
States: 5
Addressing: register
Flags: none

Stack, I/O, and Machine Control Group:

This group of instructions performs I/O, manipulates the Stack, and alters internal control flags.

Unless otherwise specified, **condition flags are not affected by any instructions in this group.**

PUSH rp (Push)

((SP) − 1) ← (rh)
((SP) − 2) ← (rl)
(SP) ← (SP) − 2

The content of the high-order register of register pair rp is moved to the memory location whose address is one less than the content of register SP. The content of the low-order register of register pair rp is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. **Note: Register pair rp = SP may not be specified.**

1	1	R	P	0	1	0	1

Cycles: 3
States: 11
Addressing: reg. indirect
Flags: none

PUSH PSW (Push processor status word)

((SP) − 1) ← (A)
$((SP) - 2)_0$ ← (CY), $((SP) - 2)_1$ ← 1
$((SP) - 2)_2$ ← (P), $((SP) - 2)_3$ ← 0
$((SP) - 2)_4$ ← (AC), $((SP) - 2)_5$ ← 0
$((SP) - 2)_6$ ← (Z), $((SP) - 2)_7$ ← (S)
(SP) ← (SP) − 2

The content of register A is moved to the memory location whose address is one less than register SP. The contents of the condition flags are assembled into a processor status word and the word is moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two.

1	1	1	1	0	1	0	1

Cycles: 3
States: 11
Addressing: reg. indirect
Flags: none

FLAG WORD

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	Z	0	AC	0	P	1	CY

POP rp (Pop)

(rl) ← ((SP))
(rh) ← ((SP) + 1)
(SP) ← (SP) + 2

The content of the memory location, whose address is specified by the content of register SP, is moved to the low-order register of register pair rp. The content of the memory location, whose address is one more than the content of register SP, is moved to the high-order register of register pair rp. The content of register SP is incremented by 2. **Note: Register pair rp = SP may not be specified.**

1	1	R	P	0	0	0	1

Cycles: 3
States: 10
Addressing: reg. indirect
Flags: none

POP PSW (Pop processor status word)

(CY) ← $((SP))_0$
(P) ← $((SP))_2$
(AC) ← $((SP))_4$
(Z) ← $((SP))_6$
(S) ← $((SP))_7$
(A) ← ((SP) + 1)
(SP) ← (SP) + 2

The content of the memory location whose address is specified by the content of register SP is used to restore the condition flags. The content of the memory location whose address is one more than the content of register SP is moved to register A. The content of register SP is incremented by 2.

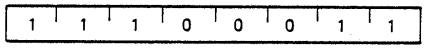
1	1	1	1	0	0	0	1

Cycles: 3
States: 10
Addressing: reg. indirect
Flags: Z,S,P,CY,AC

XTHL (Exchange stack top with H and L)

(L) ⟷ ((SP))
(H) ⟷ ((SP) + 1)

The content of the L register is exchanged with the content of the memory location whose address is specified by the content of register SP. The content of the H register is exchanged with the content of the memory location whose address is one more than the content of register SP.

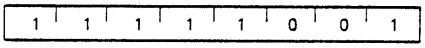


Cycles: 5
States: 18
Addressing: reg. indirect
Flags: none

SPHL (Move HL to SP)

(SP) ⟵ (H) (L)

The contents of registers H and L (16 bits) are moved to register SP.

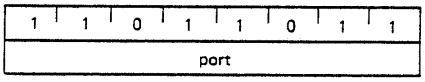


Cycles: 1
States: 5
Addressing: register
Flags: none

IN port (Input)

(A) ⟵ (data)

The data placed on the eight bit bi-directional data bus by the specified port is moved to register A.

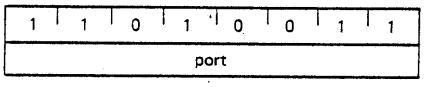


Cycles: 3
States: 10
Addressing: direct
Flags: none

OUT port (Output)

(data) ⟵ (A)

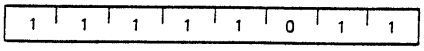
The content of register A is placed on the eight bit bi-directional data bus for transmission to the specified port.



Cycles: 3
States: 10
Addressing: direct
Flags: none

EI (Enable interrupts)

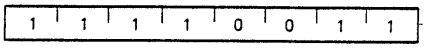
The interrupt system is enabled **following the execution of the next instruction.**



Cycles: 1
States: 4
Flags: none

DI (Disable interrupts)

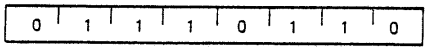
The interrupt system is disabled **immediately following the execution of the DI instruction.**



Cycles: 1
States: 4
Flags: none

HLT (Halt)

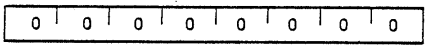
The processor is stopped. The registers and flags are unaffected.



Cycles: 1
States: 7
Flags: none

NOP (No op)

No operation is performed. The registers and flags are unaffected.



Cycles: 1
States: 4
Flags: none

INSTRUCTION SET

Summary of Processor Instructions

Mnemonic	Description	Instruction Code[1] D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock[2] Cycles
$MOV_{r1,r2}$	Move register to register	0	1	D	D	D	S	S	S	5
MOV M, r	Move register to memory	0	1	1	1	0	S	S	S	7
MOV r, M	Move memory to register	0	1	D	D	D	1	1	0	7
HLT	Halt	0	1	1	1	0	1	1	0	7
MVI r	Move immediate register	0	0	D	D	D	1	1	0	7
MVI M	Move immediate memory	0	0	1	1	0	1	1	0	10
INR r	Increment register	0	0	D	D	D	1	0	0	5
DCR r	Decrement register	0	0	D	D	D	1	0	1	5
INR M	Increment memory	0	0	1	1	0	1	0	0	10
DCR M	Decrement memory	0	0	1	1	0	1	0	1	10
ADD r	Add register to A	1	0	0	0	0	S	S	S	4
ADC r	Add register to A with carry	1	0	0	0	1	S	S	S	4
SUB r	Subtract register from A	1	0	0	1	0	S	S	S	4
SBB r	Subtract register from A with borrow	1	0	0	1	1	S	S	S	4
ANA r	And register with A	1	0	1	0	0	S	S	S	4
XRA r	Exclusive Or register with A	1	0	1	0	1	S	S	S	4
ORA r	Or register with A	1	0	1	1	0	S	S	S	4
CMP r	Compare register with A	1	0	1	1	1	S	S	S	4
ADD M	Add memory to A	1	0	0	0	0	1	1	0	7
ADC M	Add memory to A with carry	1	0	0	0	1	1	1	0	7
SUB M	Subtract memory from A	1	0	0	1	0	1	1	0	7
SBB M	Subtract memory from A with borrow	1	0	0	1	1	1	1	0	7
ANA M	And memory with A	1	0	1	0	0	1	1	0	7
XRA M	Exclusive Or memory with A	1	0	1	0	1	1	1	0	7
ORA M	Or memory with A	1	0	1	1	0	1	1	0	7
CMP M	Compare memory with A	1	0	1	1	1	1	1	0	7
ADI	Add immediate to A	1	1	0	0	0	1	1	0	7
ACI	Add immediate to A with carry	1	1	0	0	1	1	1	0	7
SUI	Subtract immediate from A	1	1	0	1	0	1	1	0	7
SBI	Subtract immediate from A with borrow	1	1	0	1	1	1	1	0	7
ANI	And immediate with A	1	1	1	0	0	1	1	0	7
XRI	Exclusive Or immediate with A	1	1	1	0	1	1	1	0	7
ORI	Or immediate with A	1	1	1	1	0	1	1	0	7
CPI	Compare immediate with A	1	1	1	1	1	1	1	0	7
RLC	Rotate A left	0	0	0	0	0	1	1	1	4
RRC	Rotate A right	0	0	0	0	1	1	1	1	4
RAL	Rotate A left through carry	0	0	0	1	0	1	1	1	4
RAR	Rotate A right through carry	0	0	0	1	1	1	1	1	4
JMP	Jump unconditional	1	1	0	0	0	0	1	1	10
JC	Jump on carry	1	1	0	1	1	0	1	0	10
JNC	Jump on no carry	1	1	0	1	0	0	1	0	10
JZ	Jump on zero	1	1	0	0	1	0	1	0	10
JNZ	Jump on no zero	1	1	0	0	0	0	1	0	10
JP	Jump on positive	1	1	1	1	0	0	1	0	10
JM	Jump on minus	1	1	1	1	1	0	1	0	10
JPE	Jump on parity even	1	1	1	0	1	0	1	0	10
JPO	Jump on parity odd	1	1	1	0	0	0	1	0	10
CALL	Call unconditional	1	1	0	0	1	1	0	1	17
CC	Call on carry	1	1	0	1	1	1	0	0	11/17
CNC	Call on no carry	1	1	0	1	0	1	0	0	11/17
CZ	Call on zero	1	1	0	0	1	1	0	0	11/17
CNZ	Call on no zero	1	1	0	0	0	1	0	0	11/17
CP	Call on positive	1	1	1	1	0	1	0	0	11/17
CM	Call on minus	1	1	1	1	1	1	0	0	11/17
CPE	Call on parity even	1	1	1	0	1	1	0	0	11/17
CPO	Call on parity odd	1	1	1	0	0	1	0	0	11/17
RET	Return	1	1	0	0	1	0	0	1	10
RC	Return on carry	1	1	0	1	1	0	0	0	5/11
RNC	Return on no carry	1	1	0	1	0	0	0	0	5/11
RZ	Return on zero	1	1	0	0	1	0	0	0	5/11
RNZ	Return on no zero	1	1	0	0	0	0	0	0	5/11
RP	Return on positive	1	1	1	1	0	0	0	0	5/11
RM	Return on minus	1	1	1	1	1	0	0	0	5/11
RPE	Return on parity even	1	1	1	0	1	0	0	0	5/11
RPO	Return on parity odd	1	1	1	0	0	0	0	0	5/11
RST	Restart	1	1	A	A	A	1	1	1	11
IN	Input	1	1	0	1	1	0	1	1	10
OUT	Output	1	1	0	1	0	0	1	1	10
LXI B	Load immediate register Pair B & C	0	0	0	0	0	0	0	1	10
LXI D	Load immediate register Pair D & E	0	0	0	1	0	0	0	1	10
LXI H	Load immediate register Pair H & L	0	0	1	0	0	0	0	1	10
LXI SP	Load immediate stack pointer	0	0	1	1	0	0	0	1	10
PUSH B	Push register Pair B & C on stack	1	1	0	0	0	1	0	1	11
PUSH D	Push register Pair D & E on stack	1	1	0	1	0	1	0	1	11
PUSH H	Push register Pair H & L on stack	1	1	1	0	0	1	0	1	11
PUSH PSW	Push A and Flags on stack	1	1	1	1	0	1	0	1	11
POP B	Pop register pair B & C off stack	1	1	0	0	0	0	0	1	10
POP D	Pop register pair D & E off stack	1	1	0	1	0	0	0	1	10
POP H	Pop register pair H & L off stack	1	1	1	0	0	0	0	1	10
POP PSW	Pop A and Flags off stack	1	1	1	1	0	0	0	1	10
STA	Store A direct	0	0	1	1	0	0	1	0	13
LDA	Load A direct	0	0	1	1	1	0	1	0	13
XCHG	Exchange D & E, H & L Registers	1	1	1	0	1	0	1	1	4
XTHL	Exchange top of stack, H & L	1	1	1	0	0	0	1	1	18
SPHL	H & L to stack pointer	1	1	1	1	1	0	0	1	5
PCHL	H & L to program counter	1	1	1	0	1	0	0	1	5
DAD B	Add B & C to H & L	0	0	0	0	1	0	0	1	10
DAD D	Add D & E to H & L	0	0	0	1	1	0	0	1	10
DAD H	Add H & L to H & L	0	0	1	0	1	0	0	1	10
DAD SP	Add stack pointer to H & L	0	0	1	1	1	0	0	1	10
STAX B	Store A indirect	0	0	0	0	0	0	1	0	7
STAX D	Store A indirect	0	0	0	1	0	0	1	0	7
LDAX B	Load A indirect	0	0	0	0	1	0	1	0	7
LDAX D	Load A indirect	0	0	0	1	1	0	1	0	7
INX B	Increment B & C registers	0	0	0	0	0	0	1	1	5
INX D	Increment D & E registers	0	0	0	1	0	0	1	1	5
INX H	Increment H & L registers	0	0	1	0	0	0	1	1	5
INX SP	Increment stack pointer	0	0	1	1	0	0	1	1	5
DCX B	Decrement B & C	0	0	0	0	1	0	1	1	5
DCX D	Decrement D & E	0	0	0	1	1	0	1	1	5
DCX H	Decrement H & L	0	0	1	0	1	0	1	1	5
DCX SP	Decrement stack pointer	0	0	1	1	1	0	1	1	5
CMA	Complement A	0	0	1	0	1	1	1	1	4
STC	Set carry	0	0	1	1	0	1	1	1	4
CMC	Complement carry	0	0	1	1	1	1	1	1	4
DAA	Decimal adjust A	0	0	1	0	0	1	1	1	4
SHLD	Store H & L direct	0	0	1	0	0	0	1	0	16
LHLD	Load H & L direct	0	0	1	0	1	0	1	0	16
EI	Enable interrupts	1	1	1	1	1	0	1	1	4
DI	Disable interrupt	1	1	1	1	0	0	1	1	4
NOP	No-operation	0	0	0	0	0	0	0	0	4

NOTES: 1. DDD or SSS – 000 B – 001 C – 010 D – 011 E – 100 H – 101 L – 110 Memory – 111 A.
2. Two possible cycle times, (5/11) indicate instruction cycles dependent on condition flags.

B. Pseudo-Ops. "Pseudo-op" is the name given to Assembly Language instructions that do not produce any machine code, but which direct the Assembler to perform its operations. The DOS Assembler provides op-codes for reserving storage space, defining the contents of memory locations and controlling the parameters of the Assembler's operation.

The following table is an alphabetical list of pseudo-ops along with their formats and functions. In these descriptions, e designates an address expression, and n designates a name. All other notation conventions are the same as in the rest of the DOS manual.

Table 4-A. DOS Assembler Pseudo-Ops

Instruction Format	Description
CMN[/<block name>/] <n1>, [<n2>, ...]	Common definition. The names n1, n2, . . . are declared to be in the Common block with the designated block name. If the block name is omitted, Blank Common is used. Each name is assumed to require one byte unless it is written in the form N(m) where m is an address expression that gives the length in bytes of the area assigned to the name N. If another CMN statement is encountered with the same block name, the first address assigned by the second statement directly follows the last address assigned by the first statement.
DATA <n1> [,<n2>],...	The names n1, n2, . . . are

	defined to be in the Data area. Each name is assumed to require one byte unless it has the form N(m) where m is an address expression that gives the length in bytes of the area assigned to N.
DB <e1> [e2] [,...] or DB"<character string>"	Define Byte. The address expressions e1, e2, ... are evaluated and stored in successive bytes in memory. The character string form stores the ASCII codes of each character in successive bytes. The two forms may be mixed in a single statement. Character Constants are treated as character strings unless they are components of address expressions.
DC "<character string>"	Define Character. The characters in the string are stored one byte per character. The high-order bit of each byte is set to zero except for the last byte which has its high order bit set to 1. This arrangement allows quick searches for the end of the string.
DS <e>	The address expression e is evaluated and defines the number of bytes of space that are allocated. The contents of the space are not affected. All names used in e must be defined prior to the DS statement.

DW <e1>[,e2] [,...]	Define Word. The address expressions e1, e2, ... are evaluated and stored as 16 bit (two-byte) words. The addresses conform to the 8080 address convention that the low-order byte comes first and the high-order byte comes second. All addresses and address offsets are handled in this way, so the DW statement must be used to define addresses.
END <e>	END is the last statement of each program. The address expression e is the execution address of the program. Specifying e=0 (absolute) is equivalent to specifying no execution address.
ENDIF	Terminates the conditional assembly started by a previous IFF or IFT statement.
ENTRY <n1>[,n2] [,...]	Define Entry Points. The names n1, n2, ... are names of entry points in other programs and are defined as names in the program being assembled. The names must appear in an ENTRY statement before they appear as labels.
EQU <e>	Define Equivalence. The address expression e is evaluated and assigned to the label of the EQU statement. The label is required and may not have appeared previously as a label or in a DMN

	or DATA statement. All names used in e must have been defined previous to the EQU statement.
EXT <n1> [,n2] [,...]	The names n1, n2, ... are defined to be external references. They may not have been used as labels or in a CMN or DATA statement.
IFF <e>	Conditional Assembly - False. If the value of the address expression e is false, (=0 absolute), then all of the statements until the next ENDIF are assembled. If the value is true, the statements are not assembled. Conditional assemblies may not be nested.

4-4. Assembler Error Messages

Assembler error messages are printed in the leftmost column of the source code listing on the line in which the error occurred. The error codes are as follows:

Table 4-B. Assembler Error Messages

Code	Meaning
2	Second operand missing. An instruction that requires two operands was only given one.
A	Absolute required. Data, Common, External or Relative address was given where an Absolute value was required.
B	Block Name error. A Common or Data block name was invalid.
C	Too many Common blocks. Only 17 Common blocks are allowed.
D	Digit invalid. Valid digits are 0 - 9 in decimal, 0 - 7 in octal and 0 - 9 and A - F in hexadecimal.
E	Expression error. Error in the syntax, symbols or position of an address expression.
F	Operand field too long.
L	Label error.
M	Multiply defined name.

N	Name too long. Six characters is maximum.
O	Op-code invalid. An Op-code was encountered which is not in the list of op-codes recognized by the Assembler.
P	Phase error. Probably an error in the Assembler. Please report errors to the MITS, Inc., Software Department.
Q	Quoted string error. The ending quotation mark was missing from a character string.
T	Field or line terminated too soon.
U	Undefined name.
V	Value invalid. An address expression value was negative, too large or otherwise unusable.

ALTAIR DOS DOCUMENTATION
SECTION V
LINKING LOADER

5. THE LINKING LOADER

5-1. Introduction

The output file of the Assembler is a relocatable object code module. That is, it is a machine language program module (object code) that can be loaded by the appropriate loading program--anywhere in memory and executed (relocatable). Moreover, the Assembler allows the module produced by an assembly to refer symbolically to addresses in other modules as long as all of the modules that refer to each other are loaded into memory at the same time (see page 71, EXT pseudo-op).

The program that loads relocatable modules into memory and links their symbolic references to the proper addresses is called the Linking Loader (LINK). In the simplest case, where an entire program is contained in one module, LINK loads the program into memory and causes control to jump to its starting address.

In the more complex case, where several modules are to be loaded into memory and linked together to form a single large program, LINK serves many functions. It loads the modules and makes sure that bytes of a module are not destroyed by loading subsequent modules in overlapping locations. It makes the connections between all external references and the addresses to which they refer. It prints lists of those external references for which no addresses have been defined. It can even search the disks for files to resolve these undefined references and automatically load them. All of these functions are controlled by the Linking Loader's commands which are described in Table 5-A. For an explanation of the use of LINK in this case, see Appendix E.

If the system disk is mounted on drive zero, the Linking Loader is loaded and run by typing the following command to the Monitor:

```
.LINK
```

When LINK starts, it prints the following message:

```
DOS LINK VER x.x
*
```

The asterisk means LINK is ready to receive commands.

Table 5-A. Linking Loader Commands

L <file> <device> [<address at which to load relocatable module>]	Loads a module at the specified address. The module is loaded from the specified disk. The module must be in LINK's relocatable code format. If the loading address is not specified, the default address is 24000_8 for the first module to be loaded and the next available location above the previous module for all subsequent modules. The L command automatically adds a * to the file name. For an example of the use of the L command, see Appendix E, Section 2.
A	Displays the names in all of the currently loaded modules and their assigned addresses. Undefined names are displayed with asterisks instead of addresses.
U	Displays all undefined names in all current modules.
S <device>	For each undefined entry point name, LINK searches the specified device for a relocatable file by that name and loads it. For an example of the use of the S command, see Appendix E, Section 2.
E	Exits to the Monitor
X [execution address]	Begins execution of the program at execution address . If the execution address is omitted, X branches to the address in the

last encountered END statement. If no END statement has been encountered, X branches to location 24000_8.

5-2. Address Chaining

Each time LINK encounters a reference to a symbol that has not yet been defined, it enters the address of the reference into a chain. Each entry in the chain contains a pointer to the previous entry. The last entry contains zero absolute. When the symbol is defined, LINK goes through the chain again from the last entry to the first, replacing the contents of each entry with the assigned address of the symbol. As a result of this process, each reference to the symbol points to the correct address.

LINK handles external references by saving the unresolved chains from all of the modules. The contents of the first entry in a chain for one module is the address of the top of the chain for the previously loaded module.

The U command can be used to display the undefined symbols in all loaded modules.

5-3. Relocatable Object Code Module Format

The Assembler creates and LINK uses files which conform to the Relocatable Object Code Module format. Each module consists of records of 1024 bits each. A record is made up of a number of load items, each one of which is preceded by at least one control bit.

A. If the first bit is 0, the next eight bits are loaded as an absolute data byte. If the first bit is 1, the next two bits are input as a control field as follows:

B.

Control Bits	Action
01	The following 16 bits are loaded as a relocated address after adding the relocation base address.
10	The following 16 bits are to be loaded as a Data block reference address after adding the Data base.

11	The following 16 bits are to be loaded as a Common block reference address by adding the current Common base.
00	The next 9 bits are to be input as a control field and the following 16 bits as an address.

C. The 9-bit control field has the following format:

aannnxxxx

where aa designates the type of the address

aa	Type
00	Absolute
01	Relocated, relocation base is added before loading.
10	Data reference. Data base is added before loading.
11	Common reference, current Common base is added before loading.

nnn is the length, in bytes, of the program or common block name. When nnn = 0, the name is blank. If a name is specified, it immediately follows the address in the module.

xxxx is a 4 bit control field as follows:

xxxx	Action
1	Define Common Size. The address is interpreted as the size of the Common block that has the specified name. This type of item may be preceded only by Define Entry Name items. The program with the largest blank Common block must be loaded first. All programs which refer to named Common blocks must define them to be the same size.
2	Define Data Size. The address is interpreted as the size of the Data area. If this item is preceded only by Define Entry Name and Define Common Size items, normal memory allocation takes place.

If, however, Data block references occur before this item is loaded, the Data base is assigned to be the address of the first location from the top of memory, and all Data block reference addresses are subtracted from rather than added to the base.

3 Set Location Counter. The address is loaded into the loading location counter.

4 Address Chain. The current value of the loading location counter is placed in each element of the chain whose top element is the address.

5 Set Common Base. The assigned address of the named Common block is the current Common base.

6 Chain & Call an External Name. The name is placed into the loader table, if it is not already there. The address chain whose top element has the specified address is linked to the chain for the name if it has not yet been loaded or to the name (if it has been loaded).

7 Define Entry Point. The address is assigned to the named entry point.

8 Define Program Limit. The address is that of the first location after the program.

14 End of Record. This record indicates the end of the program being loaded and the end of data in this record. A is the execution address.

15 End of Module. End of load module. Control returns to the loader.

ALTAIR DOS DOCUMENTATION
SECTION VI
DEBUG

6. DEBUG PACKAGE

6-1. Introduction

The Debug package is a system program which provides facilities for debugging Assembly Language programs. Commands allow the following operations:

1. Display the contents of memory locations, registers or flags in several modes (octal, decimal, etc.)
2. Modify the contents of memory locations, registers or flags.
3. Insert, display and remove breakpoints to initiate pauses in program execution.
4. Start execution of the program at any address or at any breakpoint.

A. Running Debug. After the system disk is mounted in drive zero, Debug is entered from the Monitor by typing

.DEBUG

Debug indicates that it is loaded and running by printing

DOS DEBUG VER x.x

on the terminal. At this point, it is ready to receive commands. The Monitor may be reentered by typing R.

B. Addressing Modes. Debug can display, modify or transfer program control to any point in memory. In addition, entry to Debug causes the registers and condition flags to be stored in memory, making them available for display or modification.

Most of the Debug commands may be preceded by an address. This address may be expressed in any one of several modes.

1) Explicit. Anywhere an address is expected, a number is interpreted as an octal address. A number preceded by a pound sign (#) is interpreted as a decimal address. The address is entered into an address pointer in Debug. All commands operate on the location in the address pointer. The current contents of the address pointer may be accessed by typing a period (.). Thus,

the Debug command

./

displays the contents of the location whose address is currently in the address pointer. The use of the period is optional, in this case, since

./

and

/

cause the same operation to be performed.

2) Relative. An address may be specified in the following form:

<address> ± <offset>

For example:

100 + 10, the location whose address is 100_8 + 10_8 or . - 2 refers to the location whose address is that of the current location minus 2_8.

Two special cases of indirect addressing involve the <line feed> and <↑> commands.

<line feed> increments the address pointer and displays the contents of the resulting location.

<↑> (<^> on some terminals) decrements the address pointer and displays the contents of the resulting location.

In both cases, the increment in the symbolic I/O mode (see Section 2-1) is the length of the current instruction - 1, so that the next location displayed is that of the next instruction. In the W mode, the increment is 2 bytes and in all other modes the increment is one byte.

Typing an equal sign (=) after a relative address specification causes Debug to print the resultant address.

3) Indirect. Typing <tab> (Control/I) refers to the location whose address is the contents of the current

location. For example:

```
70/  JMP 5000 <tab>
5000/  SHLD 4750
```

Typing 70/ in the symbolic I/O mode W causes Debug to display the instruction at 70 which is a JMP to location 5000. Typing <tab>, which is equivalent to .<tab>, causes Debug to reference the instruction at location 5000. Subsequently, typing / causes the instruction at location 5000 to be displayed.
Typing <tab> when the current location is the low order byte of a two-byte address or the low order register of a register pair causes the address pointer to be loaded with the contents of both bytes of the address or the pair of registers.

4) Register. When Debug is entered, or when a breakpoint is encountered, Debug stores the contents of the registers and condition flags in memory in the following order:

Register	Remarks
F	Condition Flags

Bit	Meaning
0	Carry
2	Even Parity
4	Half Carry (for decimal arithmetic)
6	Zero
7	Sign (One means the MSB of result was 1)

Register	Remarks
A	Accumulator
C	Note: The low order register of a pair is first)
B	
E	
D	
L	

H

S Low order byte

S High order byte

Once a register has been opened, typing <line feed> or <↑> causes the next or preceding register in the list to be accessed and displayed.

6-2. Display

Typing the following command:

<address>/

where the address is in any mode, causes Debug to display the contents of the specified location in the current I/O mode.

A. I/O Modes. Debug displays the contents of locations in several modes which may be specified by the programmer. The I/O mode is specified by typing dollar sign ($) or <ESCAPE> (<Altmode> on some terminals) followed by a letter.

Letter	I/O Mode
O	Octal
D	Decimal
W	Double byte octal. Displays contents of two successive locations. This is used primarily to display addresses.
A	ASCII. The characters displayed have ASCII codes equal to the contents of the location.
S	Symbolic. The instruction at the location is displayed in Assembly Language symbolic form. All bytes of the instruction are displayed, but address bytes are displayed in octal form.

If no I/O mode is specified, Debug proceeds as if the mode were specified as octal. Typing a semicolon (;) instead of / displays the contents of the current location in octal, regardless of the current I/O mode.

B. Displaying a Range of Locations. Typing the following command:
<address 1>, <address 2>T
displays the contents of all the locations from <address 1> to <address 2>, inclusive, in the current I/O mode.

6-3. Modify

The contents of a location may be modified by displaying the current contents of the location and then typing the new contents. For example

```
50/ XRA A  ORA A  <cr>
./ ORA A
```

The instruction ORA A replaces the original XRA A. All input after the display is used to modify the current location until the location is filled or until a delimiter is typed. The normal delimiter is <cr>. Other delimiters are as follows:

<line feed>	displays the next location
<↑>	displays the previous location
/ or ;	displays the modified contents of the current location
<tab>	displays contents of the location addressed by current location (typed as Control/I).
<ESCAPE>, +, @, !, =	are special and terminate input even though they have no specific function in this context

Input is interpreted according to the current I/O mode. If the input cannot be interpreted, "?" is printed on the terminal and the command must be repeated.

6-4. Breakpoints

Breakpoints provide the ability to pause in the execution of a program at any point and examine the contents of memory locations, registers and condition flags. A breakpoint is set by the X command, which has the following form:

<address> X

This command sets the next available breakpoint at the specified address. Eight breakpoints are available (numbered 0 - 7). When a breakpoint is encountered during execution of the program, the following message is printed on the terminal:

<number> BREAK@ <address>

Execution is suspended until it is restarted by a P or G command.

The positions of all the breakpoints in use can be displayed by the Q command:

Q<cr>

Example:

```
10X
20X
377X
Q
0 @ 10
1 @ 20
2 @ 377
```

Any (or all) breakpoints may be removed by the Y command:

```
Y
```

or

```
Y<number>
```

If no number is specified, all breakpoints are removed. If a number is specified, only that breakpoint is removed.

6-5. Controlling Execution

Debug may be used to control the execution of a program by means of the G and P commands.

A. The G Command. Execution can be started at any location by the G command:

```
<address>G
```

where the address is the location where execution is to start.

B. The P Command. Execution can be made to proceed from a breakpoint by means of the P command:

```
[<number>] P
```

If the number is typed, execution proceeds from the specified breakpoint. If the number is omitted, the most recently encountered breakpoint is specified. The P command cannot be used if no breakpoint has been encountered or if the breakpoint with the specified number has not been assigned.

C. Breakpoints and Execution Commands. When a G or P command is executed, Debug replaces the bytes at the breakpoint addresses with RST instructions. These instructions cause control to be transferred to locations 0, 7, 17, 27, 37, ... 77. At these locations, JMP instructions branch to a breakpoint handling routine in Debug. The bytes that were replaced are saved in a table and stored after the breakpoint is executed.

When a P command is executed, Debug reconstructs the instruction at the breakpoint by referring to the table and executes that instruction before branching to the instruction after the breakpoint. If the instruction at the breakpoint is itself a CALL, JMP or RST instruction, Debug branches to the proper location.

When a breakpoint RST is executed, the breakpoint routine saves all registers and condition flags and restores the original byte in the instruction string. In operation, the breakpoint processing procedure is transparent to the programmer and program execution is unaffected, except for the pauses initiated by the breakpoints.

6-6. Using Debug with Relocated Programs

The Assembler produces relocatable code modules that can be loaded in any place in memory by the Linking Loader. Thus, the addresses of program statements are not determined until the program is loaded. In order to use Debug on such programs, special functions are provided for handling base addresses.

Typing an apostrophe (') recalls the execution address returned by the Linking Loader for the current load module. Thus, the statement

```
'G
```

causes Debug to start execution of the module at the Linking Loader execution address.

The execution address may or may not be the first location in the program. For this reason, Debug also includes the capability of storing any address and recalling it for use in any Debug command. The statement

```
<address>%
```

stores the address and

```
&
```

recalls it for use. The address may be that of the first location in a module, common or data block, etc.

ALTAIR DOS DOCUMENTATION
SECTION VII
MISCELLANEOUS SYSTEM PROGRAMS

7. MISCELLANEOUS SYSTEM PROGRAMS

7-1. INIT

INIT is a system program that allows the initialization of the system (the number of disks, disk files, etc.) to be changed without reloading the system. INIT is run by typing

.INIT

to the Monitor. INIT then prints the question

MEMORY SIZE?

and the initialization dialog proceeds exactly as it does when the system is loaded (see Section 1-2c, p. 7).

7-2. CNS

CNS allows the console through which the user issues commands to be changed to another terminal. To use CNS, type

.CNS <channel> <sense switch>

to the Monitor, where <channel> is the octal data channel number of the new console's I/O board, and <sense switch> is the new I/O board's octal sense switch setting. The data channel is the low order channel of the board and the sense switch settings are shown in Table 1-A on page 5.

For example, to switch to a terminal using a 2SIO board with 2 stop bits through channel 20, the following command is typed:

.CNS 20 0

7-3. SYSENT

SYSENT is a system program file that contains addresses of several Monitor routines that are available for user program use. The following routines are available:

ABORT	exits to the Monitor and prints "PROGRAM ABORTING" on the terminal
EXIT	exits to the Monitor and prints "PROGRAM EXITING" on the terminal

ABORT and EXIT both return control from the program to the Monitor and close all files. The program name is found in location TASKNM (see below). ABORT is generally used to exit under error conditions while EXIT is used under normal exit conditions.

IO — allows access to the Monitor Call I/O routines. The following sequence is used in the calling program

```
CALL IO
DW  (address of Request Control
    Block)
```

See Appendix C for more information on Monitor Calls and Request Control Blocks.

Two special routines are used to print text messages.

TASKNM — contains the address of the memory area where ABORT and EXIT find the name of the calling program. The program name must be stored at this location before an ABORT or EXIT call is executed.

MSG — prints a user selected message on the terminal. The following sequence is used:

```
CALL   MSG
DW     (address of first byte
       of message)
```

MSG prints the message bytes until it prints a byte with the most significant bit set to one. Thus, the message should be stored with a DC pseudo-op.

To use the routine in SYSENT, the desired names must be defined as External names in the calling program. (See EXT statement, Table 4-A.) When the calling program is loaded into memory for execution, SYSENT must also be loaded. The following Linking Loader command is used for this purpose:

```
L SYSENT 0
```

This command loads SYSENT just above the user program.

7-4. LIST

LIST is a BASIC language routine that allows DOS Assembler listing files to be printed on a line printer. To use LIST, BASIC must be running and the DOS disk must be mounted. The following command runs LIST

RUN"LIST",<device number>

where the device number is that of the disk drive upon which the DOS disk is mounted.

LIST asks for the name of the program (the % sign is added automatically) and the device number of the disk on which the listing file resides. The listing is then printed on the system line printer.

ALTAIR DOS DOCUMENTATION
APPENDICES

APPENDIX A. ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
ØØØ	NUL	Ø43	+	Ø86	V
ØØ1	SOH	Ø44	,	Ø87	W
ØØ2	STX	Ø45	-	Ø88	X
ØØ3	ETX	Ø46	.	Ø89	Y
ØØ4	EOT	Ø47	/	Ø9Ø	Z
ØØ5	ENQ	Ø48	Ø	Ø91	[
ØØ6	ACK	Ø49	1	Ø92	\
ØØ7	BEL	Ø5Ø	2	Ø93	]
ØØ8	BS	Ø51	3	Ø94	^
ØØ9	HT	Ø52	4	Ø95	<
Ø1Ø	LF	Ø53	5	Ø96	'
Ø11	VT	Ø54	6	Ø97	a
Ø12	FF	Ø55	7	Ø98	b
Ø13	CR	Ø56	8	Ø99	c
Ø14	SO	Ø57	9	1ØØ	d
Ø15	SI	Ø58	:	1Ø1	e
Ø16	DLE	Ø59	;	1Ø2	f
Ø17	DC1	Ø6Ø	<	1Ø3	g
Ø18	DC2	Ø61	=	1Ø4	h
Ø19	DC3	Ø62	>	1Ø5	i
Ø2Ø	DC4	Ø63	?	1Ø6	j
Ø21	NAK	Ø64	@	1Ø7	k
Ø22	SYN	Ø65	A	1Ø8	l
Ø23	ETB	Ø66	B	1Ø9	m
Ø24	CAN	Ø67	C	11Ø	n
Ø25	EM	Ø68	D	111	o
Ø26	SUB	Ø69	E	112	p
Ø27	ESCAPE	Ø7Ø	F	113	q
Ø28	FS	Ø71	G	114	r
Ø29	GS	Ø72	H	115	s
Ø3Ø	RS	Ø73	I	116	t
Ø31	US	Ø74	J	117	u
Ø32	SPACE	Ø75	K	118	v

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
Ø33	!	Ø76	L	119	w
Ø34	"	Ø77	M	12Ø	x
Ø35	#	Ø78	N	121	y
Ø36	S	Ø79	O	122	z
Ø37	%	Ø8Ø	P	123	{
Ø38	&	Ø81	Q	124	|
Ø39	'	Ø82	R	125	
Ø4Ø	(	Ø83	S	126	
Ø41	)	Ø84	T	127	DEL
Ø42	*	Ø85	U		

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout

APPENDIX B

DISK INFORMATION

1. FORMAT OF THE ALTAIR FLOPPY DISK

1-1. Track Allocation

Track	Use
0 - 5	DOS Memory Image
6 - 69	Space for either Random or Sequential files
70	Directory Track
71 - 76	Space for Sequential files only

1-2. Sector Format

There are 32 sectors per track and 137 bytes per sector. Of these bytes, 128 are available for data storage.

Tracks 0 - 5

Byte	Use
0	Track number + 128 decimal
1 - 2	Sixteen bit address of the next higher location in memory than the highest location saved on this sector
3-130	128 bytes of DOS code
131	Stop byte (255 decimal)
132	Checksum. Sum of the bytes 3 - 130 with no carry out of one byte

Tracks 6 - 76

Byte	Use
0	Most significant bit always on. Contains track number plus 200 octal.
1	(Sector number)*17 MOD 32
2	File number from directory. Zero means this sector is not part of any file. If the sector is the first of a group of 8 sectors, 0 means the whole group is free.
3	Number of data bytes written (0 to 128). This is always 128 for random file data blocks. For random file index blocks, this number is the number of groups allocated for this file.
4	Checksum. Sum of bytes 3 - 134 with no carry out of one byte.

Byte	Use
5, 6	Pointer to the next group of the file. The first byte is the track number and the second byte is the sector number. Zero indicates the end of the file.
7 - 134	Data
135	Stop byte (255 decimal)
136	Unused

1-3. The Directory Track

The Directory takes all of track 70. Each sector has 8 file name records, each 16 bytes long. The format of the sector is as follows:

Byte	Use
0 - 7	File name
8, 9	Pointer to the start of the file (track, sector).
10	File mode. 2=sequential, 4=random
11 - 15	Unused

If the first byte of the file name is 0, the file has been deleted. If the first byte is 255 decimal, the file is the last in the directory and all file name records after it are ignored.

2. RANDOM FILES

2-1. Format of Random Files

A random file may contain any number of sectors. The first two sectors are the "index blocks." The "Number of Data Bytes" field in the first block indicates the number of groups currently allocated to this file. The next 256 bytes in the two blocks give the designations of the data sectors in the file in the order they occupy in the file. The upper two bits in the byte give the group number and the lower 6 bits give the track number - 6.

2-2. Using Random Files

The user must allocate a 128 byte buffer for each random file to be open at one time in the program. A Random Read or Write transfers an entire 128 byte block at a time into or out of the buffer assigned to that file.

The format of the data in the buffer is defined by the user.

APPENDIX C. MONITOR CALLS

Since the Monitor contains all the I/O routines for all of the peripheral devices in the system, it is not necessary for the programmer to write I/O routines for each program. Instead, the program can call the Monitor to handle all input and output.

For this reason, DOS I/O is device-independent. The programmer need not consider the idiosyncracies of individual I/O devices when a program is being written, and the I/O device can be chosen at the time the program is executed.

The instruction sequence for calling the Monitor from an Assembly language routine is as follows:

```
CALL IO                              ;IO IS DEFINED IN SYSENT
DW (Request Control Block address) ;A SYSTEM PROGRAM FILE (SEE
                                     SECTION 7-3).
```

The Request Control Block (RCB) is a block of data which provides the information the Monitor needs to perform the requested operation.

The first two bytes in every Request Control Block have the same significance. The first byte is always the operation code byte which tells the Monitor the action being requested. The second byte is a status byte which is set to zero if the operation is completed successfully and to a non-zero value if an error occurred. The error codes are in Appendix

In the list that follows, the Request Control Blocks for each I/O Monitor call are given, beginning with the third byte. When an RCB is constructed, DB statements can be used to define the byte quantities and DW to define the two-byte quantities. This is because the two-byte quantities are interpreted as addresses and must conform to the 8080's format for addresses (first byte is the low order byte).

I/O MONITOR CALLS

Operation	Code	Description
Open	104	Prepares a file for input or output. Assigns a file number to the file. A file must be opened before information can be transferred to or from it. The next Read or Get operation after Open begins with the first byte in the file.

Byte	Function
3	File number. The file is referred to by this number until it is closed.
4	File type. The bits of the file type byte have the following significance: 0 - sequential input 1 - sequential output 2 - random. Open for input and output simultaneously. 7 - explicit device specification. If bit 7 is on, transfer takes place through the device specified in bytes 5 and 6. Otherwise, bytes 5 and 6 are ignored and transfer takes place through the last device used for this file. Note: Bit 0 is the least significant bit. Only one bit may be on at one time.
5	Kind of Device 0 - Teletype 1 - cassette tape 6 - floppy disk
6	Device number
7, 8	Address of file name area

Close	105	Ends the connection between a file number and a file. Normal exit from a system program or jumping to location zero causes all files to be closed.

Byte	Function
3	File number

Read	102	Reads a number of bytes from a sequential file - either on disk or on another I/O device

Byte	Function
3	File number
4	Mode. The bits of the mode byte have the following significance: Bit 1 on - Echo. Prints all characters as they are entered. Bit 1 off - no echo. Bit 2 on - ASCII. Control/R Control/U and Rubout recognized, input terminates on <cr>. Bit 2 off - Absolute binary code. Note: Bit 0 is the least significant bit.
5, 6	Address of input buffer.
7, 8	Number of bytes to be transferred (two-byte quantity interpreted as an address)
9, 10	Number of bytes actually transferred (interpreted as an address). This operation begins by reading the next byte after the last byte to be read and reads the specified number of bytes.

Write	103	Writes a number of bytes into a file on a disk or another I/O device. The bytes are written after the last byte in the file.
	Byte	Function
	3	File number
	4	Mode. The bits of the mode byte have the following significance: Bit 2 on - ASCII. Adds nulls to the end of the line, expands tabs. Bit 2 off - Absolute. Note: Bit zero is the least significant bit.
	5, 6	Address of write buffer
	7, 8	Number of characters to be written (interpreted as an address)
	9, 10	Number of bytes actually transferred (interpreted as an address)
Random Read	4	Reads a 128-byte record from a random file on disk. The record is read into a 128 byte buffer in memory which must have been previously allocated. An error results if a Random Read is performed on a sequential file.
	Byte	Function
	3	File number
	4, 5	Address of memory buffer
	6, 7	Record number (interpreted as an address)
Random Write	5	Writes a 128 byte record into a random file. The record is written from a 128 byte memory buffer. An error results if a Random Write is performed to a sequential file.

	Byte	Function
	3	File number
	4, 5	Address of memory buffer
	6, 7	Record number (interpreted as an address)
Get Character	2	Reads the next character (1 byte) from an input file. If the file is on disk, it must be opened for input. The first Get after Open reads the first character in the file.

	Byte	Function
	3	File number
	4	Byte reserved for the character to be read
Put Character	3	Writes a character (1 byte) on an output file. The character is added to the end of the file. If it is a disk file, the file must be opened for output first.

	Byte	Function
	3	File number
	4	Character to be written
Block Input	107	Reads a sector (128 bytes) from a disk file* into a buffer in memory. Returns the address of the first data byte in the buffer and a pointer to the number of bytes in the block.

	Byte	Function
	3	File number
	4, 5	Pointer to number of bytes in the block
	6, 7	Pointer to first available data byte

*Block Input may be used to input data from a terminal. In that case, only 1 byte is transferred into the buffer. Use of Block Input in this

way may save programming effort, but Get Character is much faster and more efficient.

Block Output	110	Writes a sector (128 bytes) to a disk file*. Returns the addresses of the first byte of the next 128-byte buffer to be written and the number of empty bytes in the buffer. To write a block of data, the Block Output routine is called to get pointers to the memory buffer. The buffer is then filled with data to be output and the Block Output routine is called again to write the data. Each successive Block Output call returns pointers to be used by the next Block Output call.

Byte	Function
3	File number
4, 5	Pointer to the number of bytes left empty in the buffer. When this number is zero, the buffer is full.
6, 7	Address of the first byte in the buffer.

*Block Output may be used to output data to a terminal. In that case, each Block Output call outputs one byte.

These Monitor calls are used in the following manner: The Input or Output routine is called to get the pointers to the buffer. In the Input case, the buffer is filled with input data. In the Output case, the program must fill the buffer with data to be output. As each byte is transferred either to or from the buffer, the byte counter (pointed to by bytes 4 and 5) is decremented. When the counter reaches zero, the transfer to or from the buffer is complete. Calling Block Output again writes the buffer onto the specified disk file and returns new pointers. Calling Block Input again reads another sector of data and returns new pointers.

In addition to these I/O Monitor Calls, Monitor Calls are available which perform the operations of the Monitor commands. These calls allow files to be opened, saved and deleted; disks to be mounted and dismounted, etc. without having to return control to the Monitor. The first two bytes of each of the command Monitor Calls are the same as the I/O Monitor Calls except for the codes. The listings below show the rest of the bytes of the Request Control Blocks.

Operation	Code	Description
Initialize	45	Same as DIN command
	Byte	Function
	3	Kind of device (disks are the only devices currently supported). Byte = 6.
	Byte	Function
	4	Device number
Rename	44	Same as REN command
	Byte	Function
	3	Kind of device = 6 for disk
	4	Device number
	5, 6	Address of 8-byte old name field
	7, 8	Address of 8-byte new name field
Delete	43	Same as DEL command
	Byte	Function
	3	Kind of device = 6 for disk
	4	Device number
	5, 6	Address of 8 byte file name
Directory	42	Same as DIR command
	Byte	Function
	3	Kind of device = 6 for disk.
	4	Device number
	5, 6	File number where the output of the directory is to be written. The file must be open for output.
Dismount	41	Same as DSM command.

	Byte	Function
	3	Kind of device = 6 for disk
	4	Device number
Mount	40	Same as MNT command.
	Byte	Function
	3	Kind of device = 6 for disk
	4	Device number
Save	106	Same as SAV command.
	Byte	Function
	3	Kind of device 6 for disk 0 for Teletype
	4	Device number
	5, 6	Address of 8 byte file name
Load	100	Same as LOA command
	Byte	Function
	3	Kind of device 0 for Teletype 1 for cassette tape 6 for floppy disk
	4	device number
	5, 6	address of 8 byte file number
	7, 8	address of first byte to be saved
	9, 10	address of last byte to be saved
	11, 12	starting address

APPENDIX D. ABSOLUTE LOAD TAPE FORMAT

The paper tape dump of an object program consists of 3 records. The Begin/Name record is first, and carries the name of the program and comments (version number, date, etc.) The program records follow the Begin/Name record. The last record is an end-of-file record. The formats of the records are as follows:

A. Begin/Name Record

Byte 1	125Q	Begin record sync byte
2-4	Name	Program name
5-N	15Q	Terminates the Begin/Name record

B. Program Record

Byte 1	74Q	Program record sync byte
2		Number of bytes in this record
3, 4	Load Address	Low order byte is first
5-N	Program Data	
N+5	Checksum	All bytes except the first two are added with no carry to generate a checksum byte used to detect load errors.

C. End-of-File Record

Byte 1	170Q	EOF Record sync byte
2, 3	Begin Execution Address	

APPENDIX E. THE FILE COPY UTILITY

1. As an example of the use of the various facilities of DOS to solve a specific problem, the listing of a file copying routine is given in this appendix.

This program copies a file from one file and device to another. Any file on any device in the system may be copied to any other device with this program.

The program is highly structured, with a central routine (COP) that calls a number of other routines to perform specific actions.

To copy a file, run the copy program by typing the following command to the Monitor:

```
.COP
```

The program is stored on disk as an absolute binary file so it is loaded and run immediately. When the program starts, it prints the following messages:

```
COPY FILE
SET UP INPUT
```

It then asks for the type of device from which the file is to be copied. The user answers with "FDS" for a disk or "TTY" for the terminal. At this point, the copy program asks the device number (0, if there is only one device of that type) and the name of the file to be copied. If the device is "TTY", no file name need be specified. After the input parameters have been entered, the program prints

```
SET UP OUTPUT
```

and asks the device type, number and file name for output. If the output device is "TTY", no output file name need be specified.

When the copy action is complete, the program exits.

This Appendix lists the main routine COP and some of the more important or instructive subroutines. For a complete listing of the routines, use COP to copy them to the terminal. To do this, specify the output device as TTY and copy the following routines.

```
&DN      &TABLE      &ASK
&DTYP    &COP        &SYSENT
&LDEM    &CMPB
&MOVB    &AANS
```

2. To run the copy program from the Assembly Language source files on the system disk, it is first necessary to assemble all of the files in the list above. To do this, type the following command:

```
.ASM COP 0
```

when the file is assembled, ASM prints

```
000000 ERRORS DETECTED
ANY MORE ASSEMBLIES?
```

The programmer replies to this question with the name of the next program to be assembled. This process continues until all of the programs in the list have been assembled. To load these modules into memory and link them together into the copy program, the Linking Loader is run with the following command:

```
*LINK
```

When LINK prints its prompt asterisk, the main copy program module COP can be run with the following command:

```
*L COP 0
```

At this point, LINK loads the module into memory and resolves the references to all symbolic addresses. Since numerous other symbols are as yet undefined, DOS prints a list of these symbols as follows:

```
TSKNM    * MSG     * DTYP    * DN      * ASK
* MOV8    * IO      * EXIT    * BDEX    *
ABORT    * GDEX    *
```

The asterisks after each file number indicate that the names are undefined. These names are all those of entry points in the modules that have not been loaded.

To load some of the required modules, the following command may be typed:

```
*S 0
```

The S command adds asterisks to the undefined names and searches the specified disk for files with the resulting names. When LINK finds such a file, it loads and links it. Finally, LINK prints a list of those entry names that are still undefined:

```
TSKNM    *    MSG    *    MOV8    *    IO
*    EXIT    *    ABORT    *
```

Entry point MOV8 is contained in file MOVB, so that it can be defined by the following command:

```
*L MOVB 0
```

The remaining entry names are in file SYSENT which is loaded with the following command:

```
*L SYSENT 0
```

Now that all of the required modules are loaded and linked together, the entire program is ready to be executed with the following command:

```
*X
```

The copy program starts up and prints its prompt questions as above.

COP LISTING

The following statements define the entry point and external references.

```
000100          ENTRY   COP
000200          EXT     EXIT,ABORT
000300          EXT     TASKNM,MSG
000400          EXT     MOV8,IO
000500          EXT     DTYP,DN,ASK
000600          EXT     GDEX,BDEX
000700 ;
000800 ;IDENTIFY PROGRAM AND SET RADIX
000900 ;
001000 COP      LXI     H,COPID ;GET PRGID
001100          SHLD    TASKNM  ;PUT AWAY
001200          CALL    MSG     ;DISPLAY IT
001300          DW      COPID
```

The setup routines are basically a series of Monitor Calls. They ask the operator for the file name and disk number, open the required files and check to make sure everything is operating properly.

```
001400 ;
001500 ;SET UP INPUT FILE
001600 ;
001700          CALL    MSG     ;TEL OPR WHATS GOING ON
001800          DW      SETUIN
001900          CALL    DTYP    ;INPUT DEVICE TYPE
002000          STA     DTIN
002100          CALL    DN      ;DEVICE NUMBER
002200          STA     DNIN
002300          CALL    ASK     ;FILE NAME
002400          DW      ASFNM
002500          LXI     D,FNIN  ;PUT IT AWAY
```

```
002600          CALL    MOV8
002700          CALL    IO      ;OPEN FILE
002800          DW      RBINOP
002900          LDA     STINOP  ;CHECK STATUS
003000          ORA     A
003100          JNZ     NOINOP  ;UNABLE TO OPEN
003200          LDA     DTIN    ;IS INPUT DEVICE A DISK
003300          CPI     6
003400          JNZ     CHRIN   ;NO - DO INPUT BY CHARACTERS
003500          LXI     H,BLKGC ;SET UP GC FOR
003510 ;¬                          BLOCK INPUT ROUTINE
003600          SHLD    GCROUT
003700          CALL    IO      ;SET U  BLOCK GET POINTERS
003800          DW      BLGCRB
003900          JMP     SETO    ;GO SET UP OUTPUT
004000 CHRIN    LXI     H,CHRGC ;USE CHRGC ROUTINE
004100          SHLD    GCROUT
004200 ;
004300 ;SET UP OUTPUT FILE
004400 ;
004500 SETO     CALL    MSG     ;TELL OPR WHATS GOING ON
004600          DW      SETUOU
004700          CALL    DTYP    ;DEVICE TYPE
004800          STA     DTOU
004900          CALL    DN      ;DEVICE NUMBER
005000          STA     DNOU
005100          CALL    ASK     ;FILE NAME
005200          DW      ASFNM
005300          LXI     D,FNOU  ;PUT IT AWAY
005400          CALL    MOV8
005500          CALL    IO      ;OPEN FILE
005600          DW      RBOUOP
005700          LDA     STOUOP  ;CHECK STATUS
005800          ORA     A
005900          JNZ     NOOUOP  ;UNABLE TO OPEN
006000          LDA     DTOU    ;IS OUTPUT DEVICE DISK
006100          CPI     6
006200          JNZ     CHROU   ;NO DO OUTPUT BY CHAR
006300          LXI     H,BLKPC ;SET UP PC FOR
006310 ;                           BLOCK PUT ROUTINE
006400          SHLD    PCROUT
006500          CALL    IO      ;SET UP BLOCK PUT POINTERS
006600          DW      BLPCRB
006700          JMP     MINIT   ;GO DO MISC INIT
006800 CHROU    LXI     H,CHRPC ;SET UP OUTPUT BY CHAR
006900          SHLD    PCROUT
007000 ;
007100 ;MISC INIT
007200 ;
007300 MINIT    CALL    ILD     ;INPUT LEADER
007400          CALL    OLD     ;OUTPUT LEADER
```

The copy loops call the get character and put character routines to copy binary bytes or ASCII coded characters.

```
007500 ;
007600 ;MAIN COPY LOOPS
007700 ;
007800          LDA     FNIN    ;GET FILE TYPE
007900          CPI     "&"     ;EDIT SOURCE?
008000          JZ      ASCCOP  ;YES - IS ASCII FILE
008100          CPI     "$"     ;EDIT BACKUP FILE?
008200          JZ      ASCCOP  ;YES - IS ASCII FILE
008300          CPI     "%"     ;LISTING FILE?
008400          JZ      ASCCOP  ;YES - IS ASCII FILE
008500 ;                         ;NO - MUST BE BINARY
008600 ;
008700 ;BINARY COPY LOOP
008800 ;
008900 BINCL1   MVI     B,15    ;SET COUNTER
009000 BINCLP   CALL    GC      ;GET CHARACTER
009100          DW      BINEOF  ;EOF ROUTINE
009200          CALL    PC      ;PUT BINARY BYTE
009300          CPI     0377    ;RUBOUT?
009400          JNZ     BINCL1  ;NO - RESET COUNTER & LOOP
009500          DCR     B       ;ONE LESS RUBOUT TO GO
009600          JZ      EXIT    ;ALL DONE
009700          JMP     BINCLP  ;LOOP
009800 BINEOF   MVI     B,15    ;ADD RUBOUT EOF MARKER
009900          MVI     A,0377  ;RUBOUT
010000 BINEO1   CALL    PC      ;OUTPUT RUBOUT
010100          DCR     B       ;ONE LESS TO GO
010200          JNZ     BINEO1  ;LOOP IF NOT DONE
010300          JMP     EXIT    ;ALL DONE
010400 ;
010500 ;ASCII COPY
010600 ;
010700 ASCCOP   LDA     DTOU    ;CHECK DEVICE TYPE
010800          CPI     6       ;IS IT FDS
010900          JNZ     ASCCL2  ;NO - MUST EXPAND CTL I,ETC.
011000 ASCCL1   CALL    GC      ;GET CHARACTER
011100          DW      ASCEOF  ;EOF ROUTINE
011200          CALL    PC      ;OUTPUT ASC CHAR TO DISK,
011210 ;                          NO TAB EXPAND
011300          CPI     032     ;IS CHAR CTL Z
011400          JZ      EXIT    ;YES - ALL DONE
011500          JMP     ASCCL1  ;NO LOOP
011600 ASCEOF   MVI     A,032   ;ADD CTL Z TO FILE
011700          CALL    PC      ;OUTPUT IT
011800          JMP     EXIT    ;ALL DONE
011900 ASCCL2   CALL    GC      ;GET CHARACTER
012000          DW      ASCEOF  ;EOF ROUTINE
012100          STA     DAPC2   ;PUT CHAR AWAY
012200          CALL    IO      ;OUTPUT IT
```

```
012300          DW      RBPC2
012400          CPI     032     ;IS CHAR CTL Z?
012500          JZ      EXIT    ;YES - ALL DONE
012600          JMP     ASCCL2  ;NO LOOP
```

Get character uses block input Monitor Calls to read data from the input file. The routine checks for input errors and end-of-file marks.

```
012700 ;
012800 ;GET CHARACTER ROUTINES
012900 ;
013000 GC       PUSH    H       ;SAVE [H,L]
013100          LHLD    GCROUT  ;GET ADDRESS OF ROUTINE TO USE
013200          PCHL            ;JUMP TO IT
013300 GCNWBL   CALL    IO      ;SET UP POINTERS FOR NEW BLOCK
013400          DW      BLGCRB
013500          LDA     BLGCST  ;CHECK STATUS
013600          CPI     025     ;IS IT EOF
013700          POP     H       ;RESTORE [H,L]
013800          JZ      BDEX    ;TAKE EOF EXIT
013900          PUSH    H       ;SAVE [H,L]
014000          ORA     A       ;ANY ERRORS
014100          JNZ     ABORT   ;YES - BAIL OUT
014200 BLKGC    LHLD    BLGCCP  ;GET POINTER TO
014210 ;                         NUMBER OF BYTES LEFT
014300          MOV     A,M     ;GET NBR BYTES LEFT
014400          ORA     A
014500          JZ      GCNWBL  ;IS ZERO MUST GET ANOTHER BLOCK
014600          DCR     M       ;ONE LESS
014700          LHLD    BLGCDP  ;GET POINTER TO DATA
014800          MOV     A,M     ;GET DATA
014900          INX     H       ;ADVANCE POINTER
015000          SHLD    BLGCDP  ;PUT POINTER AWAY
015100          POP     H       ;RESTORE [H,L]
015200          JMP     GDEX    ;TAKE NORMAL EXIT
015300 CHRGC    POP     H       ;RESTORE [H,L]
015400          CALL    IO      ;GET CHARACTER
015500          DW      RBGC    ;CHECK STATUS
015600          LDA     STGC
015700          CPI     025     ;EOF?
015800          JZ      BDEX    ;YES
015900          ORA     A       ;ERROR STATUS
016000          JNZ     ABORT   ;YES - BAIL OUT
016100          LDA     DAGC
016200          JMP     GDEX
```

Put character uses block output Monitor Calls to write data into the output file.

```
016300 ;
016400 ;PUT CHARACTER ROUTINES
016500 ;
016600 PC       PUSH    H       ;SAVE [H,L]
016700          LHLD    PCROUT  ;GET ADDRESS OF ROUTINE TO USE
```

```
016800          PCHL                ;JUMP TO IT
016900 BLKPC    PUSH    PSW         ;SAVE DATA
017000          LHLD    BLPCCP      ;POINTER TO NUMBER
017010 ;                             OF BYTES LEFT IN BUFFE
017100          MOV     A,M         ;GET NUMBER OF BYTES LEFT
017200          ORA     A           ;IS IT ZERO?
017300          JNZ     BLKPCS      ;NO STUFF BYTE
017400          CALL    IO          ;SET UP POINTERS FOR NEW BLOCK
017500          DW      BLPCRB
017600          LDA     BLPCST      ;CHECK STATUS
017700          ORA     A
017800          JNZ     ABORT       ;NO GOOD - BAIL OUT
017900 BLKPCS   DCR     M           ;ONE LESS BYTE
018000          LHLD    BLPCDP      ;GET POINTER TO DATA
018100          POP     PSW         ;RESTORE DATA
018200          MOV     M,A         ;PUT DATA IN BUFFER
018300          INX     H           ;ADVANCE POINTER
018400          SHLD    BLPCDP      ;PUT POINTER AWAY
018500          POP     H           ;RESTORE [H,L]
018600          RET                 ;ALL DONE
018700 CHRPC    POP     H           ;RESTORE [H,L]
018800          PUSH    PSW         ;SAVE CHARACTER
018900          STA     DAPC        ;STORE CHARACTER
019000          CALL    IO          ;OUTPUT IT
019100          DW      RBPC
019200          LDA     STPC        ;CHECK STATUS
019300          JNZ     ABORT
019400          POP     PSW         ;RESTORE CHARACTER
019500          RET                 ;ALL DONE
019600 ;
019700 ;TAKE CARE OF LEADER
019800 ;
019900 ILD      RET                 ;***
020000 OLD      RET                 ;***
020100 ;
020200 ;ERROR BAILOUTS
020300 ;
020400 NOINOP   CALL    MSG
020500          DW      MSNOIN
020600          JMP     ABORT
020700 NOOUOP   CALL    MSG
020800          DW      MSNOOU
020900          JMP     ABORT
021000 MSNOIN   DB      015
021100          DB      012
021200          DC      "INPUT FILE OPEN ERROR"
021300 MSNOOU   DB      015
021400          DB      012
021500          DC      "OUTPUT FILE OPEN ERROR"
```

The following Request Control Blocks correspond to COP's Monitor Calls.

```
021600 ;
021700 ;OPEN INPUT FILE REQUEST BLOCK
021800 ;
021900 ;OPEN W/ ERROR MSG SUPPRESSION
022000 RBINOP  DB      0104+0200
022100 STINOP  DS      1         ;STATUS
022200         DB      1         ;FIL NBR
022300         DB      1+0200    ;SEQ IN,EXP DEV
022400 DTIN    DS      1         ;DEV TYPE
022500 DNIN    DS      1         ;DEV NBR
022600         DW      FNIN      ;PTR TO FILE NAME
022700 FNIN    DS      8         ;FILE NAME
022800 ;
022900 ;OPEN OUTPUT FILE REQUEST BLOCK
023000 ;
023100 ;OPEN W/ ERROR MSG SUPPRESSION
023200 RBOUOP  DB      0104+0200
023300 STOUOP  DS      1         ;STATUS
023400         DB      2         ;FILE NBR
023500         DB      2+0200    ;SEQ OUT,EXP DEV
023600 DTOU    DS      1         ;DEVICE TYPE
023700 DNOU    DS      1         ;DEV NUMBER
023800         DW      FNOU      ;PTR TO FILE NAME
023900 FNOU    DS      8         ;FILE NAME
024000 ;
024100 ;CHARACTER GET REQUEST BLOCK
024200 ;
024300 RBGC    DB      2         ;CHRGET
024400 STGC    DS      1         ;STATUS
024500         DB      1         ;FILE NBR
024600 DAGC    DS      1         ;DATA
024700 ;
024800 ;CHARACTER PUT REQUEST BLOCK
024900 ;
025000 RBPC    DB      3         ;CHRPUT
025100 STPC    DS      1         ;STATUS
025200         DB      2         ;FILE NBR
025300 DAPC    DS      1         ;DATA
025400 ;
025500 ;REQUEST BLOCK TO SET UP CHRGET POINTERS INTO D
025600 ;
025700 BLGCRB  DB      0107      ;SET UP BLK GET POINTERS
025800 BLGCST  DS      1         ;STATUS BYTE
025900         DB      1         ;INPUT FILE NUMBER
026000 BLGCCP  DS      2         ;POINTER TO NUMBER
026010 ;                          LEFT IN BLOCK
026100 BLGCDP  DS      2         ;POINTER TO DATA
026200         DS      2         ;RESERVED FOR MONITOR
026300 ;
026400 ;REQUEST BLOCK TO SET UP CHRPUT POINTERS INTO D
```

```
026500 ;
026600 BLPCRB   DB      0110    ;SET UP BLK PUT POINTERS
026700 BLPCST   DS      1       ;STATUS BYTE
026800          DB      2       ;OUTPUT FILE NBR
026900 BLPCCP   DS      2       ;POINTER TO SPACE
026910 ;                         LEFT IN BLOCK
027000 BLPCDP   DS      2       ;POINTER TO DATA
027100          DS      2       ;RESERVED FOR MONITOR
027200 ;
027300 ;CHAR PUT W/ TAB EXPANSIION
027400 ;
027500 RBPC2    DB      0103    ;WRITE
027600          DS      1       ;STATUS
027700          DB      2       ;OUTPUT FILE NUMBER
027800          DB      0       ;ASCII
027900          DW      DAPC2   ;PTR TO BUFFER
028000          DW      1       ;SIZE OF BUFFER
028100          DS      2       ;NUMBER TRANSFERED
028200 DAPC2    DS      1       ;DATA
028300 ;
028400 ;MISC
028500 ;
028600 GCROUT   DS      2       ;ADDRESS OF GC ROUTINE TO USE
028700 PCROUT   DS      2       ;ADDRESS OF PC ROUTINE TO USE
028800 COPID    DB      015     ;CR
028900          DB      012     ;LF
029000          DC      "COPY FILE"
```

The following are messages for the dialog with the operator.

```
029100 ASFNM    DB      015
029200          DB      012
029300          DC      "ENTER FILE NAME "


029400 SETUIN   DB      015
029500          DB      012
029600          DC      "SET UP INPUT"


029700 SETUOU   DB      015
029800          DB      012
029900          DC      "SET UP OUTPUT"


030000          END     COP
```

2SIO

Load Sense Switches 2 stop bits - none up
1 stop bit - A8 up

Bootstrap Loader

Octal Address	Octal Data
ØØØ	Ø76
ØØ1	ØØ3
ØØ2	323
ØØ3	Ø2Ø
ØØ4	Ø76
ØØ5	ØXX (XX = 21 for 2 stop bits, 25 for 1 stop bit)
ØØ6	323
ØØ7	Ø2Ø
Ø1Ø	Ø41
Ø11	3Ø2
Ø12	Ø77
Ø13	Ø61
Ø14	Ø32
Ø15	ØØØ
Ø16	333
Ø17	Ø2Ø
Ø2Ø	Ø17
Ø21	32Ø
Ø22	333
Ø23	Ø21
Ø24	275
Ø25	31Ø
Ø26	Ø55
Ø27	167
Ø3Ø	3ØØ
Ø31	351
Ø32	Ø13
Ø33	ØØØ

PIO

Load Sense Switches A10, A8 - up

Bootstrap Loader

Octal Address	Octal Code
ØØØ	Ø41
ØØ1	3Ø2
ØØ2	Ø77
ØØ3	Ø61
ØØ4	Ø23
ØØ5	ØØØ
ØØ6	333
ØØ7	ØØ4
Ø1Ø	346
Ø11	ØØ1
Ø12	31Ø
Ø13	333
Ø14	ØØ5
Ø15	275
Ø16	31Ø
Ø17	Ø55
Ø2Ø	167
Ø21	3ØØ
Ø22	351
Ø23	ØØ3
Ø24	ØØØ

SIO

Load Sense Switches A9 - up

Bootstrap Loader

Octal Address	Octal Data
ØØØ	Ø41
ØØ1	3Ø2
ØØ2	Ø77
ØØ3	Ø61
ØØ4	Ø22
ØØ5	ØØØ
ØØ6	333
ØØ7	ØØØ
Ø1Ø	Ø17
Ø11	33Ø
Ø12	333
Ø31	ØØ1
Ø14	275
Ø15	31Ø
Ø16	Ø55
Ø17	167
Ø2Ø	3ØØ
Ø21	351
Ø22	ØØ3
Ø23	ØØØ

ACR

Load Sense Switches A9, A8 - up

Bootstrap Loader

Octal Address	Octal Data
ØØØ	Ø41
ØØ1	3Ø2
ØØ2	Ø77
ØØ3	Ø61
ØØ4	Ø22
ØØ5	ØØØ
ØØ6	333
ØØ7	ØØ6
Ø1Ø	Ø17
Ø11	33Ø
Ø12	333
Ø13	ØØ7
Ø14	275
Ø15	31Ø
Ø16	Ø55
Ø17	167
Ø2Ø	3ØØ
Ø21	351
Ø22	ØØ3

Ø23	ØØØ

4PIO

Load Sense Switches A 10 - up

Bootstrap Loader

Octal Address	Octal Data
ØØØ	257
ØØ1	323
ØØ2	Ø4Ø
ØØ3	323
ØØ4	Ø41
ØØ5	Ø76
ØØ6	Ø54
ØØ7	323
Ø1Ø	Ø4Ø
Ø11	Ø41
Ø12	3Ø2
Ø13	Ø77
Ø14	Ø61
Ø15	Ø33
Ø16	ØØØ
Ø17	333
Ø2Ø	Ø4Ø
Ø21	ØØ7
Ø22	33Ø
Ø23	333
Ø24	Ø41
Ø25	275
Ø26	31Ø
Ø27	Ø55
Ø3Ø	167
Ø31	3ØØ
Ø32	351
Ø33	Ø14
Ø34	ØØØ

Microsoft CP/M BASIC·

Addendum to Microsoft BASIC Manual
for Users of CP/M Operating Systems

A CP/M version of BASIC (ver 4.5) is now available from Microsoft. This version of BASIC is supplied on a standard size 3740 single density diskette. The name of the file is MBASIC.COM. To run MBASIC, bring up CP/M and type the following:

```
A>MBASIC <carriage return>
```

The system will reply:

```
xxxx Bytes Free
BASIC Version 4.5
(CP/M Version)
Copyright 1977  (C) by Microsoft
Ok
```

You are now ready to use MBASIC. MBASIC is identical to Altair Disk BASIC version 4.1, with the following exceptions:

1. MBASIC requires 17K of memory.(A 28K or larger CP/M system is recommended).

2. The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to CP/M. The format of the command line is:

   ```
   A>MBASIC [<filename>] [/F:<number of files>]
          [/M:<highest memory location>]
   ```

   Items enclosed in brackets are optional.

   If <filename> is present, MBASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement (see below) to return to CP/M when they have finished, allowing the next program in the batch stream to execute.

   If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is

omitted, the number of files defaults to 3.

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

NOTE

Both <number of files> and <highest memory location> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

A>MBASIC PAYROLL.BAS	Use all memory and 3 files, load and execute PAYROLL.BAS.
A>MBASIC INVENT/F:6	Use all memory and 6 files, load and execute INVENT.BAS.
A>MBASIC /M:32768	Use first 32K of memory and 3 files.
A>MBASIC DATACK/F:2/M:&H9000	Use first 36K of memory, 2 files, and execute DATACK.BAS

3. The DSKF function is not supported by MBASIC. Use CP/M STAT.

4. The FILES statement in MBASIC takes the form FILES[<filename>]. If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the file name or extension will match any file or any extension.

   Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*
FILES "TEST?.BAS"
```

5. The LOF(x) function returns the number of records present in the last extent read or written (usually by a PUT or GET).

6. CSAVE and CLOAD are not implemented.

7. LLIST and LPRINT assume a 132 character wide printer and write their output to the CP/M LST: device.

8. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is used.

9. Filenames themselves follow the normal CP/M naming conventions.

10. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than nine characters long.

11. The error messages "DISK NOT MOUNTED", "DISK ALREADY MOUNTED", "OUT OF RANDOM BLOCKS", and "FILE LINK ERROR" are not included in MBASIC.

12. The CONSOLE statement is not included.

13. To return to CP/M use the SYSTEM command or statement. SYSTEM closes all files and then performs a CP/M warm start. Control-C always returns to MBASIC, not to CP/M.

14. If you wish to change diskettes during MBASIC operation, use RESET. RESET closes all files and then forces CP/M to re-read all diskette directory information. Never remove diskettes while running MBASIC unless you have given a RESET command. The RESET statement takes the place of the MOUNT and UNLOAD statements in Altair BASIC.

15. MBASIC will operate properly on both Z-80 and 8080 systems.

16. MBASIC does not use any of the restart (RST) instruction vectors.

17. The FRCINT routine is located at 103 hex and the MAKINT routine at 105 hex (add 1000 hex for ADDS versions). These routines are used to convert the argument to an integer for assembly language subroutines.

18. If the LEFT$ or RIGHT$ string functions have zero as the number of characters argument, they will return the null (length zero) string.

19. The ERR() Disk error function is not supported as CP/M handles all disk error recovery.

20. Control-H (backspace) deletes the last character typed and is echoed to the terminal.

21. RESTORE <line number> may now be used to set the DATA pointer to a specific line.

22. All error messages and prompts are printed with lower case characters when appropriate.

23. Control-S may be used to cause program execution to pause. In the suspended execution state, control-C will cause a return to BASIC's command level, and any other character will cause the program to resume execution.

24. The EOF function may be used with random files. If a GET is done past end of file, EOF will = -1. This may be used to find the size of a file using a binary search or other algorithm.

25. LSET/RSET may be used on any string. The previous restriction to FIELDed strings has been eliminated.

26. The string function INPUT$(<number of characters> [,[#]<file number>]) may be used to read <number of characters> from either the console or a disk file. If the console is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt execution of the INPUT$ function.

27. VARPTR(#<file number>) returns the address of the disk data buffer for file <file number>.

BASIC Reference Manual

Addenda, April, 1977

1. Page 33, sub-paragraph b:

   LINE INPUT ["<prompt string>",]; <string variable name>

   CHANGE TO:

   LINE INPUT ["<prompt string>";] <string variable>

2. Page 40, Paragraph 5-3b, line 9:

   The of the <integer expression> is the starting address of . . .

   CHANGE TO:

   The <integer expression> is the starting address of . . .

3. Page 41. Insert the following paragraphs between Paragraphs 3 and 4.

   ADDITION:

   The string returned by a call to USR with a string argument is that string the user's routine sets up in the descriptor. Modifying [D,E] does not affect the returned string. Therefore, the statement:

   C$=USR(A$)

   results in A$ also being set to the string assigned to C$. To avoid modifying A$ in this statement, we would use:

   C$=USR(A$+" ")

   so that the user's routine modifies the descriptor of a string temporary instead of the descriptor for A$.

   A string returned by a user's routine should be completely within the bounds of the storage area used by the original string. Increasing a string's length in a user routine is guaranteed to cause problems.

4. Page 49, last paragraph, line 7:

   . . . leading $ signs, nor can negative numbers be output unless the sign is forced to be trailing.

   CHANGE TO:

   . . . leading $ signs.

5. Page 59, last line:

520 CLOSE #1

CHANGE TO:

520 CLOSE 1

6. Page 70, CLEAR [<expression>] explanation:

Same as CLEAR but sets string space to the value . . .

CHANGE TO:

Same as CLEAR but sets string space (see 4-1) to the value . . .

7. Page 70, CLOAD <string expression> explanation, second line:

. . . character of STRING expression> to be . . .

CHANGE TO:

. . . character of <STRING expression> to be . . .

8. Page 71:

CSAVE*<array name> 8K (cassette), Disk

CHANGE TO:

CSAVE*<array name> 8K (cassette), Extended, Disk

9. Page 75. Insert the following after LET and before LPRINT.

ADDITION:

LINE INPUT LINE INPUT "prompt string"; string variable name

Extended, Disk

LINE INPUT prints the prompt string on the terminal and assigns all input from the end of the prompt string to the carriage return to the named string variable. No other prompt is printed if the prompt string is omitted. LINE INPUT may not be edited by Control/A.

10. Page 76, POKE explanation, second line:

. . . If I is negative, address is 65535+I, . . .

CHANGE TO:

. . . If I is negative, address is 65536+I, . . .

11. Page 80, OCT$:

OCT$ OCT$(X) 8K, Extended, Disk

CHANGE TO:

OCT$ OCT$(X) Extended, Disk

12. Page 81:

SPACE$ SPACE$(I) 8K, Extended, Disk

CHANGE TO:

SPACE$ SPACE$(I) Extended, Disk

13. Page 91, line 4:

. . . question (see Appendix E).

CHANGE TO:

. . . question (see Appendix H).

14. Page 95, first paragraph, line 3:

. . . For instructions on loading Disk BASIC, see Appendix E.

CHANGE TO:

. . . For instructions on loading Disk BASIC, see Appendix H.

15. Page 103, line 11:

C (in extended) retains CONSOLE function.

CHANGE TO:

C (in Extended and Disk) retains CONSOLE and all other functions.

16. Page 112, Paragraph 4, Line 3:

USRLOC for 4K and 8K Altair BASIC version 4.0 is 111 decimal.

CHANGE TO:

USRLOC for 4K and 8K Altair BASIC version 4.0 is 111 octal.

17. Page 114, third paragraph, line 2:

. . . by the first character of the STRING expression>.

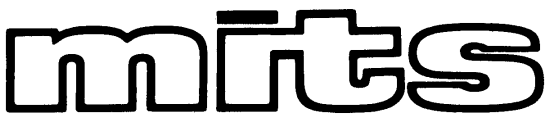
CHANGE TO:

. . . by the first character of the <string expression>. Note that the program named A is saved by CSAVE"A".

18. Index, line 12:

ADDITION:

NULL 72



2450 Alamo SE
Albuquerque, NM 87106